9

Labels and Legends

In this chapter, we discuss annotation of lattice displays by adding labels and legends. As usual, there are various levels of control available to the user, with corresponding differences in the amount of work involved. Most common needs for annotation are satisfied by various labels giving descriptive names for the variables and titles for the entire plot. Legends are usually needed to explain the correspondence between varying graphical parameters such as color, plotting character, and so on, and the quantitative information they represent.

9.1 Labels

Most high-level lattice functions allow four standard labels: `main`, `sub`, `xlab`, and `ylab`. Apart from their positions, they are treated identically for the most part.[1] They can be specified as a character string, as an expression (in which case they are interpreted as LaTeX-like markup, see `?plotmath`), or as a list.[2] In the first two cases, the string or expression is used as the label. The label can be a vector, in which case the components are evenly spread out (this allows row- or column-specific labels). In the third case, when `xlab`, `ylab`, and so on, are lists, the label can be specified as the `label` component. Other components, usually graphical parameters, but possibly ones controlling placement, are passed on to the grid function `textGrob()` to construct a suitable label. The `label` component can be omitted from the list, in which case the default label is used.

By default, `main` and `sub` are omitted in most displays, and `xlab` and `ylab` default to something appropriate, usually the expression for the corresponding variable in the formula, except when they are factors, in which case the label is omitted. Type

[1] `cloud()` and `wireframe()` interpret `xlab` and `ylab` differently, and allow a `zlab`.

[2] For more flexibility, they can also be specified as an arbitrary grob.

```
> demo("labels", package = "lattice")
```

to see some usage examples.

9.2 Legends

Legends, also called keys, usually serve to clarify the meaning of different graphical parameters (symbols, colors, etc.) used in a graphic. They are particularly important in grouped displays (where data from different groups are superposed within panels) and displays where a color gradient encodes a numeric variable (e.g., false-color level plots of three-dimensional surfaces). Legends can also be useful in other contexts; common examples are ones identifying orientation or scale in maps.

In some ways, legends are a weak point in the Trellis design. In the uses described above, as in most other uses, a legend describes features of the display created by the panel function. However, the Trellis model of separating the control of different elements of a display does not include any formal mechanism for direct communication between the processes controlling the panel display and the legend. Consequently, the only general approach that allows useful legends to be created automatically is to have both processes draw from a common source of information. For the collection of high-level functions built into the lattice package, this works reasonably well through the use of the `auto.key` and `colorkey` arguments. To understand these arguments though, we must first discuss the underlying processes that generate legends.

9.2.1 Legends as grid graphical objects

Although this fact is not overly emphasized in this book, the lattice package uses the low-level tools provided by the grid package to do all rendering. This choice is nowhere as important as it is in the context of legends. grid allows the creation of sophisticated "graphical objects" (grobs) that can not only be plotted, but also queried to determine their width and height. This is important in order to allocate the right amount of space for them, especially for legends, because they may have quite arbitrary structure. For full generality, legends in a lattice plot can be specified as arbitrary grobs. For most purposes, it suffices to use the predefined functions `draw.key()` and `draw.colorkey()`, which both produce specialized and highly structured grobs of a certain kind. As we soon show, the user needs no knowledge of grid or grobs to use these functions.

The `draw.key()` function

The `draw.key()` function accepts an argument called `key` and returns a grob.[3] The grob represents a legend containing a series of components laid out in the

[3] It can also draw the grob, a fact we use to create Figure 12.1.

form of a table, possibly divided into multiple blocks. The components can be text, points, lines, or rectangles. These can appear in an arbitrary order, and each component can be repeated or be completely absent. The legend can also have a title.

All this can be achieved through the `key` argument, which must be a list. All its components must be named, of which the names `text`, `points`, `lines`, and `rectangles` may be repeated. Each component named `text` contributes a column of text in the legend, each component named `points` contributes a column of points, and so on, in the order in which they appear in `key`. Each of these components must be lists, containing zero or more graphical parameters specified as vectors. The only special cases are the `text` components, which must have a vector of character strings or expressions as their first component.

Graphical parameters are usually specified as components of the `text`, `points`, `lines`, and `rectangles` lists. They can also be specified directly as components of `key`, but with lower precedence. Valid graphical components are `cex`, `col`, `lty`, `lwd`, `font`, `fontface`, `fontfamily`, `pch`, `adj`, `type`, `size`, `angle`, and `density`, although not all of these apply to all components. Most of these parameters are standard, with the following exceptions.

`adj`
: This parameter controls justification of text. Meaningful values are between 0 (left justified) and 1 (right justified).

`type`
: This parameter is only relevant for lines; `"l"` results in a plain line, `"p"` produces a point, and `"b"` and `"o"` produce both together.

`size`
: This parameter determines the width of rectangles and lines in character widths.

`angle`, `density`
: These parameters are included for compatibility with S-PLUS code, but are currently unimplemented. They are intended to control the details of cross-hatching in rectangles.

Unless otherwise specified (see `rep` below), it is assumed that all columns (except the `text` ones) will have the same number of rows. This common number is taken to be the largest of the lengths of the graphical components, including the ones specified directly in `key`. For a `text` component, the number of rows is the length of its first component, which must be a character or expression vector. Several other components of `key` affect the final legend, as described next.

`rep`
: This can be a scalar logical, defaulting to `TRUE`, in which case all non-text columns in the key are replicated to be as long as the longest. This can be suppressed by specifying `rep = FALSE`, in which case the length of each column will be determined by components of that column alone.

`divide`
: When `type` is `"b"` or `"o"` in a `lines` component, each line is divided by these many point symbols.

`title`
: A character string or expression giving a title for the key.

`cex.title`
: A `cex` factor for the title.

`lines.title`
: Amount of vertical space allocated for the title, in multiples of its own height. Defaults to 2.

`transparent`
: A scalar logical, indicating whether the key area should have a transparent background. Defaults to `FALSE`, but see the next entry.

`background`
: The background color for the legend, which defaults to the default background setting. Note that this default is often `"transparent"`, in which case `transparent = FALSE` will have no visible effect.

`border`
: This can either be a color for the border, or a scalar logical. In the latter case, the border color is black if `border = TRUE`, and no border is drawn if it is `FALSE` (the default).

`between`
: This can be a numeric vector giving the amount of blank space (in terms of character widths) surrounding each column. The specified width is split equally on both sides of a column.

`padding.text`
: This indicates how much space (padding) should be left above and below each row containing text, in multiples of the default. This padding is in addition to the normal height of any row that contains text, which is the minimum amount necessary to contain all the text entries.

`columns`
: The name of this parameter is somewhat misleading, because it specifies not the number of columns in the key, but rather the number of column-blocks into which the key is to be divided. Specifically, rows of the key are divided into these many blocks, which are then drawn side by side.

`between.columns`
: Space between column blocks, in addition to `between`.

The `draw.colorkey()` function

The `draw.colorkey()` function is in many ways much simpler. It too accepts an argument called `key`, and produces a grob that represents a color gradient, along with tick marks and labels that provide calibration for the colors. The legend is defined by the following components of `key`.

`space`
: The intended location of the key, possible values being `"left"`, `"right"` (the default), `"top"`, and `"bottom"`. This only affects the grob to the extent that it determines the orientation of the color gradient (vertical in the first two cases, horizontal in the last two) and the location of the tick marks relative to the gradient (always facing "outwards").

`col`
: The vector of colors used in the legend. The number of colors actually shown is one less than `length(at)` (see below); `col` is replicated if it is shorter, and a subset chosen by sampling linearly if longer. The same rule is used by `panel.levelplot()` and `panel.wireframe()` when appropriate.

`at`
: It is always assumed that the colors supplied represent discrete bins along some numeric interval (although the tick mark labels can be manipulated to suggest otherwise). `at` is a numeric vector defining these bins. Specifically, they determine where the colors change, and must be in ascending order. There is no requirement for the `at` values to be equispaced.

`labels`
: This can be a character vector (or expression) for labeling the `at` values, but this use is unusual. More commonly, `labels` is a list, which itself has one or more of the components `at`, `labels`, `cex`, `col`, `font`, `fontface`, and `fontfamily`. This works much as does `scales` (see Chapter 8), in the sense that the `at` and `labels` components, defining the tick mark locations and labels, are determined automatically if unspecified.

`tick.number`
: Suggested number of ticks, used when the tick mark locations are unspecified.

`width`
: A multiplier to control the width, or rather the thickness, of the key. When the key is horizontal (`space` is `"top"` or `"bottom"`), this actually controls the height.

`height`
: One interesting feature of the grobs produced by `draw.colorkey()` is that they are "expandable" in one direction; the color bar does not have an absolute length, but expands to fit in the space available. This component determines what proportion of the available space the legend will occupy. As with `width`, the name of this component is misleading when `space` is `"top"` or `"bottom"`.

9.2.2 The `colorkey` argument

A color gradient as produced by `draw.colorkey()` is only relevant for two high-level lattice functions: `levelplot()` and `wireframe()` (the latter only when `drape = TRUE`). For these functions, the legend can be controlled by the `colorkey` argument. The legend can be suppressed with `colorkey = FALSE`, and enabled with `colorkey = TRUE`, the latter being the default whenever a color gradient is used. Alternatively, `colorkey` can be a list, in which case it is used as the `key` argument in `draw.colorkey()`. The most common use of this is to change the location of the legend, for example, with `colorkey = list(space = "top")`. The only component of `key` without a reasonable default in `draw.colorkey()` is `at`, which in the case of `levelplot()` and `wireframe()` defaults to the corresponding `at` argument in the high-level function. Adding a color key in other high-level functions is possible, but more involved, as we have seen in Figure 5.6.

9.2.3 The `key` argument

Unlike `draw.colorkey()`, which is designed for a fairly specific purpose, `draw.key()` is intended to be quite general. The `key` argument, accepted by all high-level functions (including `levelplot()` and `wireframe()`), allows legends produced by `draw.key()` to be added to a plot. Such a `key` argument can be a list as accepted by `draw.key()`, with the following additional components also allowed.

`space`
: This specifies the intended location of the key, possible values being `"left"`, `"right"`, `"top"` (the default), and `"bottom"`.

`x`, `y`, `corner`
: These components specify an alternative positioning of the legend inside the plot region. `x` and `y` determine the location of the corner of the key given by `corner`. Common values of `corner` are `c(0, 0)`, `c(1, 0)`, `c(1,1)`, and `c(0,1)`, which denote the corners of the unit square, but fractional values are also allowed. `x` and `y` should be numbers between 0 and 1, giving coordinates with respect to either the whole display area, or just the subregion containing the panels. The choice is controlled by `lattice.getOption("legend.bbox")`, which can be `"full"` or `"panel"` (the default).

Figure 8.6 gives an example of a simple but nontrivial legend produced using the `key` argument, as does Figure 9.2 later in this chapter. These examples demonstrate the flexibility of `draw.key()`. However, in practice, most legends are associated with a grouping variable, supplied as the `groups` argument. The generality of `draw.key()` is unnecessary for such legends, which typically have exactly one column of text (containing the levels of `groups`), and at most one column each of points, lines, or rectangles. Furthermore, if the different graphical parameters associated with different levels of `groups` are obtained from the global settings, the contents of these columns are also entirely predictable.

One way to create such standard legends is to use the `Rows()` function, which is useful in extracting a subset of graphical parameters suitable for use as a component in key. Consider the Car93 dataset, which contains information on several 1993 passenger car models (Lock, 1993; Venables and Ripley, 2002), and can be loaded using

```
> data(Cars93, package = "MASS")
```

For our first example, we plot the midrange price against engine size, conditioning on AirBags, with Cylinders as a grouping variable. To make things interesting, we leave out the level "rotary", which is represented only once in the data:

```
> table(Cars93$Cylinders)
     3      4      5      6      8 rotary
     3     49      2     31      7      1
```

As the first five levels of Cylinders are plotted, we can extract the corresponding default graphical settings as

```
> sup.sym <- Rows(trellis.par.get("superpose.symbol"), 1:5)
> str(sup.sym)
List of 6
 $ alpha: num [1:5] 1 1 1 1 1
 $ cex  : num [1:5] 0.7 0.7 0.7 0.7 0.7
 $ col  : chr [1:5] "#000000" "#000000" "#000000" ...
 $ fill : chr [1:5] "#EBEBEB" "#DBDBDB" "#FAFAFA" ...
 $ font : num [1:5] 1 1 1 1 1
 $ pch  : num [1:5] 1 3 6 0 5
```

This can now be used in a call to xyplot() to produce Figure 9.1.

```
> xyplot(Price ~ EngineSize | reorder(AirBags, Price), data = Cars93,
         groups = Cylinders, subset = Cylinders != "rotary",
         scales = list(y = list(log = 2, tick.number = 3)),
         xlab = "Engine Size (liters)",
         ylab = "Average Price (1000 USD)",
         key = list(text = list(levels(Cars93$Cylinders)[1:5]),
                    points = sup.sym, space = "right"))
```

This computation can be simplified using a convenience function called simpleKey(), which returns a list suitable for use as the key argument. The first argument to simpleKey() (text) must be a vector of character strings or expressions, giving the labels in the text column. It can also be given logical arguments points, lines, and rectangles specifying whether a corresponding column will be included in the key; if TRUE, the graphical parameters for the corresponding component are constructed using calls to trellis.par.get() and Rows() as above. The settings "superpose.symbol" is used for points, "superpose.line" for lines, and "superpose.polygon" for rectangles. Further arguments to simpleKey() are simply retained as elements of the list returned. Thus, an alternative call producing Figure 9.1 is

```
> xyplot(Price ~ EngineSize | reorder(AirBags, Price), data = Cars93,
         groups = Cylinders, subset = Cylinders != "rotary",
```

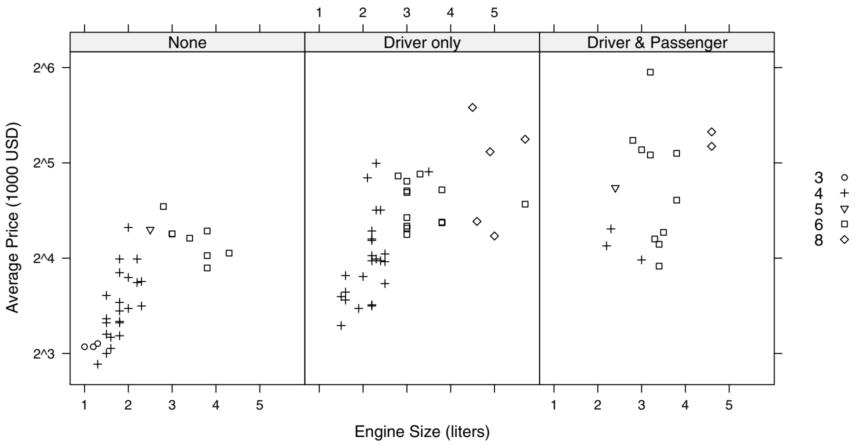

Figure 9.1. Average (of basic and premium) price of cars plotted against engine size. The data are separated into panels representing number of airbags (ordered by mean price), and the number of cylinders is used as a grouping variable within each panel.

```
scales = list(y = list(log = 2, tick.number = 3)),
xlab = "Engine Size (liters)",
ylab = "Average Price (1000 USD)",
key = simpleKey(text = levels(Cars93$Cylinders)[1:5],
                space = "right", points = TRUE))
```

9.2.4 The problem with settings, and the `auto.key` argument

This approach, although appearing to be effective at first glance, breaks down if we consider the possibility of changes in the settings. As we saw in Chapter 7, presentation of a graphic is not entirely defined by its contents; that is, the same *"trellis"* object can be plotted multiple times using different themes, resulting in the use of different graphical parameters. This is especially relevant for grouped displays, where color might be used to distinguish between groups when available, and other parameters such as plotting character and line type used otherwise. This choice is determined by the theme in use when the object is plotted, and thus, it is impossible to determine the legend prior to that point. The problem with the approach described above, using `simpleKey()`, is that the legend is instead determined fully when the object is created.

The solution is to postpone the call to `simpleKey()` until plotting time. This can be achieved through the `auto.key` argument, which can simply be a list containing arguments to be supplied to `simpleKey()`. Thus, yet another call that produces Figure 9.1 is

```
> xyplot(Price ~ EngineSize | reorder(AirBags, Price), data = Cars93,
         groups = Cylinders, subset = Cylinders != "rotary",
         scales = list(y = list(log = 2, tick.number = 3)),
         xlab = "Engine Size (liters)",
         ylab = "Average Price (1000 USD)",
         auto.key = list(text = levels(Cars93$Cylinders)[1:5],
                         space = "right", points = TRUE))
```

This version will update the legend suitably when the resulting object is plotted with different themes. In fact, the `auto.key` approach allows for more intelligent defaults, and it is usually possible to omit the `text` component (which defaults to the group levels) as well as the `points`, `lines`, and `rectangles` components (which have function-specific defaults). One can simply use `auto.key = list(space = "right")` in the above call, or even `auto.key = TRUE` which would use the default `space = "top"`. Unfortunately, in both these cases, the omitted level (`"rotary"`) will be included in the legend.

9.2.5 Dropping unused levels from groups

For conditioning variables and primary variables that are factors, levels that are unused after the application of the `subset` argument in a high-level call are usually omitted from the display. It is difficult to do the same with unused levels of `groups`. This is a consequence of the design; `groups` is passed to the panel function as a whole, and appropriate panel-specific subsets are extracted using the `subscripts` argument. `subscripts` refers to rows in the original data before applying `subset`, and so, `groups` must also be available in its entirety. Dropping levels inside the panel function is not an option, as some levels may be present in some panels, but not in others.

This behavior can sometimes be frustrating, and often the simplest solution is to subset the data beforehand, possibly using the `subset()` function. One more operation is required to omit the unused levels, as `subset()` does not do so itself. In the following call, which is yet another way to produce Figure 9.1, this is done inline when specifying `groups`.

```
> xyplot(Price ~ EngineSize | reorder(AirBags, Price),
         data = subset(Cars93, Cylinders != "rotary"),
         groups = Cylinders[, drop = TRUE],
         scales = list(y = list(log = 2, tick.number = 3)),
         xlab = "Engine Size (liters)",
         ylab = "Average Price (1000 USD)",
         auto.key = list(space = "right"))
```

Many other examples that use `auto.key` can be found throughout this book.

9.2.6 A more complicated example

Although rare, there are nonetheless occasions where `auto.key` is not sufficient. We finish this section with one such example, where two grouping

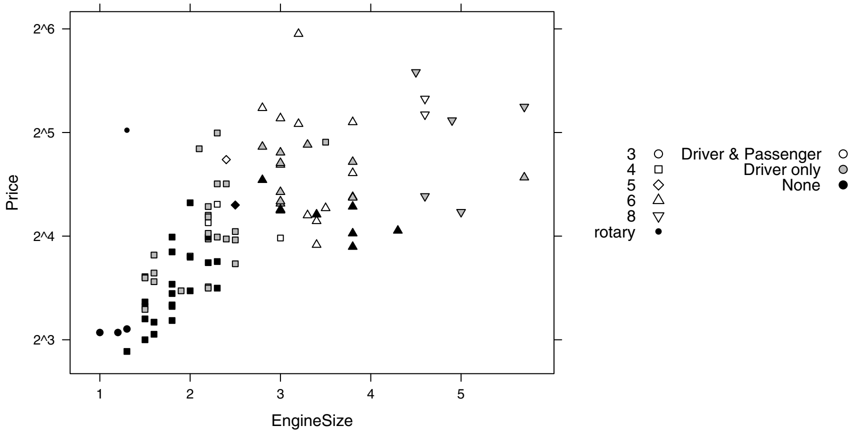

Figure 9.2. An alternative to Figure 9.1, with both `Cylinders` and `AirBags` now used as grouping variables, encoded by different graphical attributes (plotting character and fill color). The associated legend has to be constructed explicitly using the `key` argument.

variables are used concurrently, with levels distinguished by varying two different graphical parameters. In particular, our goal is to produce an alternative form of Figure 9.1, where in addition to `Cylinders`, `AirBags` is also a grouping variable rather than a conditioning variable. Consequently, the legend must contain two columns of text, one for each grouping variable, of different lengths. Figure 9.2 is produced by the following code.

```
> my.pch <- c(21:25, 20)
> my.fill <- c("transparent", "grey", "black")
> with(Cars93,
       xyplot(Price ~ EngineSize,
              scales = list(y = list(log = 2, tick.number = 3)),
              panel = function(x, y, ..., subscripts) {
                  pch <- my.pch[Cylinders[subscripts]]
                  fill <- my.fill[AirBags[subscripts]]
                  panel.xyplot(x, y, pch = pch,
                               fill = fill, col = "black")
              },
              key = list(space = "right", adj = 1,
                         text = list(levels(Cylinders)),
                         points = list(pch = my.pch),
                         text = list(levels(AirBags)),
                         points = list(pch = 21, fill = my.fill),
                         rep = FALSE)))
```

The use of `with()` allows us to refer to elements of `Cars93` by name inside the panel function.

9.2.7 Further control: The `legend` argument

Legends produced by `draw.key()` can be quite general, but they are ultimately limited in scope. The `legend` argument, although more involved in its use, provides far greater flexibility. This flexibility is afforded by the ability to specify the legend as an arbitrary grob, or alternatively a function, called at plotting time, that produces a grob. We give an example illustrating the use of this feature, but do not discuss it in much detail as it is rarely useful to the casual user. Details can be found in the online documentation.

Our example is a heatmap, which is a graphical representation of a hierarchical clustering of rows and/or columns of a matrix. We consider again the `USArrests` dataset, which tabulates the number of arrests for various crimes in 1973 per 100,000 residents in the 50 U.S. states. Our goal is to cluster the states, which can be done with the `hclust()` function.

```
> hc1 <- hclust(dist(USArrests, method = "canberra"))
> hc1 <- as.dendrogram(hc1)
```

We coerce the result to a *"dendrogram"* object before manipulating it further. The next step is to find a permutation of the states that arranges them in the "right" order; there is more than one such permutation, and we determine one that retains grouping by region (given by the `state.region` dataset) as much as possible.

```
> ord.hc1 <- order.dendrogram(hc1)
> hc2 <- reorder(hc1, state.region[ord.hc1])
> ord.hc2 <- order.dendrogram(hc2)
```

We are now almost ready to draw our heatmap. Our first attempt might be

```
> levelplot(t(scale(USArrests))[, ord.hc2])
```

where the states are reordered, each variable is scaled to make the units comparable, and the data matrix is transposed to produce a tall (rather than wide) display. Of course, this will not show the actual clustering, which is where the `legend` argument comes in. The lattice package has no built-in support for plotting dendrograms, but it does allow new legends to be designed and used. The `dendrogramGrob()` function in the latticeExtra package conveniently produces a grob representing a given dendrogram, and can be used as follows to produce Figure 9.3.

```
> library("latticeExtra")
> region.colors <- trellis.par.get("superpose.polygon")$col
> levelplot(t(scale(USArrests))[, ord.hc2],
            scales = list(x = list(rot = 90)),
            colorkey = FALSE,
            legend =
            list(right =
```

```
                  list(fun = dendrogramGrob,
                       args =
                       list(x = hc2, ord = ord.hc2,
                            side = "right", size = 10, size.add = 0.5,
                            add = list(rect =
                              list(col = "transparent",
                                 fill = region.colors[state.region])),
                            type = "rectangle"))))
```

Here, the normal color key is disabled as the units lose their meaning after scaling. Instead, we put in the dendrogram as the legend on the right side. The specification of `legend` is fairly simple in an abstract sense; it is a list with a component `right` indicating that the legend should be placed to the right of the panel(s), which in turn consists of components `fun`, which is a function that returns a grob, and `args`, which is a list of arguments supplied to `fun`. For the interpretation of the arguments provided to `dendrogramGrob()`, see the corresponding help page.

Writing a function such as `dendrogramGrob()` requires familiarity with the grid package.[4] For those interested in traveling that road, `dendrogramGrob()` can serve as a useful template.

9.3 Page annotation

Another form of annotation is available through the `page` argument to a high-level plot, which must be a function that is executed once for every page, with the page number as its only argument. The function must use grid-compliant plotting commands (which include lattice panel functions), and is called with the whole display area as the default viewport and the native coordinate system set to the unit square $[0,1] \times [0,1]$. An obvious use of this argument is to add page numbers to multipage lattice plots; for example, as

```
page = function(n) {
    panel.text(lab = sprintf("Page %d", n), x = 0.95, y = 0.05)
}
```

Such a function could be set as the global default:

```
> lattice.options(default.args = list(page = function(n) {
      panel.text(lab = sprintf("Page %d", n), x = 0.95, y = 0.05)
  }))
```

in which case all subsequent lattice plots would include a page number. Another possible use of `page` is to perform some interactive task after a page is drawn, such as placing a legend by clicking on a location in the display area; an example is shown in Figure 12.1.

[4] In particular, making sure that the legend "expands" to exactly fit the panel, even when the plot is resized, involves the concepts of frames and packing.

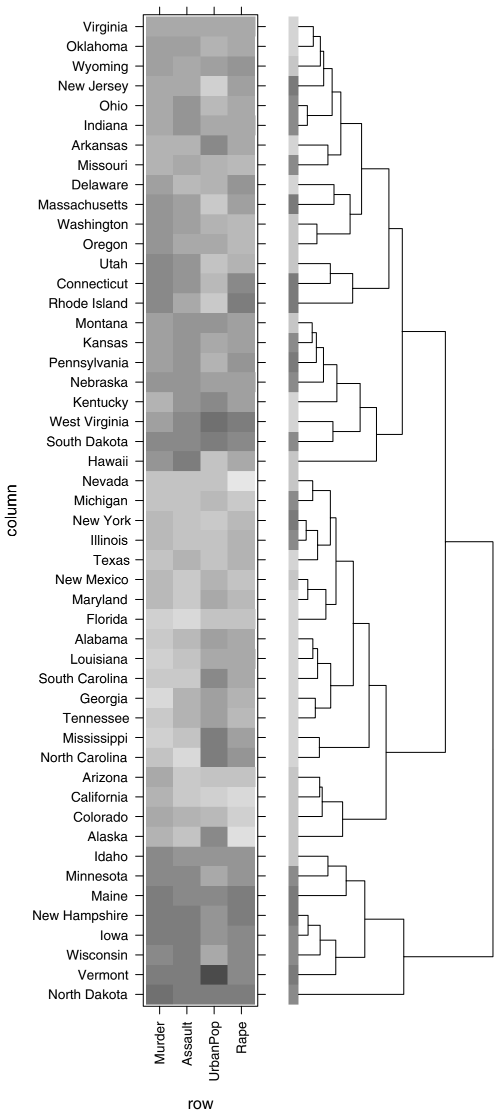

Figure 9.3. A heatmap created with the standard `levelplot()` function along with a nonstandard legend representing a hierarchical clustering. The thin strip at the root of the dendrogram represents a grouping of the states based on geographical location (south, northeast, etc.). Unlike the standard `heatmap()` function, this implementation puts no restrictions on the aspect ratio.