# 7

# Graphical Parameters and Other Settings

In the second part of this book, we take a detailed look at features that are common to all high-level lattice functions, providing a uniform interface to control their output. We start, in this chapter, by describing the system of user settable graphical parameters and other global options.

Graphical parameters are often critical in determining the effectiveness of a plot. Such parameters include obvious ones such as colors, symbols, line types, and fonts for the various elements of a graph, as well as more subtle ones such as the length of tick marks or the amount of space separating different components of the graph. The parameters used in lattice displays are highly customizable. Many of them can be controlled directly by specifying suitable arguments in a high-level function call. Most derive their default values from a system of common global settings that can also be modified by the user. The latter approach has two primary benefits: it allows good global defaults to be specified, and it provides a consistent "look and feel" to lattice graphics while letting the user retain ultimate control.

Not all parameters of interest are graphical. For example, a user may dislike the default argument value `as.table = FALSE` (which orders panels starting from the lower-left corner rather than the upper-left one), and wish to change the default globally rather than specify an additional argument in every call. Several such non-graphical parameters can be customized, through a slightly different system of global options. Both these systems are discussed in this chapter.

## 7.1 The parameter system

In this section, we present some essential background information about the graphical parameter system. The parameters that are actually available for use and their effect are detailed in the next section.

### 7.1.1 Themes

Choosing good graphical parameters is a nontrivial task. For grouped displays in particular, one needs colors, plotting characters, line types, and so on, that mesh well together, yet are distinctive enough that each stands out from the others. Furthermore, a choice of colors that is good for points and lines may not be good as a fill color (e.g., for bar charts). The settings system in lattice allows the user to specify a coherent collection of graphical parameters, presumably put together by an expert,[1] to be used as a common source consistently across all high-level plots. Such collections of parameters are henceforth referred to as *themes*.

Unfortunately, it is even harder to find a single theme that is optimal for all display media, to say nothing of individual tastes. Color is vastly more effective than plotting characters or line types in distinguishing between groups; however, black and white printing is often considerably cheaper. Even when available, a good choice of colors for printing may not be as good for viewing on a computer monitor or an overhead projector as these involve fundamentally different physical mechanisms; colors in print are produced by subtracting basic colors, whereas color on monitors and projectors is produced by adding basic colors. Furthermore, the same specification may produce different actual colors on different hardware, sometimes because of the hardware settings, sometimes simply because of differences in the hardware.

### 7.1.2 Devices

In traditional S graphics, no special consideration was given to the target media. Following the original Trellis implementation in S, lattice attempts to rectify this situation, although not with unqualified success, by allowing graphical settings to be associated with specific devices. As the reader should already know, R can produce graphics on a number of output devices. Each supported platform has a native screen device,[2] as well as several file-based devices. The latter include vector formats such as PDF, Postscript®, SVG (scalable vector graphics), and EMF (on Windows), as well as bitmap formats such as JPEG and PNG. lattice allows a different theme to be associated with each of these devices.

Unfortunately, this does not really solve the problem of settings specific to target media. It is common for PDF documents to be viewed on a screen or projected (especially presentation slides) as well as printed. It is also fairly common practice to print a graphic displayed on the screen using `dev.print()` and related functions, often via a GUI menu, which simply recreates the graph

[1] We do not discuss how one might design an effective collection of settings, as that is beyond the scope of this book. See Ihaka (2003) for a helpful discussion of colors in presentation graphics. The packages RColorBrewer and colorspace provide some useful tools for working with color.

[2] Typically one of `x11`, `quartz`, and `windows`.

without changing the settings to match the target device. Thus, the availability of device-specific settings is only beneficial if the user is disciplined enough, and such settings are possibly confusing for the casual user. For this reason, all devices currently use a common color theme by default, with the exception of `postscript`, which uses a black and white theme. It is possible for the user to associate other themes as the default for specific devices, and a procedure to do so is outlined later in this chapter. Even if one is not interested in device-specific themes, it is helpful to keep the preceding discussion in mind when reading the rest of this section.

### 7.1.3 Initializing a graphics device

In traditional R graphics, devices are initialized by calling the corresponding device function (e.g., `pdf()`, `png()`, etc.). If a plotting function is called with no open device, the one specified by `getOption("device")` (typically the native screen device) is opened automatically. This works for lattice plots as well, as long as one is content using the default theme. However, for finer control over the theme used, it is more convenient to initialize a device through the wrapper function `trellis.device()`. It has the following arguments.

`device`
: This argument determines the device that will be opened (unless `new = FALSE`). It can be specified either as a device function (e.g., `pdf`) or as the name of such a function (e.g., `"pdf"`). By default, `getOption("device")` is used.

`color`
: Every device has a default theme associated with it, which can be modified fully or partially via the `theme` argument described below, or after the device is opened. This default theme can be one of two choices, one color and one black and white. The `color` argument is a logical flag that determines this choice; it defaults to `FALSE` for postscript devices, and to `TRUE` for all others.

`theme`
: This argument allows modifications to the default theme to be specified. Details are given below. This argument defaults to `lattice.getOption("default.theme")`.

`new`
: This is a logical flag indicating whether a new device should be initialized. It only makes sense to set this to `FALSE` when one wishes to reset the currently active device's theme to the settings determined by other arguments. An alternative is to use the `trellis.par.set()` function described later.

`retain`

This is also a logical flag. Once a device is open, its settings can be modified. When another instance of the same device is opened later using `trellis.device()`, the settings for that device are usually reset to its defaults. This can be prevented by specifying `retain = TRUE`. Note that settings for different devices are always treated separately, that is, opening a `postscript()` device does not alter the `pdf()` settings (but it does reset the settings of any `postscript` device that is already open).

A theme (as in the `theme` argument above) can be specified either as a list containing parameter values, or a function that produces such a list when called. The structure of the list is discussed a little later. If `theme` is a function, it will not be supplied any arguments, but the device is guaranteed to be open when it is called, so one may use the `.Device` variable inside the function to ascertain what device has been opened. Note that `theme` only modifies the theme determined by other arguments, and need not contain all possible parameters.

One important difference between calling a device function such as `pdf()` directly, and calling it through `trellis.device()`, is that the latter resets the device theme to the initial defaults, undoing any prior changes (unless `retain = TRUE`). This is often the easiest way to recover from experiments with settings that have gotten out of hand.

### 7.1.4 Reading and modifying a theme

Most elements of a lattice graphic can be controlled by some theme parameter. Of course, one must know which one for this fact to be useful. Once a device is open, the theme associated with it can be queried and modified using the functions `trellis.par.get()` and `trellis.par.set()`. This is best illustrated by an example. Consider Figure 7.1, which gives an alternative visualization of the `VADeaths` dataset, similar to Figure 4.2. The plot is produced by

```
> vad.plot <-
      dotplot(reorder(Var2, Freq) ~ Freq | Var1,
              data = as.data.frame.table(VADeaths),
              origin = 0, type = c("p", "h"),
              main = "Death Rates in Virginia - 1940",
              xlab = "Number of deaths per 100")
> vad.plot
```

Because the absolute rates are encoded by a line "dropping" down to the origin, the light grey reference lines are now somewhat redundant. Let us try, as an exercise, to remove them from the graph. The parameters in a theme are generally identified by names descriptive of their use, and the parameters of the reference line happen to be determined by the settings named `"dot.line"`.

```
> dot.line.settings <- trellis.par.get("dot.line")
> str(dot.line.settings)
```

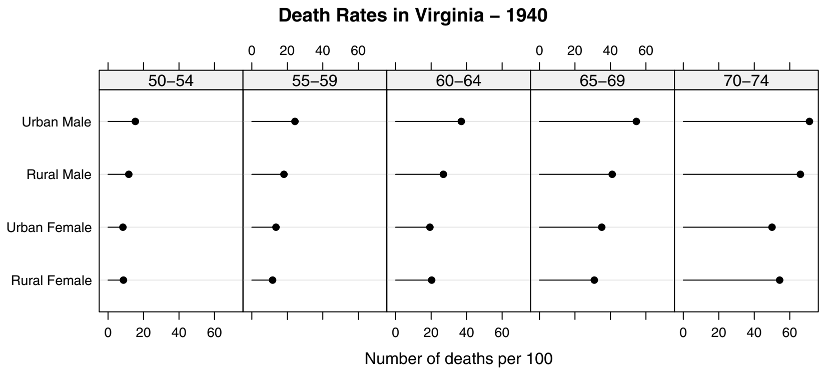

**Figure 7.1.** A dot plot of death rates in Virginia in the year 1940 across population groups, conditioned on age groups. The rates are encoded by length as well as position, through line segments joining the points to the origin.

```
List of 4
 $ alpha: num 1
 $ col  : chr "#E6E6E6"
 $ lty  : num 1
 $ lwd  : num 1
```

As the output of `str()` suggests, the result is a list of graphical parameters[3] that control the appearance of the reference lines. The simplest way to omit the reference lines is to make them transparent. This can be done by modifying the `dot.line.settings` variable and then using it to change the settings:

```
> dot.line.settings$col <- "transparent"
> trellis.par.set("dot.line", dot.line.settings)
```

While we are at it, let us also double the thickness of the lines being shown, whose parameters are obtained from the `"plot.line"` settings:

```
> plot.line.settings <- trellis.par.get("plot.line")
> str(plot.line.settings)
List of 4
 $ alpha: num 1
 $ col  : chr "#000000"
 $ lty  : num 1
 $ lwd  : num 1
```

[3] Where possible, `lattice` follows the standard naming conventions for graphical parameters: `col` for color, `lty` for line type, `lwd` for line width, `pch` for plotting character, and `cex` for character size. Fonts can be controlled by `font`, or `fontface` and `fontfamily` for finer control. In addition, `alpha` is used for partial transparency (often referred to as alpha-channel transparency for historical reasons) on devices that support it. See the `?par` help page for further details, including valid ways to specify color and line type.

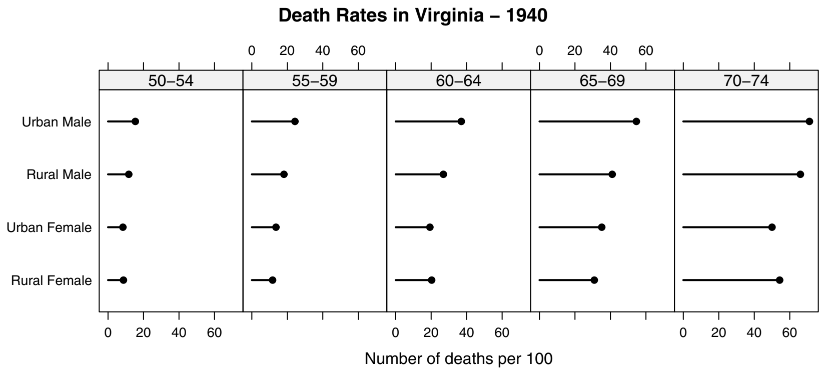

**Figure 7.2.** Death rates in Virginia. An alternative version of Figure 7.1 with a slightly modified theme. The reference lines, which are mostly redundant, have been removed, and the widths of the line segments have been doubled.

```
> plot.line.settings$lwd <- 2
> trellis.par.set("plot.line", plot.line.settings)
```

We can now simply replot the previously saved object to produce Figure 7.2.

```
> vad.plot
```

An alternative solution that does not require the settings to be modified is to write a suitable panel function. Even though this is not necessary in this example, it is instructive as an illustration of how theme parameters might provide defaults in a panel function.

```
> panel.dotline <-
      function(x, y,
               col = dot.symbol$col, pch = dot.symbol$pch,
               cex = dot.symbol$cex, alpha = dot.symbol$alpha,
               col.line = plot.line$col, lty = plot.line$lty,
               lwd = plot.line$lwd, alpha.line = plot.line$alpha,
               ...)
  {
      dot.symbol <- trellis.par.get("dot.symbol")
      plot.line <- trellis.par.get("plot.line")
      panel.segments(0, y, x, y, col = col.line, lty = lty,
                     lwd = lwd, alpha = alpha.line)
      panel.points(x, y, col = col, pch = pch,
                   cex = cex, alpha = alpha)
  }
```

This panel function explicitly draws the line segments and points to create the display. Thanks to lazy evaluation, the default parameters are obtained from the theme active when the panel function is called. This panel function can now be used in a call such as

```
> update(vad.plot, panel = panel.dotline)
```

It is left as an exercise to the reader to verify that the thickness of the lines in the resulting plot depends on whether the "plot.line" settings were modified beforehand.

### 7.1.5 Usage and alternative forms

Both trellis.par.get() and trellis.par.set() apply to the theme associated with the currently active device. trellis.par.get(), called with a name argument, returns the associated parameters as a list. When called without a name argument, it returns the full list of settings. trellis.par.set() can be called analogously with arguments name and value, as shown above. However, this is not its only valid form. More than one parameter can be set at once as named arguments, so the two trellis.par.set() calls earlier can be replaced by the single call

```
> trellis.par.set(dot.line = dot.line.settings,
                  plot.line = plot.line.settings)
```

In fact, the replacements may be "incomplete", in the sense that only components being modified need to be supplied. In other words, the above is equivalent to

```
> trellis.par.set(dot.line = list(col = "transparent"),
                  plot.line = list(lwd = 2))
```

Finally, any number of parameters can be supplied together as a list, for example,

```
> trellis.par.set(list(dot.line = list(col = "transparent"),
                       plot.line = list(lwd = 2)))
```

This last option is a convenient way to specify a complete user-defined theme, that is, a subcollection of parameters that provides an alternative look and feel. This is in fact the form that the theme argument in trellis.device() must take when it is a list, and the same applies to its return value when theme is a function.

### 7.1.6 The par.settings argument

As noted above, trellis.par.set() modifies the current theme. Often, one wants to associate specific parameter values with a particular call rather than globally modify the settings. This can be achieved using the par.settings argument in any high-level lattice call. Whenever the resulting object is plotted, whether immediately or later with a different theme active, these settings are temporarily in effect for the duration of the plot, after which the settings revert to whatever they were before. For example, the following will re-create Figure 7.2 with or without the earlier calls to trellis.par.set().

```
> update(vad.plot,
         par.settings = list(dot.line = list(col = "transparent"),
                             plot.line = list(lwd = 2)))
```

This paradigm is particularly useful, in conjunction with the `auto.key` argument, for grouped displays with non-default graphical parameters. The convenience function `simpleTheme()` can often be used to create a suitable value for `par.settings` with little effort.

## 7.2 Available graphical parameters

As explained in the previous section, the graphical parameter system can be viewed as a collection of named settings, each controlling certain elements in lattice displays. To take advantage of the system, either by modifying themes or by using them as defaults in custom panel functions, the user must know the names and structures of the settings available. The full list is subject to change, but the most current list can always be obtained by inspecting the contents of a theme, for example, using

```
> names(trellis.par.get())
```

Most of these settings have a common pattern: their value is simply a list of standard graphical parameters such as `col`, `pch`, and so on. Figure 7.3 lists these settings along with their component parameters.

These settings can be broadly divided into two types based on their purpose. Some are intended to control elements common to most lattice displays. These include

`par.xlab.text`, `par.ylab.text`, `par.main.text`, `par.sub.text`
    which control the various labels (in addition, `par.zlab.text` controls the $z$-axis label in `cloud()` and `wireframe()`),

`strip.background`, `strip.shingle`, `strip.border`
    which control certain aspects of the strips through the default strip function `strip.default()`, and

`axis.text`, `axis.line`
    which control the appearance of axes.

Other settings are meant for use by panel functions. Some of these have very specific targets; for example,

`box.dot`, `box.rectangle`, `box.umbrella`
    are used by `panel.bwplot()`,

`dot.line`, `dot.symbol`
    are used by `panel.dotplot()`,

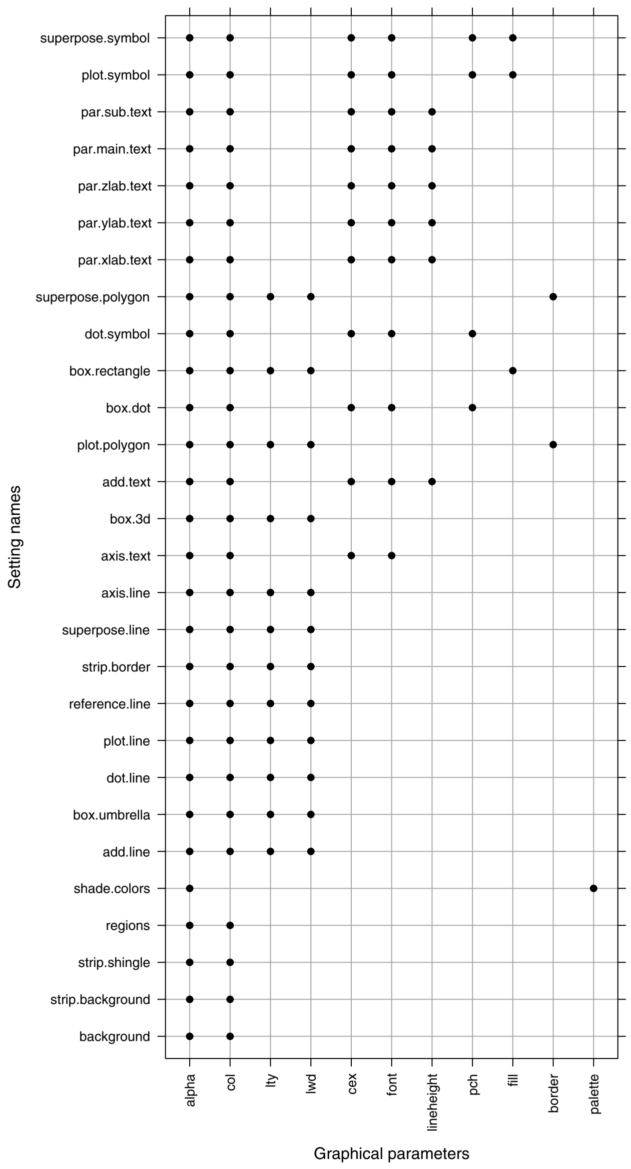

**Figure 7.3.** The standard graphical settings (at the time of writing). Each setting has a specific purpose, and consists of one or more graphical parameters. The parameter names (`col`, `pch`, etc.) follow the usual R conventions for the most part. The `fontface` and `fontfamily` parameters may be used for finer control over fonts wherever `font` is allowed (see `?gpar` in the `grid` package).

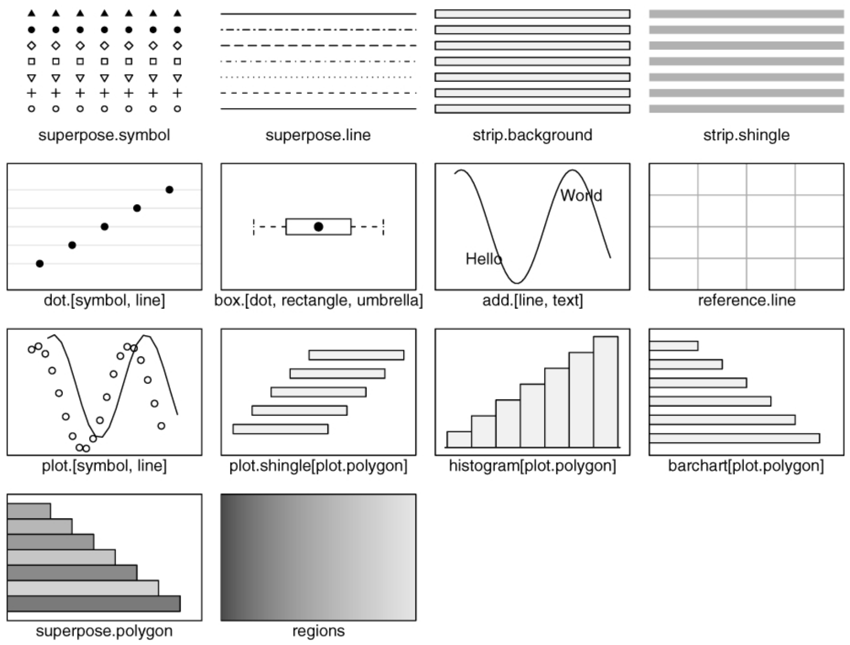

**Figure 7.4.** A graphical summary of the black and white theme used throughout this book, as produced by `show.settings()`. The other primary built-in parameter scheme available in lattice is the color scheme used for the color plates, which is also the default on all screen devices. Most use of color can be justified by one of two purposes; first, to distinguish data driven elements from non-data elements such as axes, labels and reference lines, and second, to distinguish between levels of a grouping variable in superposed displays. In the default black and white theme, the first goal is largely ignored, and the second is achieved using different symbols and line types (and grey levels when necessary). A summary of the default color theme is shown in the color plates.

`plot.line`, `plot.symbol`
    are used (for the most part) by `panel.xyplot()`, `panel.densityplot()`, and `panel.cloud()`,

`plot.polygon`
    is used by `panel.histogram()` and `panel.barchart()`,

`box.3d`
    is used by `panel.cloud()` and `panel.wireframe()`, and

`regions`, `shade.colors`
    are used by `panel.levelplot()` and `panel.wireframe()`.

Other settings are more general purpose. For example,

`superpose.symbol`, `superpose.line`, `superpose.polygon`
: are used for grouped displays in various contexts, whereas

`reference.line`, `add.line`, `add.text`
: are meant for secondary elements in a display, and are used in helper panel functions such as `panel.grid()` and `panel.text()`.

The `show.settings()` function produces a graphical display summarizing a theme, as seen in Figure 7.4. A color version, summarizing the default color theme, is also shown in the color plates. Further details can usually be inferred from the setting names and the online documentation, and are not discussed here.

### 7.2.1 Nonstandard settings

Some settings do not fall into the pattern described above and deserve a separate discussion.

`clip`
: This parameter controls clipping separately for panels and strips. The default is

```
clip = list(panel = "on", strip = "on")
```

that is, graphical output produced by the panel and strip functions will be clipped to the extent of the panel and strip regions.

`fontsize`
: This parameter controls the baseline font size (before `cex` is applied) for all text and points in the plot.

`grid.pars`
: This parameter is initially unset, but can be used to specify global defaults for parameters of the underlying grid engine that cannot be otherwise specified. Examples include `lex` and `lineend`. A full list can be found on the `?gpar` help page in the grid package.

`layout.heights`, `layout.widths`
: These parameters control the amount of vertical and horizontal space allocated for the rows and columns that make up the layout of a lattice display. At the time of writing, every page of a lattice plot has rows allocated for (from top to bottom)

1. A `main` label
2. A legend (`key`)
3. A common axis at the top (for `relation = "same"`)

4. One or more `strip`s (one for each row of panels)
5. One or more `panel`s
6. Axes at the bottom of each panel (e.g., for `relation = "free"`)
7. Spaces for the `between` argument
8. A common axis at the bottom (for `relation = "same"`)
9. An `xlab` below the panel(s)
10. A `key` at the bottom
11. A `sub`-title

Of course, not all these components are used in every plot. The layout also includes rows that are just for spaces (padding) between components. All these rows have a default height, and the `layout.heights` parameter can be used to specify multipliers for the default. The exact names and their current settings can be obtained by

```
> str(trellis.par.get("layout.heights"))
List of 18
 $ top.padding      : num 1
 $ main             : num 1
 $ main.key.padding : num 1
 $ key.top          : num 1
 $ key.axis.padding : num 1
 $ axis.top         : num 1
 $ strip            : num 1
 $ panel            : num 1
 $ axis.panel       : num 1
 $ between          : num 1
 $ axis.bottom      : num 1
 $ axis.xlab.padding: num 1
 $ xlab             : num 1
 $ xlab.key.padding : num 1
 $ key.bottom       : num 1
 $ key.sub.padding  : num 1
 $ sub              : num 1
 $ bottom.padding   : num 1
```

For example, all the components with a name ending in "`padding`" can be set to 0 to make the layout as little wasteful of screen real estate as possible, while still allocating the minimum amount required for labels, legends, and the like. Some components, such as `panel`, `strip`, and `between`, are replicated for a display with multiple rows, and these components can be specified as a vector to achieve interesting results. The `layout.widths` parameter similarly controls the widths of columns in the layout; we leave the details for the reader to figure out.

`axis.components`

This parameter can be used to control the amount of space allocated for axis tick marks. It is rarely useful in practice.

## 7.3 Non-graphical options

A second set of settings is also maintained by lattice; these can be queried and modified using the functions `lattice.getOption()` and `lattice.options()`, which are analogous in behavior to `getOption()` and `options()`. These settings are global (not device-specific) and typically not graphical in nature, and primarily intended as a developer tool that allows experimentation with minimal code change. Because of its limited usefulness to the casual user, we do not discuss the available options extensively; the interested reader can find out more by inspecting the result of `lattice.options()` and reading the corresponding help page.

### 7.3.1 Argument defaults

One use of `lattice.options()` that is worth mentioning is in determining global defaults. The default layout in a lattice display counts rows from the bottom up, as in a graph, and not from the top down, as in a table. This is contrary to what many users expect. This behavior can be easily altered by specifying `as.table = TRUE` in a high-level call, but one might prefer to set a global preference instead by changing the default. This can be achieved with

```
> lattice.options(default.args = list(as.table = TRUE))
```

Default values can be set in this manner for several high-level arguments, including `aspect`, `between`, `page`, and `strip`. Some arguments in other functions also derive their defaults from settable options. The most useful of these is the `theme` argument of `trellis.device()`, which obtains its default from `lattice.getOption("default.theme")`.

## 7.4 Making customizations persistent

Customized themes can be made to persist across sessions using the R startup mechanism (see `?Startup`). There are two ways to do this, depending on whether lattice is automatically loaded during startup. In either case, the idea is to specify a default for the `theme` argument of `trellis.device()` through the options mechanism. Other options can be set at the same time. If lattice is to be loaded on startup, the following code might be included in `.First()` to change the default of `as.table` to `TRUE` and to make the standard color theme the default for all devices.

```
lattice.options(default.args = list(as.table = TRUE))
lattice.options(lattice.theme = standard.theme("pdf"))
```

A more sophisticated approach is to set a hook function that will be called only when lattice is attached through a call to `library()` or `require()`.

```
setHook(packageEvent("lattice", "attach"),
        function(...) {
            lattice.options(default.args = list(as.table = TRUE))
            lattice.options(default.theme =
                function() {
                    switch(EXPR = .Device,
                           postscript = ,
                           pdf = standard.theme(color = FALSE),
                           standard.theme("pdf", color = TRUE))
                })
        })
```

In this case the default theme is a function rather than a list, which uses the standard black and white theme as default for the `pdf()` and `postscript()` devices, and the standard color theme for all others.