SECOND EDITION

BONUS CHAPTER

R IN ACTION

Data analysis and graphics with R

Robert I. Kabacoff

MANNING

R in Action

Second Edition

Data analysis and graphics with R

by Robert I. Kabacoff

Bonus chapter 23

Copyright 2015 Manning Publications

brief contents

23 Bonus chapter Advanced graphics with the lattice package

This chapter covers

- An introduction to the lattice package
- Grouping and conditioning
- Adding information with panel functions
- Customizing a lattice graph's appearance

In this book, you created a wide variety of graphs using base functions from the `graphics` package included with R and specialized functions from author-contributed packages. In chapter 19, you learned a new syntax for creating graphs using functions from the `ggplot2` package. The `ggplot2` package offers an alternative to R's base graphics and is particularly useful when creating complex plots.

In this bonus chapter, we'll look at the `lattice` package, written by Deepayan Sarkar (2008); this package implements trellis graphics as outlined by Cleveland (1985, 1993). The `lattice` package has grown beyond Cleveland's original approach to visualizing data and now provides a comprehensive system for creating statistical graphics. Like `ggplot2`, `lattice` graphics has its own syntax, offers an alternative to the base graphics, and excels at plotting complex data. Analysts tend

to use either `lattice` or `ggplot2`, based on personal preference. Try them both and see which one you prefer.

23.1 The lattice package

The `lattice` package provides a comprehensive graphical system for visualizing univariate and multivariate data. In particular, many users turn to the `lattice` package because of its ability to easily generate trellis graphs.

A trellis graph displays the distribution of a variable, or the relationship between variables, separately for each level of one or more other variables. Consider the following question: *How do the heights of singers in the New York Choral Society vary by their vocal parts?*

Data on the heights and voice parts of choral members are provided in the singer dataset contained in the `lattice` package. In the following code

```
library(lattice)
histogram(~height | voice.part, data = singer,
    main="Distribution of Heights by Voice Pitch",
    xlab="Height (inches)")
```

`height` is the dependent variable, `voice.part` is called the *conditioning variable*, and a histogram is created for each of the eight voice parts. The graph is shown in figure 23.1. It appears that tenors and basses tend to be taller than altos and sopranos.

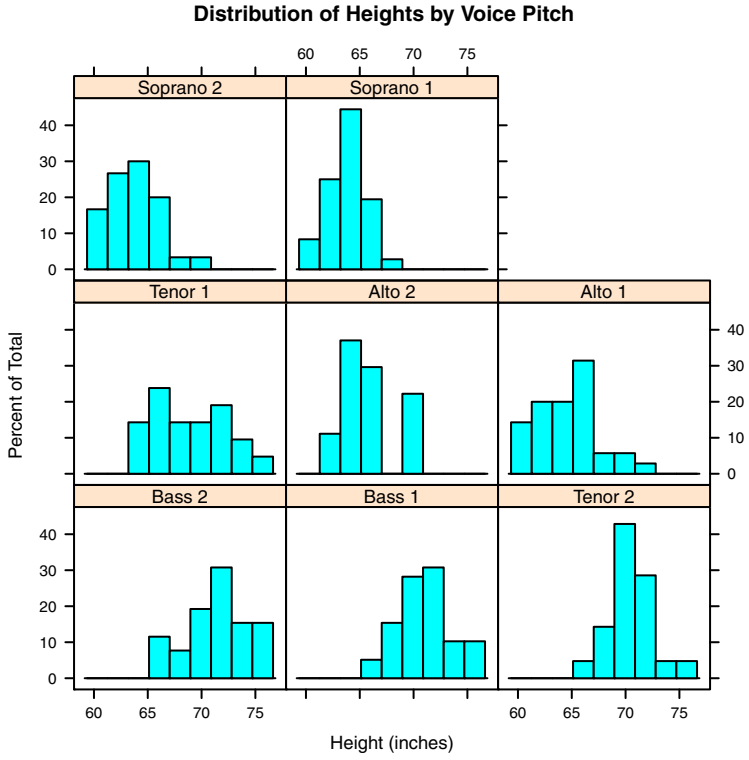

Figure 23.1 Trellis graph of singer heights by voice part

In trellis graphs, a separate *panel* is created for each level of the conditioning variable. If more than one conditioning variable is specified, a panel is created for each combination of factor levels. The panels are arranged into an array to facilitate comparisons. A label is provided for each panel in an area called the *strip*. As you'll see, the user has control over the graph displayed in each panel, the format and placement of the strip, the arrangement of the panels, the placement and content of legends, and many other graphic features.

The `lattice` package provides a wide variety of functions for producing univariate (dot plots, kernel density plots, histograms, bar charts, box plots), bivariate (scatter plots, strip plots, parallel box plots), and multivariate (3D plots, scatter plot matrices) graphs.

Each high-level graphing function follows the format

```
graph_function(formula, data=, options)
```

where

- *`graph_function`* is one of the functions listed in the second column of table 23.1.
- *`formula`* specifies the variable(s) to display and any conditioning variables.
- `data=` specifies a data frame.
- *`options`* are comma-separated parameters used to modify the content, arrangement, and annotation of the graph. See table 23.2 for a description of common options.

Let lowercase letters represent numeric variables and uppercase letters represent categorical variables (factors). The formula in a high-level graphing function typically takes the form

```
y ~ x | A * B
```

where variables on the left side of the vertical bar are called the *primary* variables and variables on the right are the *conditioning* variables. Primary variables map variables to the axes in each panel. Here, *`y~x`* describes the variables to place on the vertical and horizontal axes, respectively. For single-variable plots, replace *`y~x`* with *`~x`*. For 3D plots, replace *`y~x`* with *`z~x*y`*. Finally, for multivariate plots (scatter-plot matrix or parallel-coordinates plot), replace *`y~x`* with a data frame. Note that conditioning variables are always optional.

Following this logic, *`~x|A`* displays numeric variable *`x`* for each level of factor `A`. *`y~x|A*B`* displays the relationship between numeric variables *`y`* and *`x`* separately for every combination of factor `A` and `B` levels. *`A~x`* displays categorical variable `A` on the vertical axis and numeric variable *`x`* on the horizontal axis. *`~x`* displays numeric variable *`x`* alone. Other examples are shown in table 23.1.

To gain a quick overview of lattice graphs, try running the code in listing 23.1. The graphs are based on the automotive data (mileage, weight, number of gears, number of cylinders, and so on) included in the `mtcars` data frame. You may want to vary the formulas and view the results. (The resulting output has been omitted to save space.)

Table 23.1 Graph types and corresponding functions in the lattice package

Graph type	Function	Formula examples
3D contour plot	`contourplot()`	*z~x*y*
3D level plot	`levelplot()`	*z~y*x*
3D scatter plot	`cloud()`	*z~x*y|A*
3D wireframe graph	`wireframe()`	*z~y*x*
Bar chart	`barchart()`	*x~A* or *A~x*
Box plot	`bwplot()`	*x~A* or *A~x*
Dot plot	`dotplot()`	*~x|A*
Histogram	`histogram()`	*~x*
Kernel-density plot	`densityplot()`	*~x|A*B*
Parallel-coordinates plot	`parallelplot()`	`dataframe`
Scatter plot	`xyplot()`	*y~x|A*
Scatter-plot matrix	`splom()`	`dataframe`
Strip plots	`stripplot()`	*A~x* or *x~A*

Note: In these formulas, lowercase letters represent numeric variables and uppercase letters represent categorical variables.

Listing 23.1 Lattice plot examples

```
library(lattice)
attach(mtcars)

gear <- factor(gear, levels=c(3, 4, 5),
               labels=c("3 gears", "4 gears", "5 gears"))
cyl <- factor(cyl, levels=c(4, 6, 8),
              labels=c("4 cylinders", "6 cylinders", "8 cylinders"))

densityplot(~mpg,
            main="Density Plot",
           xlab="Miles per Gallon")

densityplot(~mpg | cyl,
            main="Density Plot by Number of Cylinders",
            xlab="Miles per Gallon")

bwplot(cyl ~ mpg | gear,
       main="Box Plots by Cylinders and Gears",
       xlab="Miles per Gallon", ylab="Cylinders")

xyplot(mpg ~ wt | cyl * gear,
       main="Scatter Plots by Cylinders and Gears",
       xlab="Car Weight", ylab="Miles per Gallon")

cloud(mpg ~ wt * qsec | cyl,
      main="3D Scatter Plots by Cylinders")
```

```
dotplot(cyl ~ mpg | gear,
        main="Dot Plots by Number of Gears and Cylinders",
        xlab="Miles Per Gallon")

splom(mtcars[c(1, 3, 4, 5, 6)],
      main="Scatter Plot Matrix for mtcars Data")

detach(mtcars)
```

High-level plotting functions in the `lattice` package produce graphic objects that can be saved and manipulated. For example,

```
library(lattice)
mygraph <- densityplot(~height|voice.part, data=singer)
```

creates a trellis density plot and saves it as object `mygraph`. But no graph is displayed. Issuing the statement `plot(mygraph)` (or simply `mygraph`) will display the graph.

It's easy to modify lattice graphs through the use of options. Common options are given in table 23.2. You'll see examples of many of these later in the chapter.

Table 23.2 Common options for lattice high-level graphing functions

Options	Description
`aspect`	A number specifying the aspect ratio (height/width) for the graph in each panel.
`col`, `pch`, `lty`, `lwd`	Vectors specifying the colors, symbols, line types, and line widths to be used in plotting, respectively.
`group`	Grouping variable (factor).
`index.cond`	List specifying the display order of the panels.
`key` (or `auto.key`)	Function used to supply legend(s) for grouping variable(s).
`layout`	Two-element numeric vector specifying the arrangement of the panels (number of columns, number of rows). If desired, a third element can be added to indicate the number of pages.
`main`, `sub`	Character vectors specifying the main title and subtitle.
`panel`	Function used to generate the graph in each panel.
`scales`	List providing axis annotation information.
`strip`	Function used to customize panel strips.
`split`, `position`	Numeric vectors used to place more than one graph on a page.
`type`	Character vector specifying one or more plotting options for scatter plots (`p` = points, `l` = lines, `r` = regression line, `smooth` = loess fit, `g` = grid, and so on).
`xlab`, `ylab`	Character vectors specifying horizontal and vertical axis labels.
`xlim`, `ylim`	Two-element numeric vectors giving the minimum and maximum values for the horizontal and vertical axes, respectively.

You can issue these options in the high-level function calls or within the panel functions discussed in section 23.3.

You can also use the `update()` function to modify a lattice graphic object. Continuing the singer example, the following

```
newgraph <- update(mygraph, col="red", pch=16,
                   cex=.8, jitter=.05, lwd=2)
```

would modify `mygraph` using red curves and symbols (`color="red"`), filled dots (`pch=16`), smaller (`cex=.8`) and more highly jittered points (`jitter=.05`), and lines of double thickness (`lwd=2`). The resulting graph is saved as `newgraph`. Now that we've reviewed the general structure of a high-level lattice function, let's look at conditioning variables in more detail.

23.2 Conditioning variables

As you've seen, one of the most powerful features of lattice graphs is the ability to add conditioning variables. If one conditioning variable is present, a separate panel is created for each level. If two conditioning variables are present, a separate panel is created for each combination of levels for the two variables. It's rarely useful to include more than two conditioning variables.

Typically, conditioning variables are factors. But what if you want to condition on a continuous variable? One approach would be to transform the continuous variable into a discrete variable using R's `cut()` function. Alternatively, the `lattice` package provides functions for transforming a continuous variable into a data structure called a *shingle*. Specifically, the continuous variable is divided into a series of (possibly) overlapping ranges. For example, the function

```
myshingle <- equal.count(x, number=n, overlap=proportion)
```

takes continuous variable *x* and divides it into *n* intervals with *proportion* overlap and equal numbers of observations in each range, and returns it as the variable `myshingle` (of class `shingle`). Printing or plotting this object (for example, `plot(myshingle)`) displays the shingle's intervals.

Once a continuous variable has been converted to a shingle, you can use it as a conditioning variable. For example, let's use the `mtcars` dataset to explore the relationship between miles per gallon and car weight conditioned on engine displacement. Because engine displacement is a continuous variable, first let's convert it to a shingle variable with three levels:

```
displacement <- equal.count(mtcars$disp, number=3, overlap=0)
```

Next, use this variable in the `xyplot()` function:

```
xyplot(mpg~wt|displacement, data=mtcars,
   main = "Miles per Gallon vs. Weight by Engine Displacement",
   xlab = "Weight", ylab = "Miles per Gallon",
   layout=c(3, 1), aspect=1.5)
```

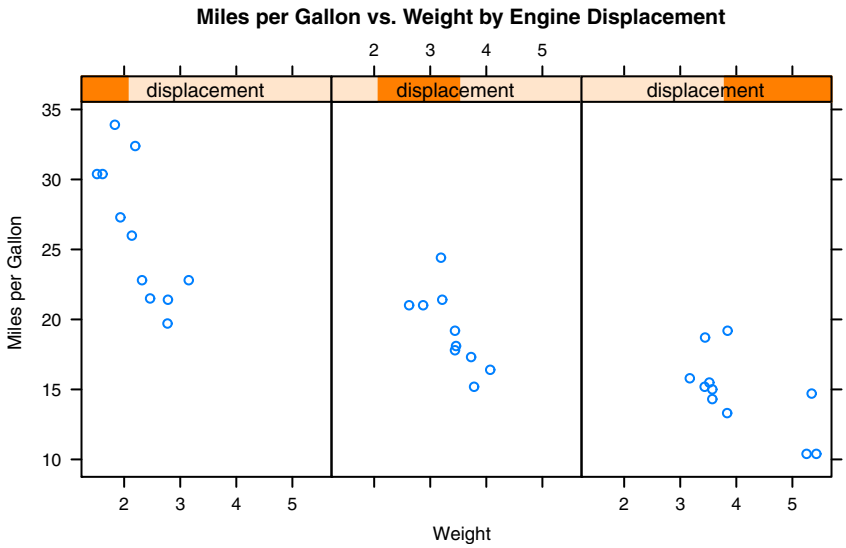

Figure 23.2 Trellis plot of miles per gallon vs. car weight conditioned on engine displacement. Because engine displacement is a continuous variable, it has been converted to three non-overlapping shingles with equal numbers of observations.

The results are shown in figure 23.2. Note that I also used options to modify the layout of the panels (three columns and one row) and the aspect ratio (height/width) in order to make comparisons among the three groups easier.

You can see that the labels in the panel strips of figure 23.1 and figure 23.2 differ. The representation in figure 23.2 indicates the continuous nature of the conditioning variable, with the darker color indicating the range of values for the conditioning variable in the given panel. In the next section, you'll use panel functions to customize the output further.

23.3 Panel functions

Each of the high-level plotting functions in table 23.1 employs a default function to draw the panels. These default functions follow the naming convention `panel.graph_function`, where *graph_function* is the high-level function. For example,

```
xyplot(mpg~wt|displacement, data=mtcars)
```

could also be written as

```
xyplot(mpg~wt|displacement, data=mtcars, panel=panel.xyplot)
```

This is a powerful feature because it allows you to replace the default panel function with a customized function of your own design. You can incorporate one or more of the 50+ default panel functions in the `lattice` package into your customized function as well. Customized panel functions give you a great deal of flexibility in designing output that meets your needs. Let's look at some examples.

In the previous section, you plotted gas mileage by automobile weight, conditioned on engine displacement. What if you want to include regression lines, rug plots, and grid lines? You can do this by creating your own panel function (see the following listing). The resulting graph is provided in figure 23.3.

Listing 23.2 `xyplot` with custom panel function

```
library(lattice)
displacement <- equal.count(mtcars$disp, number=3, overlap=0)

mypanel <- function(x, y) {
             panel.xyplot(x, y, pch=19)
             panel.rug(x, y)
             panel.grid(h=-1, v=-1)
             panel.lmline(x, y, col="red", lwd=1, lty=2)
           }

xyplot(mpg~wt|displacement, data=mtcars,
       layout=c(3, 1),
       aspect=1.5,
       main = "Miles per Gallon vs. Weight by Engine Displacement",
       xlab = "Weight",
       ylab = "Miles per Gallon",
       panel = mypanel)
```

1 **Customized panel function**

Here you wrap four separate building-block functions into your own `mypanel()` function and apply it within `xyplot()` through the `panel=` option 1. The `panel.xyplot()` function generates the scatter plot using a filled circle (`pch=19`). The `panel.rug()`

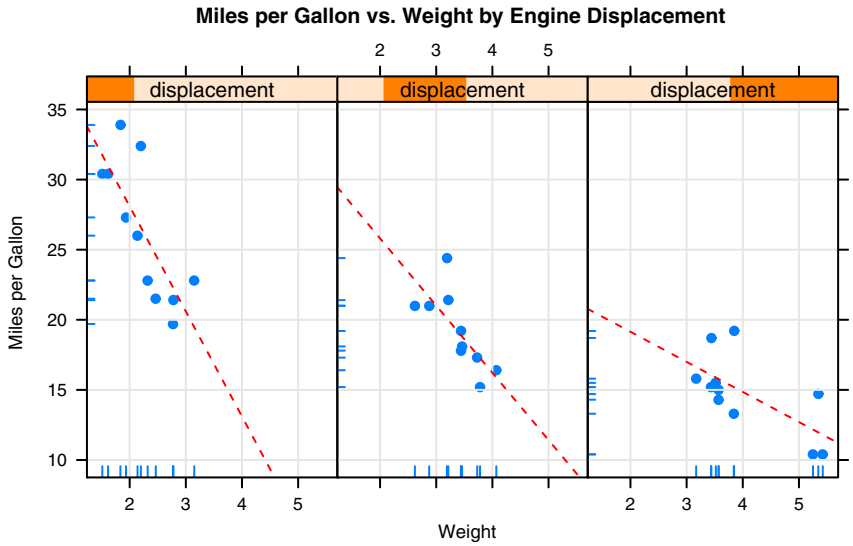

Figure 23.3 Trellis plot of miles per gallon vs. car weight conditioned on engine displacement. A custom panel function has been used to add regression lines, rug plots, and grid lines.

function adds rug plots to both the x- and y-axes of each panel. `panel.rug(x, FALSE)` or `panel.rug(FALSE, y)` would have added rugs to just the horizontal or vertical axis, respectively. The `panel.grid()` function adds horizontal and vertical grid lines (using negative numbers forces them to line up with the axis labels). Finally, the `panel.lmline()` function adds a regression line that's rendered as red (`col="red"`), dashed (`lty=2`) lines, of standard thickness (`lwd=1`). Each default panel function has its own structure and options. See the help page on each (for example, `help(panel.lmline)`) for further details.

As a second example, you'll graph the relationship between gas mileage and engine displacement (considered as a continuous variable), conditioned on type of automobile transmission. In addition to creating separate panels for automatic and manual transmission engines, you'll add smoothed fit lines and horizontal mean lines. The code is given in the following listing.

Listing 23.3 `xyplot` with a custom panel function and additional options

```
library(lattice)
mtcars$transmission <- factor(mtcars$am, levels=c(0,1),
                              labels=c("Automatic",  "Manual"))

panel.smoother <- function(x, y) {
                    panel.grid(h=-1, v=-1)
                    panel.xyplot(x, y)
                    panel.loess(x, y)
                    panel.abline(h=mean(y), lwd=2, lty=2, col="darkgreen")
                  }

xyplot(mpg~disp|transmission,data=mtcars,
       scales=list(cex=.8, col="red"),
       panel=panel.smoother,
       xlab="Displacement", ylab="Miles per Gallon",
       main="MPG vs Displacement by Transmission Type",
       sub = "Dotted lines are Group Means", aspect=1)
```

The graph produced by this code is provided in figure 23.4.

There are several things to note in this new code. The `panel.xyplot()` function plots the individual points, and the `panel.loess()` function plots nonparametric fit lines in each panel. The `panel.abline()` function adds horizontal reference lines at the mean `mpg` value for each level of the conditioning variable. (If you replaced `h=mean(y)` with `h=mean(mtcars$mpg)`, a single reference line would be drawn at the mean `mpg` value for the entire sample.) The `scales=` option renders scale annotations (the axis numbers and tick marks) in red and at 80% of the default font size.

In the previous example, you could use `scales=list(x=list(), y=list())` to specify separate options for the horizontal and vertical axes. See `help(xyplot)` for details on the many scale options available. In the next section, you'll learn how to superimpose data from groups of observations, rather than presenting them in separate panels.

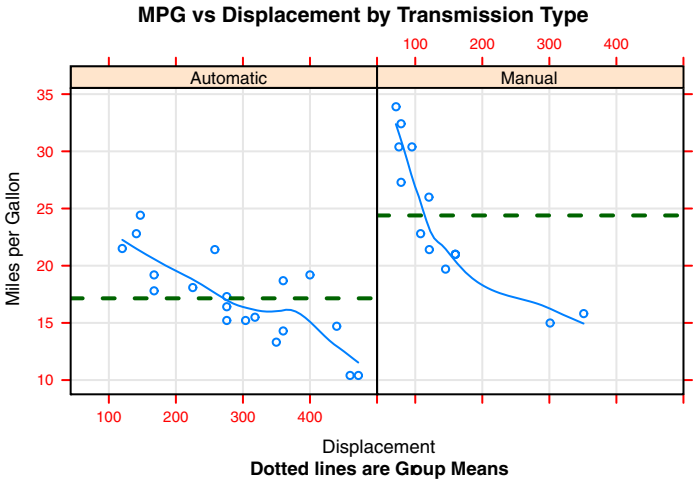

Figure 23.4 Trellis graph of miles per gallon vs. engine displacement conditioned on transmission type. Smoothed lines (loess), grids, and group mean levels have been added.

23.4 Grouping variables

When you include a conditioning variable in a lattice graph formula, a separate panel is produced for each level of that variable. If you want to superimpose the results for each level instead, you can specify the variable as a grouping variable.

Let's say that you want to display the distribution of gas mileage for cars with manual and automatic transmissions using kernel-density plots. You can superimpose these plots using this code:

```
library(lattice)
mtcars$transmission <- factor(mtcars$am, levels=c(0, 1),
                              labels=c("Automatic", "Manual"))
densityplot(~mpg, data=mtcars,
            group=transmission,
            main="MPG Distribution by Transmission Type",
            xlab="Miles per Gallon",
            auto.key=TRUE)
```

The resulting graph is presented in figure 23.5. By default, the `group=` option superimposes the plots from each level of the grouping variable. Points are plotted as open circles, lines are solid, and level information is distinguished by color. As you can see, the colors are difficult to differentiate when printed in grayscale. Later you'll learn how to change these defaults.

Note that legends and keys aren't produced by default. The option `auto.key=TRUE` creates a rudimentary legend and places it above the graph. You can make limited changes to this automated key by specifying options in a list. For example,

```
auto.key=list(space="right", columns=1, title="Transmission")
```

places the legend to the right of the graph, presents the key values in a single column, and adds a legend title.

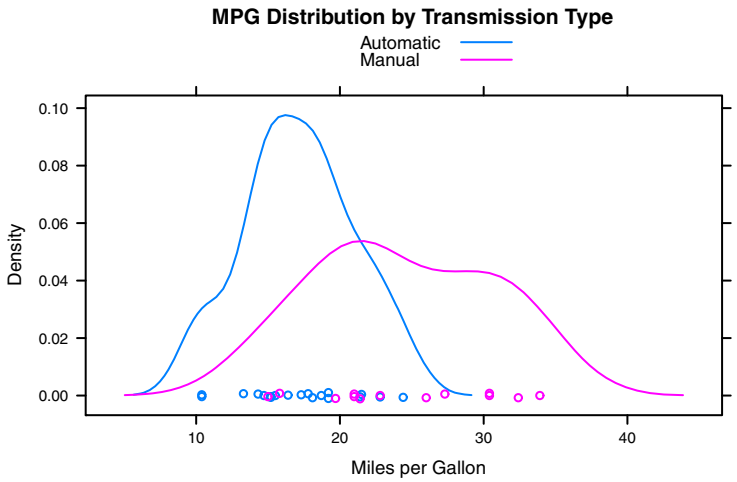

Figure 23.5 Kernel-density plots for miles per gallon grouped by transmission type. Jittered points are provided on the horizontal axis.

If you want to exert greater control over the legend, you can use the `key=` option. An example is given next. The resulting graph is shown in figure 23.6.

Listing 23.4 Kernel-density plot with a group variable and customized legend

```
library(lattice)
mtcars$transmission <- factor(mtcars$am, levels=c(0, 1),
                              labels=c("Automatic", "Manual"))

colors <- c("red", "blue")
lines  <- c(1,2)
points <- c(16,17)

key.trans <- list(title="Transmission",
                  space="bottom", columns=2,
                  text=list(levels(mtcars$transmission)),
                  points=list(pch=points, col=colors),
                  lines=list(col=colors, lty=lines),
                  cex.title=1, cex=.9)

densityplot(~mpg, data=mtcars,
            group=transmission,
            main="MPG Distribution by Transmission Type",
            xlab="Miles per Gallon",
            pch=points, lty=lines, col=colors,
            lwd=2, jitter=.005,
            key=key.trans)
```

1 **Color, line, and point specifications**

2 **Legend customization**

3 **Density plot**

Here, the plotting symbols, line types, and colors are specified as vectors 1. The first element of each vector is applied to the first level of the group variable, the second element to the second level, and so forth. A list object is created to hold the legend options 2. These options place the legend below the graph in two columns and

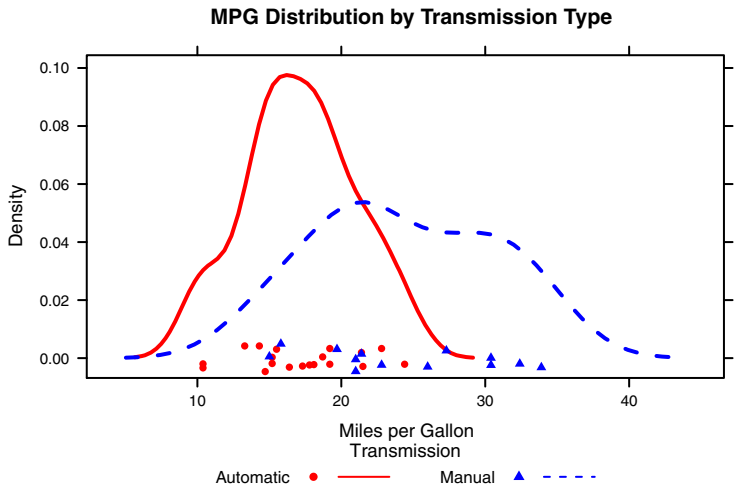

Figure 23.6 Kernel-density plots for miles per gallon grouped by transmission type. Graphical parameters have been modified, and a customized legend has been added. The custom legend specifies color, shape, line type, character size, and title.

include the level names, point symbols, line types, and colors. The legend title is rendered slightly larger than the text for the symbols.

The same plot symbols, line types, and colors are specified in the `densityplot()` function ❸. Additionally, the line width and jitter are increased to improve the appearance of the graph. Finally, the key is set to use the previously defined list. This approach to specifying a legend for the grouping variable gives you a great deal of flexibility. In fact, you can create more than one legend and place them in different areas of the graph (not shown here).

Before completing this section, let's consider an example that includes group and conditioning variables in a single plot. The `CO2` data frame, included with the base R installation, describes a study of cold tolerance of the grass species *Echinochloa crus-galli*.

The data describe carbon dioxide uptake rates (`uptake`) for 12 plants (`Plant`), at 7 ambient carbon dioxide concentrations (`conc`). Six plants were from Quebec and six plants were from Mississippi. Three plants from each location were studied under chilled conditions, and three plants were studied under non-chilled conditions. In this example, `Plant` is the group variable and both `Type` (Quebec/Mississippi) and `Treatment` (chilled/non-chilled) are conditioning variables. The following code produces the plot in figure 23.7.

Listing 23.5 `xyplot` with group and conditioning variables and customized legend

```
library(lattice)
colors <- "darkgreen"
symbols <- c(1:12)
linetype <- c(1:3)
```

```
key.species <- list(title="Plant",
                    space="right",
                    text=list(levels(CO2$Plant)),
                    points=list(pch=symbols, col=colors))

xyplot(uptake~conc|Type*Treatment, data=CO2,
       group=Plant,
       type="o",
       pch=symbols, col=colors, lty=linetype,
       main="Carbon Dioxide Uptake\nin Grass Plants",
       ylab=expression(paste("Uptake ",
              bgroup("(", italic(frac("umol","m"^2)), ")"))),
       xlab=expression(paste("Concentration ",
              bgroup("(", italic(frac(mL,L)), ")"))),
       sub = "Grass Species: Echinochloa crus-galli",
       key=key.species)
```

Note the use of `\n` to give you a two-line title and the use of the `expression()` function to add mathematical notation to the axis labels. Here, color is suppressed as a group differentiator by specifying a single color in the `col=` option. In this case, adding 12 different colors is overkill and distracts from the goal of easily visualizing the

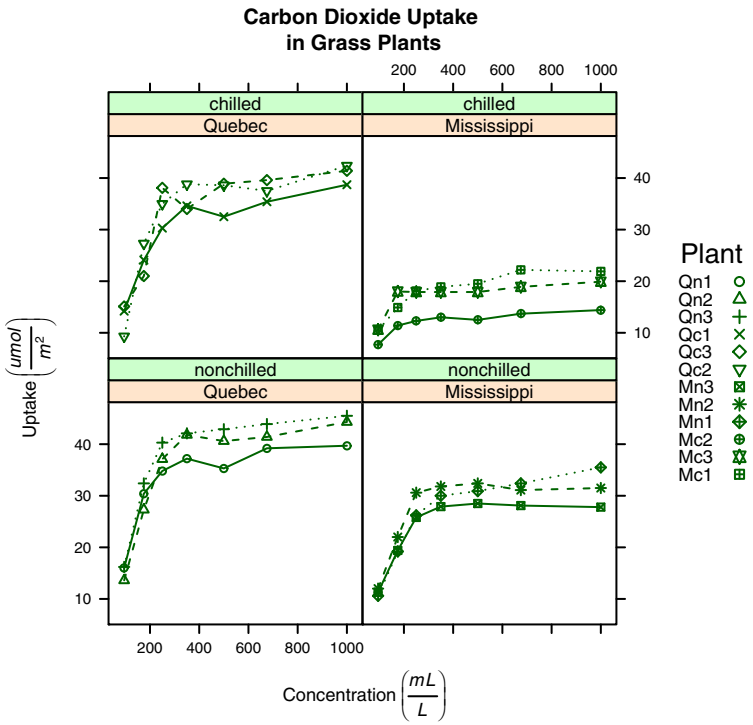

Figure 23.7 `xyplot` showing the impact of ambient carbon dioxide concentrations on carbon dioxide uptake for 12 plants in two treatment conditions and two types. `Plant` is the group variable, and `Treatment` and `Type` are the conditioning variables.

relationships in each panel. Clearly, there's something different about the Mississippi grasses in the chilled condition.

Up to this point, you've been modifying graphic elements in your charts through options passed to either the high-level graph functions (for example, `xyplot(pch=17)`) or the panel functions they use (for example, `panel.xyplot(pch=17)`). But such changes are in effect only for the duration of the function call. In the next section, you'll review a method for changing graphical parameters that persists for the duration of the interactive session or batch execution.

23.5 Graphic parameters

In chapter 3, you learned how to view and set default graphics parameters using the `par()` function. Although this works for graphs produced with R's native graphic system, lattice graphs are unaffected by these settings. Instead, the graphic defaults used by lattice functions are contained in a large list object that can be accessed with the `trellis.par.get()` function and modified through the `trellis.par.set()` function. You can use the `show.settings()` function to display the current graphic settings visually.

As an example, let's change the default symbol used for superimposed points (that is, points in a graph that includes a group variable). The default is an open circle. You'll give each group its own symbol instead.

First, view the current defaults

```
show.settings()
```

and save them into a list called `mysettings`:

```
mysettings <- trellis.par.get()
```

You can see the components of this list by using the `names()` function:

```
> names(mysettings)
 [1] "grid.pars"          "fontsize"           "background"
 [4] "panel.background"   "clip"               "add.line"
 [7] "add.text"           "plot.polygon"       "box.dot"
[10] "box.rectangle"      "box.umbrella"       "dot.line"
[13] "dot.symbol"         "plot.line"          "plot.symbol"
[16] "reference.line"     "strip.background"   "strip.shingle"
[19] "strip.border"       "superpose.line"     "superpose.symbol"
[22] "superpose.polygon"  "regions"            "shade.colors"
[25] "axis.line"          "axis.text"          "axis.components"
[28] "layout.heights"     "layout.widths"      "box.3d"
[31] "par.xlab.text"      "par.ylab.text"      "par.zlab.text"
[34] "par.main.text"      "par.sub.text"
```

The defaults that are specific to superimposed symbols are contained in the `superpose.symbol` component:

```
> mysettings$superpose.symbol

$alpha
```

```
[1] 1 1 1 1 1 1 1
$cex
[1] 0.8 0.8 0.8 0.8 0.8 0.8 0.8
$col
[1] "#0080ff"   "#ff00ff"   "darkgreen" "#ff0000"   "orange"
[6] "#00ff00"   "brown"
$fill
[1] "#CCFFFF" "#FFCCFF" "#CCFFCC" "#FFE5CC" "#CCE6FF" "#FFFFCC"
[7] "#FFCCCC"
$font
[1] 1 1 1 1 1 1 1
$pch
[1] 1 1 1 1 1 1 1
```

The symbol used for each level of a group variable is an open circle (`pch=1`). Seven levels are defined, after which the symbols recycle.

To change the default, issue the following statements:

```
mysettings$superpose.symbol$pch <- c(1:10)
trellis.par.set(mysettings)
```

You can see the effect of your changes by issuing the `show.settings()` function again. Lattice graphs now use symbol 1 (open circle) for the first level of a group variable, symbol 2 (open triangle) for the second, and so on. Additionally, symbols have been defined for 10 levels of a grouping variable, rather than 7. The changes will remain in effect until all graphic devices are closed. You can change any graphic setting in this manner.

23.6 Customizing plot strips

The default background for the panel strip is peach colored for the first conditioning variable, pale green for the second conditioning variable, and pale blue for the third. Happily, you can customize the color, font, and other aspects of these strips. You can use the method described in the previous section, or you can take greater control and write a function to customize any aspect of the strip.

Let's start with the strip function. Just as the high-level graphing functions in `lattice` allows you to specify a panel function for controlling the contents of each panel, a strip function can be specified to control the appearance of each strip.

Consider the graph shown earlier in figure 23.1. The graph displays the heights of New York Choral Society singers by voice part. The background color is peach (or is it salmon?). What if you want the strip to be light grey, the text of the strip to be black, and the font to be italicized and shrunk by 20%? You can accomplish this with the following code:

```
library(lattice)
histogram(~height | voice.part, data = singer,
    strip = strip.custom(bg="lightgrey",
            par.strip.text=list(col="black", cex=.8, font=3)),
    main="Distribution of Heights by Voice Pitch",
    xlab="Height (inches)")
```

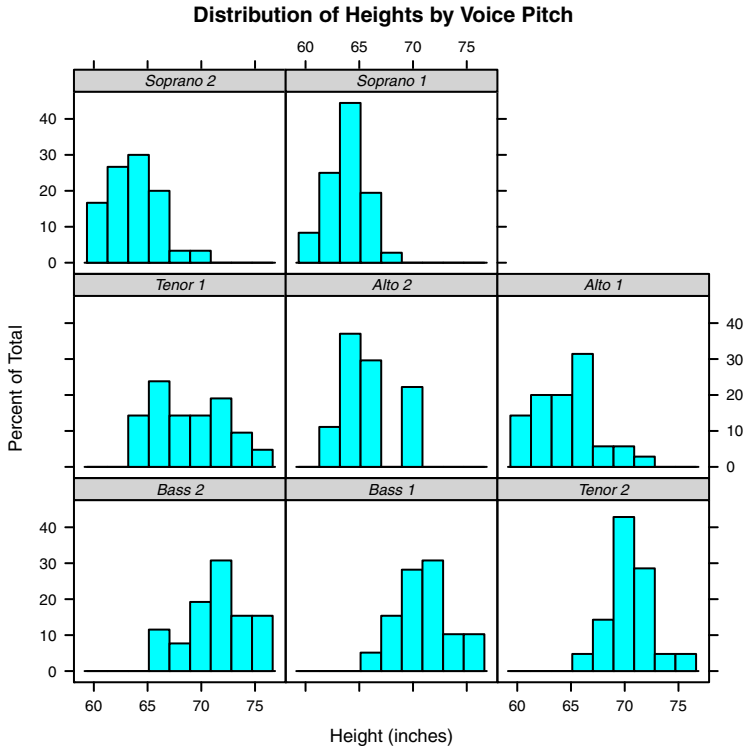

Figure 23.8 A trellis graph with a customized strip (light grey background, with a smaller, italicized font).

The resulting graph is presented in figure 23.8.

The `strip=` option specifies the function used to set the appearance of the strip. Although you can write a function from scratch (see `?strip.default`), it's often easier to change a few settings and leave the others at their default values. The `strip.custom()` function allows you to do this. The `bg` option controls the background color, and `par.strip.text` lets you control the appearance of the strip text.

The `par.strip.text` option uses a list to define text properties. The `col` and `cex` options control the text color and size. The `font` option can take the value 1, 2, 3, or 4, for normal, bold, italics, and bold italics typefaces, respectively.

The `strip=` option changes the appearance of the strips in the given graph. To change the appearance for all lattice graphs created in an R session, you can use the graphical parameters described in the previous section. The code

```
mysettings <- trellis.par.get()
mysettings$strip.background$col <- c("lightgrey", "lightgreen")
trellis.par.set(mysettings)
```

sets the strip background to `lightgrey` for the first conditioning variable and `lightgreen` for the second. The change will be in effect for the remainder of the session, or until the settings are changed again. Using graphical parameters is more convenient, but using a strip function gives you more options and greater control.

23.7 Page arrangement

In chapter 3, you learned how to place more than one graph on a page using the `par()` function. Because lattice functions don't recognize `par()` settings, you'll need a different approach for combining multiple lattice plots into a single graph. The easiest method involves saving your lattice graphs as objects and using the `plot()` function with either the `split=` or `position=` option specified.

The `split` option divides a page into a specified number of rows and columns and places graphs into designated cells of the resulting matrix. The format for the split option is

```
split=c(x, y, nx, ny)
```

which says to position the current plot at the *x, y* position in a regular array of *nx* by *ny* plots, where the origin is at the top left. For example, the following code

```
library(lattice)
graph1 <- histogram(~height | voice.part, data = singer,
          main = "Heights of Choral Singers by Voice Part" )
graph2 <- bwplot(height~voice.part, data = singer)
plot(graph1, split = c(1, 1, 1, 2))
plot(graph2, split = c(1, 2, 1, 2), newpage = FALSE)
```

places the first graph directly above the second graph. Specifically, the first `plot()` statement divides the page into one column (`nx` = 1) and two rows (`ny` = 2) and places the graph in the first column and first row (counting top-down and left-right). The

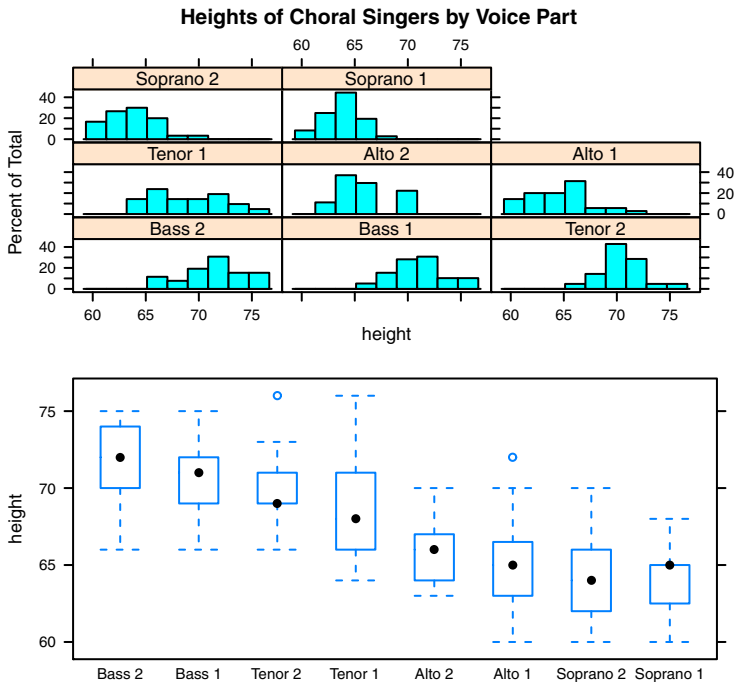

Figure 23.9 Using the `split` option to combine graphs

second `plot()` statement divides the page the same way but places the graph in the first column and second row. Because `plot()` starts a new page by default, you suppress this action by including the `newpage=FALSE` option. The plot is given in figure 23.9.

You can gain more control of sizing and placement by using the `position=` option. Consider the following code:

```
library(lattice)
graph1 <- histogram(~height | voice.part, data = singer,
          main = "Heights of Choral Singers by Voice Part")
graph2 <- bwplot(height~voice.part, data = singer)
plot(graph1, position=c(0, .3, 1, 1))
plot(graph2, position=c(0, 0, 1, .3), newpage=FALSE)
```

Here, `position=c(xmin, ymin, xmax, ymax)`, where the x-y coordinate system for the page is a rectangle with dimensions ranging from 0 to 1 on both the x- and y-axes and the origin (0,0) at bottom left. The graph is displayed in figure 23.10. To learn more about positioning graphs, see `help(plot.trellis)`.

You can also change the order of the panels in a lattice graph. The `index.cond` option in a high-level lattice graph function specifies the order of the conditioning variable levels. For the `voice.part` factor, the levels are

```
> levels(singer$voice.part)
[1] "Bass 2"    "Bass 1"    "Tenor 2"   "Tenor 1"   "Alto 2"
[6] "Alto 1"    "Soprano 2" "Soprano 1"
```

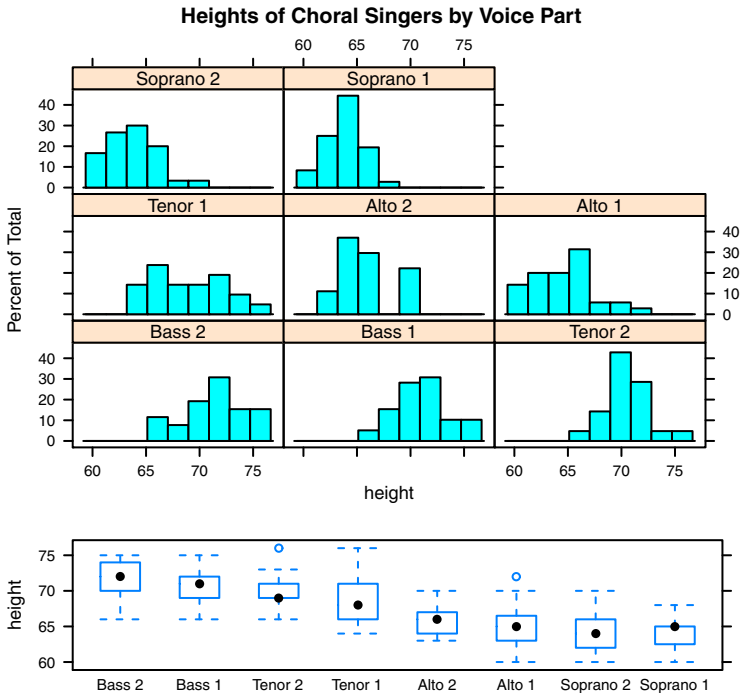

Figure 23.10 Using the `position` option to combine graphs with greater precision

Using this information,

```
histogram(~height | voice.part, data = singer,
          index.cond=list(c(2, 4, 6, 8, 1, 3, 5, 7)))
```

would place the 1 voice parts together (Bass 1, Tenor 1, …), followed by the 2 voice parts (Bass 2, Tenor 2, …). When there are two conditioning variables, include two vectors in the list. In listing 23.5, adding `index.cond=list(c(1, 2), c(2, 1))` would reverse the order of treatments in figure 23.7. The `index.cond` option is documented in `help(xyplot)`.

23.8 Going further

Lattice graphics offer a powerful and highly customizable approach to creating graphs in R. A number of useful resources can help you learn more about them. Deepayan Sarkar's "Lattice Graphics: An Introduction" (2008, http://mng.bz/jXUG) and William G. Jacoby's "An Introduction to Lattice Graphics in R" (2010, http://mng.bz/v4TO) offer excellent overviews. Sarkar's (2008) *Lattice: Multivariate Data Visualization with R* is the definitive book on the subject.

DATA/STATISTICS/PROGRAMMING

R IN ACTION Second Edition

Robert I. Kabacoff

Free eBook
SEE INSERT

Business pros and researchers thrive on data, and R speaks the language of data analysis. R is a powerful programming language for statistical computing. Unlike general-purpose tools, R provides thousands of modules for solving just about any data-crunching or presentation challenge you're likely to face. R runs on all important platforms and is used by thousands of major corporations and institutions worldwide.

R in Action, Second Edition teaches you how to use the R language by presenting examples relevant to scientific, technical, and business developers. Focusing on practical solutions, the book offers a crash course in statistics, including elegant methods for dealing with messy and incomplete data. You'll also master R's extensive graphical capabilities for exploring and presenting data visually. And this expanded second edition includes new chapters on forecasting, data mining, and dynamic report writing.

What's Inside

- Complete R language tutorial
- Using R to manage, analyze, and visualize data
- Techniques for debugging programs and creating packages
- OOP in R
- Over 160 graphs

A background in mathematics and statistics is helpful but not required. No prior experience with R is assumed.

Dr. Rob Kabacoff is a seasoned researcher and teacher who specializes in data analysis. He also maintains the popular Quick-R website at statmethods.net.

"Essential to anyone doing data analysis with R, whether in industry or academia."
—Cristofer Weber, NeoGrid

"A go-to reference for general R and many statistics questions."
—George Gaines
KYOS Systems Inc.

"Accessible language, realistic examples, and clear code."
—Samuel D. McQuillin
University of Houston

"Offers a gentle learning curve to those starting out with R for the first time."
—Indrajit Sen Gupta
Mu Sigma Business Solutions

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/RinActionSecondEdition

ISBN 13: 978-1-617291-38-8
ISBN 10: 1-617291-38-2
55999
9 781617 291388

MANNING $59.99 / Can $68.99 [INCLUDING eBOOK]