MathSoft

S-PLUS Trellis™ Graphics User's Manual

Richard A. Becker: AT&T Research
William S. Cleveland: Bell Labs

Trellis Versions 2.0 & 2.1: July 1996

MathSoft, Inc.
Seattle, Washington

The correct bibliographic reference for this document is as follows:
Becker, R. A. and Cleveland, W. S., *S-PLUS Trellis Graphics User's Manual*, Seattle: MathSoft, Inc., Murray Hill: Bell Labs, 1996.

Contents

Acknowledgements

The visualization methods of Trellis Graphics and the design and implementation of the S-PLUS code resulted from several years of hard running by Rick Becker, Bill Cleveland, and Ming Shyu at AT&T Bell Laboratories (now AT&T Research and Bell Labs). At MathSoft, Stephen Kaluzny took the baton pass with great skill and integrated the system into S-PLUS. More recently, Linda Clark at Bell Labs has joined the team, using her substantial skills to develop new applications, test the code, and maintain connections with users.

Chapter 1

About Trellis Graphics

Making graphs is very basic to data analysis. Whether you use the leading edge of statistical methods, or whether you want to quickly see the main features of your data, graphs are a must. They are the single most powerful class of tools for analyzing data.

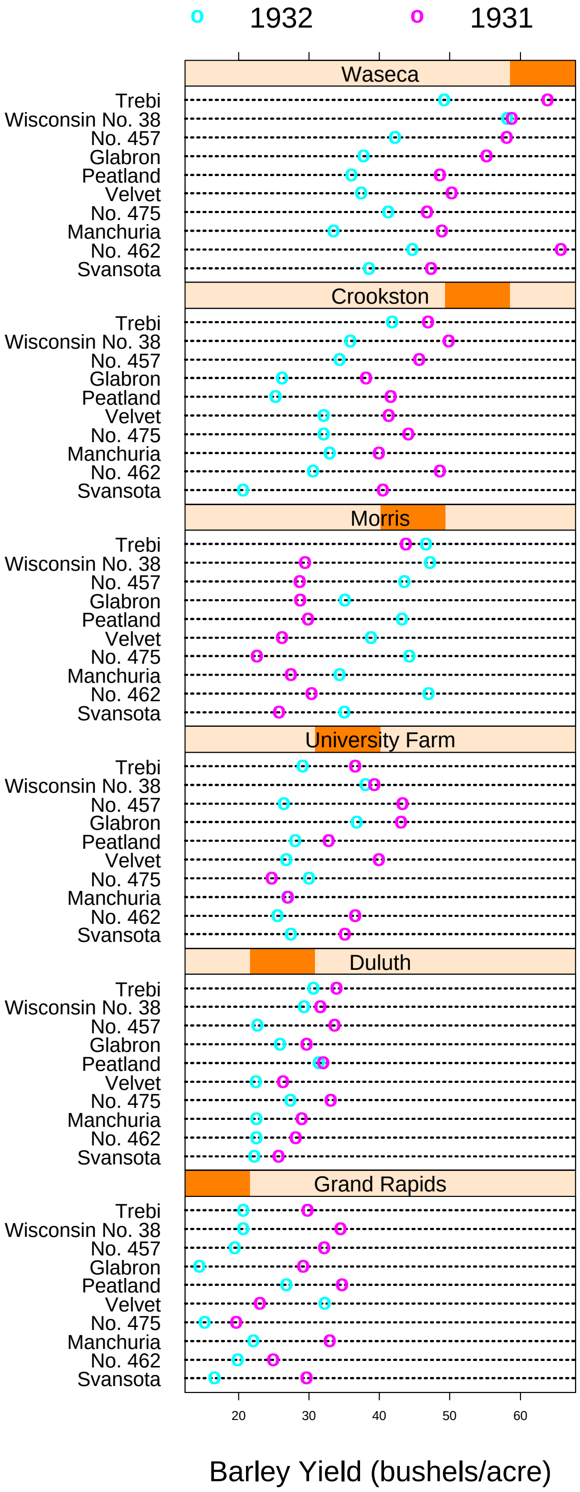
Trellis Graphics is a new system for making graphs, written using the core S-PLUS graphics functions. The Trellis software has many exciting features, some of them quite glitzy, but the true measure of a visualization system is how much it enables you to learn from your data. So in this chapter we will begin with two sets of data. Then we will discuss features.

1.1 Discovering the Morris Mistake

Figure 1.1 is a Trellis display of data from an agricultural field trial to study the crop barley. At six sites in Minnesota, ten varieties of barley were grown in each of two years. The data are the yields for all combinations of site, variety, and year, so there are $6 \times 10 \times 2 = 120$ observations. In figure 1.1, each panel displays the 20 yields at a single site.

The barley experiment was run in the 1930s. The data first appeared in a 1934 report published by the experimenters. Since then, the data have been analyzed and re-analyzed. R. A. Fisher presented the data for five of the sites in his classic book, *The Design of Experiments.* Publication in the book made the data famous, and many others subsequently analyzed them, usually to illustrate a new statistical method.

Then in the early 1990s, the data were visualized by Trellis Graphics. The result was a big surprise. Through 60 years and many analyses, an important happening in the data had gone undetected. Figure 1.1 shows the happening, which occurs at Morris. For all other sites, 1931 produced a significantly higher overall yield than 1932. The reverse is true at Morris. But most importantly, the amount by which 1932 exceeds 1931 at Morris is similar to the amounts by which 1931 exceeds 1932 at the other sites. Either an extraordinary natural event, such as disease or a local weather anomaly, produced a strange coincidence, or the years for Morris were inadvertently reversed. More Trellis displays, a statistical modeling of the data, and some background checks on the experiment led to the conclusion that the data are in error. But it was Trellis displays such as figure 1.1 that provided the "Aha!" which led to the conclusion.

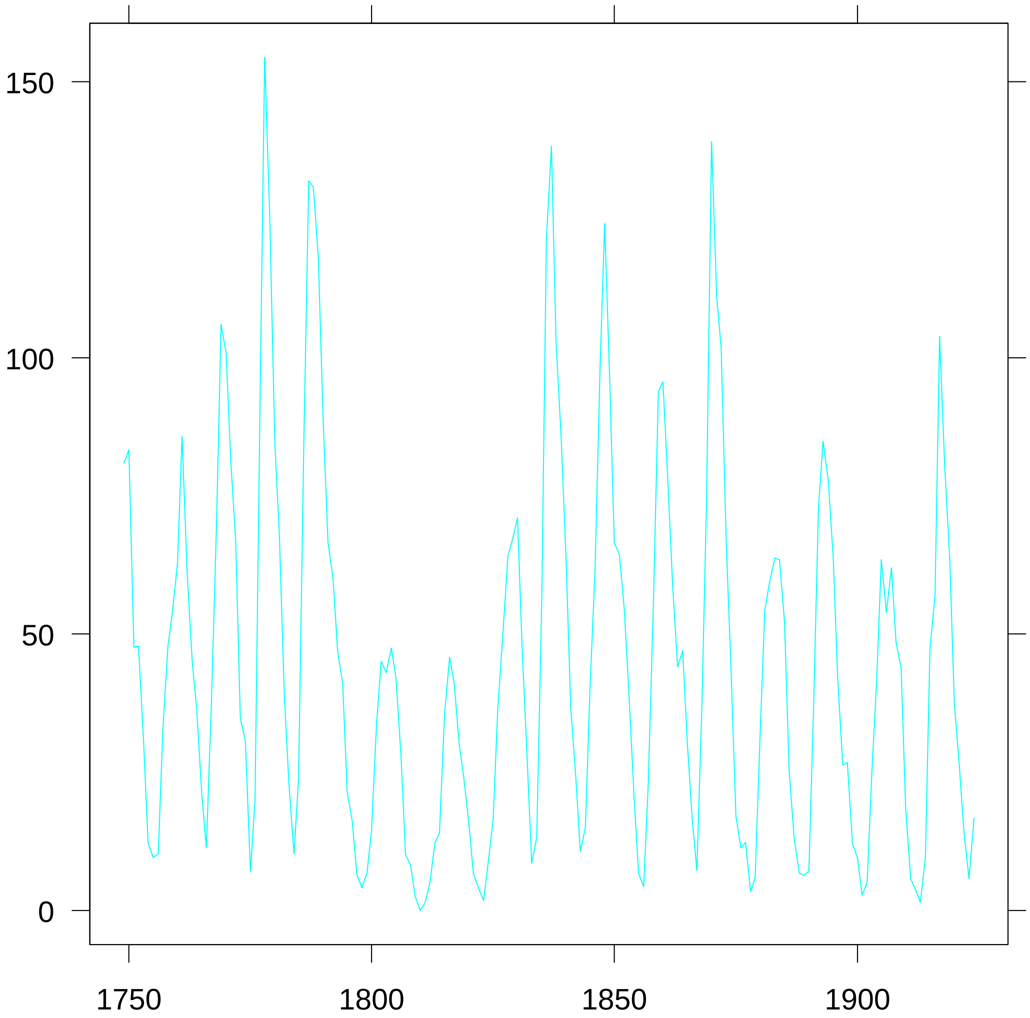


1932
1931
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
Barley Yield (bushels/acre)

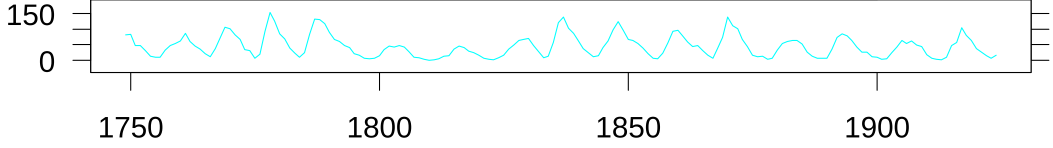
Figure 1.1

1.2 Seeing the Sunspot Cycles

The top panel of figure 1.2 graphs the yearly sunspot numbers from 1849 to 1924. The aspect ratio, the height of the data region of the graph divided by the width, is 1.0. An aspect ratio of 1.0 is what you might expect to see as a default in cases where aspect ratio has not been considered. But the graph fails to reveal an important property of the cycles. In the bottom panel, the data are graphed again, but this time the aspect ratio has been chosen by an algorithm in Trellis Graphics called *banking to 45°*. Now the property is revealed. The sunspot cycles typically rise more rapidly than they fall; this behavior is pronounced for the cycles with high peaks, is less pronounced for those with medium peaks, and disappears for those cycles with the lowest peaks. In the top panel, the aspect ratio of 1.0 prevents an accurate visual decoding of the slopes of the line segments connecting successive observations. In the bottom panel, banking allows a more accurate visual decoding of the slopes.



Sunspot Number vs. Year



Sunspot Number vs. Year

Figure 1.2

1.3 Trellis Features

Trellis Graphics is a large leap forward in helping you to understand the structure of your data, to understand the properties of models fitted to your data, and to understand how well such models describe the structure of your data. Here are a few of its many new features.

Multipanel Conditioning

Figure 1.1 illustrates *multipanel conditioning*: each panel of the figure shows the dependence of yield on variety, conditional on year and site. Multipanel conditioning is an exceptionally powerful visualization tool for studying the dependence of a response on two or more explanatory variables. It is particularly effective for ferreting out interactions. The panels are laid out into columns, rows, and pages. Figure 1.1 has only one page, but for large datasets, conditioning can result in a large number of panels, so more than one page is needed. This layout of panels is reminiscent of a garden trelliswork, and hence, the name "Trellis" Graphics.

Banking to 45°

Selecting the aspect ratio, or shape, of a graph to maximize the accuracy of our visual decoding of information was an outstanding problem of statistical graphics for decades. The solution, a breakthrough in data display, has been implemented in Trellis Graphics. Banking to 45° chooses the aspect ratio to center the absolute values of the slopes of selected line segments on 45°. Perceptual experiments have shown that this maximizes the accuracy of our visual decoding of the relative values of the slopes.

Automation

Trellis Graphics employs automation methods that save you time by automatically selecting rendering aspects—for example, multipanel layout, line types, plotting symbols, colors, and character sizes—to achieve effective visual perception of the structure of data. These automation methods are tuned to the graphics device you are using.

Tailoring Trellis Displays to Your Data

Still, even though our automation methods work well, you will want to alter displays.

You can alter what goes in the data region of your graph by altering a *panel function*, a simple procedure that describes what the panel display method should be. And you can alter panel functions to produce completely new types of displays tailored to the needs of your data.

You have very delicate control over labels and scales if you need it. Yet this control is direct and easy to exert.

1.4 Trellis and the Core S-PLUS Graphics

The core S-PLUS graphics is a collection of low-level drawing functions and graphics parameter settings. The low-level functions draw graphical elements. For example, `points()`, draws plotting symbols and `lines()` draws lines. The parameter settings govern the details of how graphical elements are rendered. For example, `pch = "+"` sets the plotting symbol to a plus sign.

Trellis Graphics employs the core graphics in two ways. First, Trellis has been implemented using the core graphics. Second, when you write a panel function to tailor the display to your data, you use features of the core graphics; typically, these are very simple features, considerably simpler than the Trellis implementation, which used just about every feature of the core.

1.5 Trellis vs. the Old S-PLUS High-Level Graphics

Since the very beginning of S-PLUS there has been a collection of high-level graphics functions that are used to display graphs. Examples are `plot()`, `qqnorm()`, and `persp()`. These routines, like Trellis Graphics, are also implemented using the core graphics.

Trellis Graphics provides more functionality than the old high-level capabilities; there are many new ways to display data, such as multipanel conditioning. It has also greatly improved some of the old display methods. For example, `wireframe()` does a better job of 3-D rendering than `persp()`. Trellis Graphics also has a better mechanism for the details of rendering graphs—aspect ratio, plotting symbols, colors, line types, panel layouts, coordinated scales on different graphs, and so forth. The defaults work better and users can now make changes with much more effective and predictable results.

Chapter 2

About This User's Manual

2.1 What Does this *Manual* Cover?

This *User's Manual* provides an introduction to Trellis Graphics Versions 2.0 and 2.1. The *Manual* is not a guide for earlier versions of Trellis Graphics.

The *Manual* was meant to be read from the beginning to teach you how to use Trellis. It was also meant to serve as a reference while you are using Trellis; the Table of Contents should be helpful for locating sections that describe the feature in which you are interested.

2.2 What You Need to Know About the Rest of S-PLUS to Use Trellis Graphics?

You can employ Trellis Graphics to do quite useful things with just a minimum of knowledge of the rest of S-PLUS. All you need to get started is a knowledge of the material in chapters 1-5, 7, and 10-11 of *A Gentle Introduction to S-PLUS*, one of the manuals available from MathSoft.

2.3 Other Reading About Trellis

This *Manual* is available in both color and black-and-white versions on the World Wide Web at

http://cm.bell-labs.com/stat/project/trellis/

This *User's Manual* does not show you examples of the use of Trellis to analyze data, complete with problem description, data description, analysis, and conclusions. Data analysis is discussed in *The Visual Design and Control of Trellis Display* by Rick Becker, Bill Cleveland, and Ming Shyu. This document is also available at the above Web site.

Even more examples can be found in the book *Visualizing Data* by Bill Cleveland. It was written at a time when Trellis Graphics was in its infancy, so not all of today's capabilities are discussed, but the examples it does have are presented in great detail. The book, along with the related book, *The Elements of Graphing Data*, is available from the publisher, Hobart Press, at books@hobart.com.

2.4 Some Important Conventions Used in this *Manual*

S-PLUS commands and expressions, names of S-PLUS objects, and the arguments of S-PLUS functions appear in bold. For example, in this *Manual* we will make use of a data frame `gas`, which has two numeric variables, `NOx` and `E`. Later we will use the function `xyplot()` and its argument `formula=` to make a scatterplot of `NOx` against `E`:

```
xyplot(formula = gas$NOx ~ gas$E)
```

In the previous paragraph we used two other conventions. The name of the function that makes the scatterplot is actually `xyplot`, but we write it as `xyplot()` to signal that this S-PLUS object is a function. Similarly, the argument is `formula`, but we write it as `formula=` to signal that we are making reference to an argument.

2.5 Data Sets

In this *Manual*, a number of datasets are used as examples. In addition, Trellis Graphics contains functions that draw graphs to show how the system works; these functions use datasets. The datasets in these examples are contained in either the Trellis library or other S-PLUS databases, or are computed by commands given in the *Manual*.

2.6 Code for the Examples in This *Manual*

There are many examples in this *Manual*. The code for drawing a display is given in the same section as the discussion of the display.

Chapter 3

Getting Started

3.1 trellis.device()

You need to have a graphics device on which to draw. If you have not specified a device, but you execute a function that draws a graph, then a color screen device is automatically set up for you.

The two devices that come up automatically can also be specified directly with `trellis.device()`. On Windows the command is

```
trellis.device(win.graph)
```

On UNIX the command is

```
trellis.device(motif)
```

For some UNIX systems, there is another screen device, `openlook`.

You can send Trellis graphs to a printer. Also, you can set up multiple devices; for example, you might have two devices that are graphics windows on your screen and one device that is a printer. Information is given about this in chapter 11.

WARNING: If you have used the old S-PLUS graphics, then you will know that you set up devices in a different way. For example, on Windows, you set up the screen device by

```
win.graph()
```

If you do this by mistake, you will find the Trellis graphs are not rendered nearly as well because the graphical parameters of the core S-PLUS graphics will not be customized to the device as they are when you use `trellis.device()`.

3.2 dev.off()

You turn off a graphics device by the function

```
dev.off()
```

or just be quitting from S-PLUS.

3.3 Trellis Objects: print.trellis() and update()

Trellis display functions return objects of class `trellis`. The expression

```
xyplot(formula = gas$NOx ~ gas$E)
```

draws a graph on the graphics device. The expression

```
foo <- xyplot(formula = gas$NOx ~ gas$E)
```

saves the graph in `foo` but does not draw it. If you then type

```
foo
```

the graph is drawn.

It is the print method for trellis objects that sends a graph to a device. For the example just given, typing `foo` causes S-PLUS to use `print(foo)` to display the graph. The reason for mentioning this is that you must sometimes explicitly use `print(foo)`—when the graph is made from a function or from a source file.

Having graphs stored as objects can make changing a display much simpler, especially when the display goes through a series of small changes, a frequent occurrence since data display is relentlessly iterative. The function `update()` changes Trellis objects. For example,

```
foo <- update(foo, main = "Dependence of NOx on E")
```

adds a title to the graph stored earlier in `foo` and stores the result back in `foo`.

3.4 Example Functions

The example functions in the Trellis library draw displays to show you the Trellis capabilities and a bit about how Trellis works. The word `example.` begins the names of all of the example functions. Figure 3.1 shows the result of executing one of these functions:

```
example.normal.qq()
```

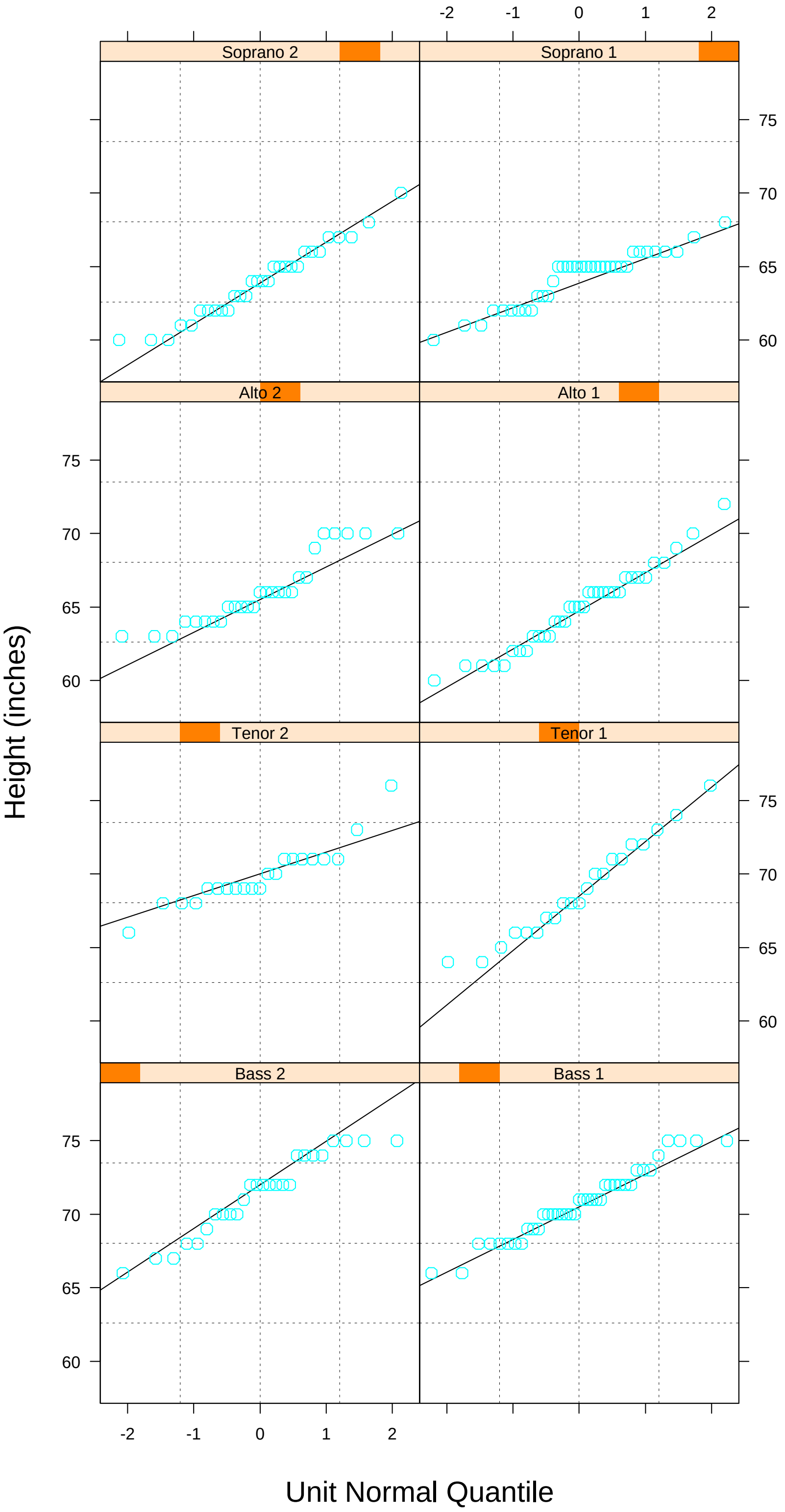

Soprano 2
Soprano 1
Alto 2
Alto 1
Tenor 2
Tenor 1
Bass 2
Bass 1
Height (inches)
Unit Normal Quantile
-2
-1
0
1
2
60
65
70
75

Figure 3.1

3.5 Online Documentation

The online help for Trellis Graphics contains a lot of detail. There is online help for any function in this *Manual*. For example,

```
?xyplot
```

documents `xyplot()`. The general display functions discussed in chapter 7 have many common arguments, so there is special online help for these arguments. Use

```
?trellis.args
```

to get information about arguments for the 2-D displays, and

```
?trellis.3d.args
```

to see argument help for the 3-D displays. Finally, you can see a list of all of the example functions using the online help:

```
?trellis.examples
```

Chapter 4

A Roadmap of Trellis Graphics

4.1 General Display Functions

The Trellis library has a collection of *general display functions* that draw different types of graphs. For example, `xyplot()` makes x-y plots, `dotplot()` makes dot plots, and `wireframe()` makes 3-D wireframe displays. The functions are *general* because they have the full capability of Trellis Graphics including multipanel conditioning.

The general display functions are introduced in chapter 7.

4.2 Common Arguments

There are a set of common arguments that all general display functions employ. The usage of some of these arguments varies, but each has a common purpose across all functions. Many of the general display functions also have arguments that are specific to the types of graphs that they draw.

The common arguments, which appear in the Table of Contents, are discussed in many chapters.

4.3 Panel Functions

Panel functions are a critical aspect of Trellis Graphics. They make it easy to tailor displays to your data even when the displays are quite complicated ones with many panels.

The data region of a panel on a graph resulting from a general display function is a rectangle that just encloses the data. The sole responsibility for drawing in a data region is given to a panel function that is an argument of the general display function. The other arguments of the general display function manage the superstructure of the graph—scales, labels, boxes around the data region, and keys. The panel function manages the symbols, lines, and so forth that encode the data in the data regions.

Panel functions are discussed in chapter 12.

4.4 Core S-PLUS Graphics

Trellis Graphics is implemented in the core S-PLUS graphics. Also, when you write a panel function you use functions and graphics parameters from the core.

Some Core S-PLUS graphics features are discussed in chapter 12.

4.5 Devices and Settings

You need an output device to see a graph. The specification of a screen device was introduced in chapter 3. Of course, you also want to send graphs to printers and to files. Trellis Graphics allows you to do this in many ways.

Sending graphs to files and printers is discussed in chapter 11.

Trellis Graphics has many settings for graph rendering details—plotting symbols, colors, line types and so forth— that are automatically chosen depending on the device you select.

Chapter 13 discusses the Trellis settings.

4.6 Data Structures

The general display functions take in data just like many of the S-PLUS modeling functions such as `lm()`, `aov()`, `glm()`, and `loess()`. This means that there is a heavy reliance on data frames. The Trellis library contains several functions that change data structures of certain types to a data frame, which makes it easier to pass the data on to the general display functions (or, in fact, on to the modeling functions).

Chapter 15 discusses these functions that create data frames.

Chapter 5

Giving Data to General Display Functions

For a graphics function to draw a graph, it needs to know the data on which the drawing is based. This chapter is about arguments to the Trellis drawing functions that allow you to specify the data.

5.1 A Data Set: gas

The data frame `gas` contains two variables from an industrial experiment with 22 runs in which the concentrations of oxides of nitrogen (NOx) in the exhaust of an engine were measured for different settings of equivalence ratio (E).

```
> names(gas)
[1] "NOx" "E"
> dim(gas)
[1] 22  2
```

5.2 formula=

The function `xyplot()` makes an x-y plot, a graph of two numerical variables; the result might be scattered points, curves, or both. `xyplot()` has its own section in chapter 7, but for now we will use it to illustrate how to specify data.

Figure 5.1 is a scatterplot of `gas$NOx` against `gas$E`:

```
xyplot(formula = gas$NOx ~ gas$E)
```

The argument `formula=` specifies the variables that are to be graphed. In this case they are `gas$NOx` and `gas$E`. For `xyplot()`, the variable to the left of the ~ goes on the vertical axis, and the variable to the right of the ~ goes on the horizontal axis. The formula `gas$NOx ~ gas$E` is read as `gas$NOx` "is graphed against" `gas$E`.

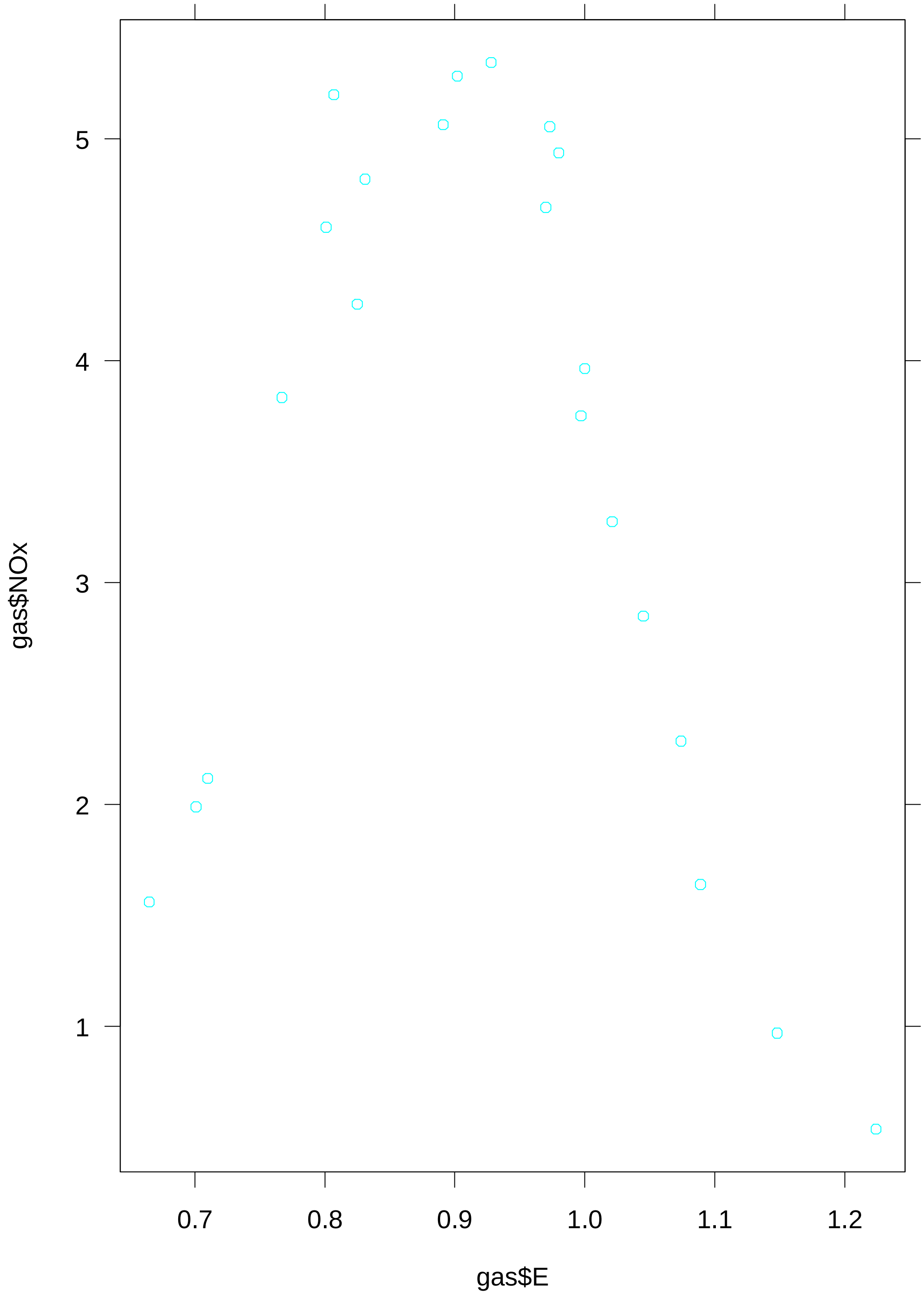


Figure 5.1

The use of `formula=` here is the same as that in the S-PLUS statistical modeling functions such as `lm` and `aov`. To the left or right of the ~ you can use any S-PLUS expression. For example, if you want to graph the log base 2 of `gas$NOx`, you can use the formula

```
log(gas$NOx,base=2) ~ gas$E
```

The argument `formula=` is a special one in Trellis Graphics. It is always the first argument of a general display function such as `xyplot()`. We can omit typing `formula=` provided the formula is the first argument. Thus the expression

```
xyplot(gas$NOx ~ gas$E)
```

also produces figure 5.1. `formula=` is the only argument that should be given by position; all others must be given by name.

Certain single-symbol operators that perform functions in S-PLUS have a special meaning in the formula language (e.g., `+`, `*`, `/`, `|`, and `:`), although Trellis, as we will see, uses only `*` and `|`. If you want to use any of these operators for their conventional meaning in any formula expression— for example, if you want to use `*` as multiplication—you must put the expression inside the identity function `I()` unless it is already given as an argument to a function. Here is an example:

```
log(2*gas$NOx,base=2) ~ I(2*gas$E)
```

We use `I()` on the right of the formula to protect against the `*` in `2*gas$E`, but not on the left because `2*gas$NOx` sits inside a function.

5.3 data=

One annoyance in the use of the above formulas is that we had to continually refer to the data frame `gas`. This is not necessary if we attach `gas` to the search list of databases. We can draw figure 5.1 by

```
attach(gas)
xyplot(NOx ~ E)
```

Another possibility is to use the argument `data=`:

```
xyplot(NOx ~ E, data = gas)
```

In this case, the variables of `gas` are available for use in `formula=` just during the execution of `xyplot()`. The effect is the same as

```
attach(gas)
xyplot(NOx ~ E)
detach(gas)
```

The use of `data=` has another benefit. In the call to `xyplot()` we see explicitly that the dataframe `gas` is being used; this can be helpful for understanding, at some future point, how the graph was produced.

5.4 subset=

Suppose you want to redo figure 5.1 and omit the observations for which `E` is 1.1 or greater. You could do this by

```
xyplot(NOx[E < 1.1] ~ E[E < 1.1], data = gas)
```

But it is a nuisance to repeat the logical subsetting, `E < 1.1`. And the nuisance would be much greater if there were many variables in the formula instead of just two. It is typically easier to use the argument `subset=` instead:

```
xyplot(NOx ~ E, data = gas, subset = E < 1.1)
```

The result is shown in figure 5.2. The argument `subset=` can be a logical or numerical vector.

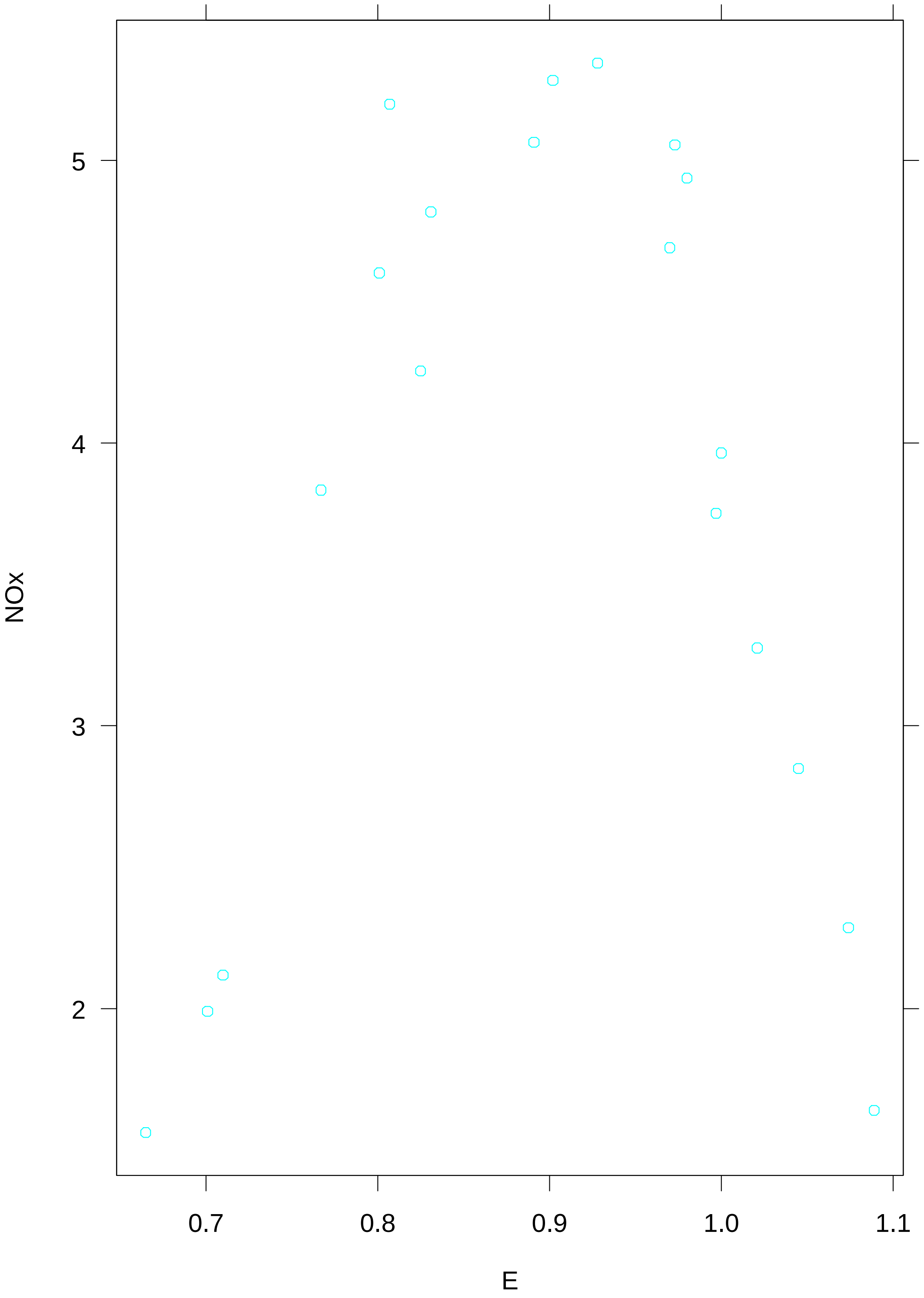


Figure 5.2

5.5 Data Frames

You can keep variables as vectors and draw Trellis displays without using data frames. Still, data frames are very convenient. But datasets are often stored, at least initially, in data structures other than data frames, so we need ways to go from data structures of various types to data frames. Functions to do this are discussed in Chapter 15.

Chapter 6

Aspect Ratio

6.1 The Aspect Ratio of a Graph is a Critical Factor

The aspect ratio of a graph, the height of a panel data region divided by its width, is a critical factor in determining how well a data display shows the structure of the data. See chapter 1 for an example where choosing the aspect ratio to carry out banking to 45° shows information in the data that cannot be seen if the graph is square, that is, has an aspect ratio of 1. More generally, any time we graph a curve, or a scatter of points with an underlying pattern that we want to assess, controlling the aspect ratio is vital. One advance of Trellis Graphics is the direct control of the aspect ratio through the argument `aspect=`.

6.2 aspect=

You can use `aspect=` to set the ratio to a specific value. In figure 6.1, the aspect ratio has been set to 3/4:

```
xyplot(NOx ~ E, data = gas, aspect = 3/4)
```

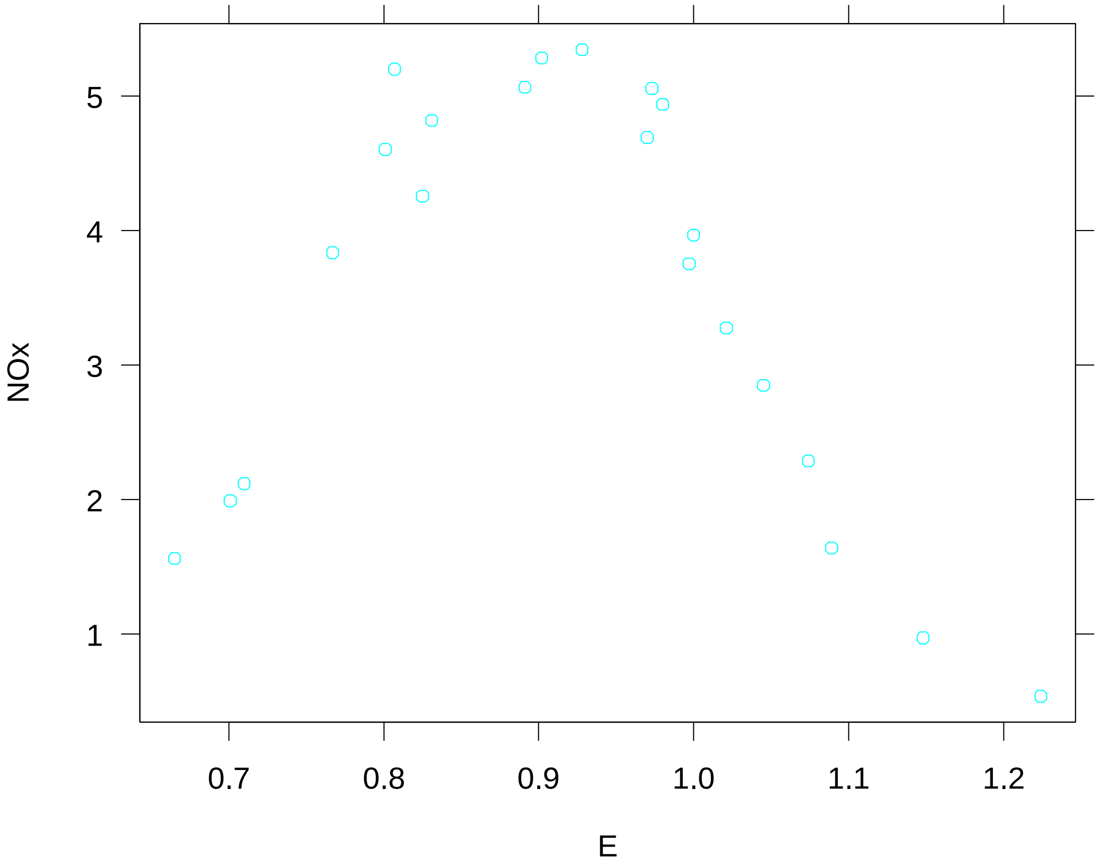


Figure 6.1

Setting `aspect = "xy"` banks line segments to 45°. Here is how it works. Suppose `x` and `y` are data points to be plotted. Consider the line segments that connect successive points. The aspect ratio is chosen so that the absolute values of the slopes of these segments are centered on 45°. This is done in figure 6.2 by the expression

```
xyplot(NOx ~ E, data = gas, aspect = "xy")
```

We have used the data themselves in this example to carry out banking, just to illustrate how it works. The resulting aspect ratio is about 0.4. Ordinarily, though, we should bank based on a smooth underlying pattern in the data; that is, we should bank based on the line segments of a fitted curve. You can do that with Trellis Graphics as well; an example will be given in chapter 16.

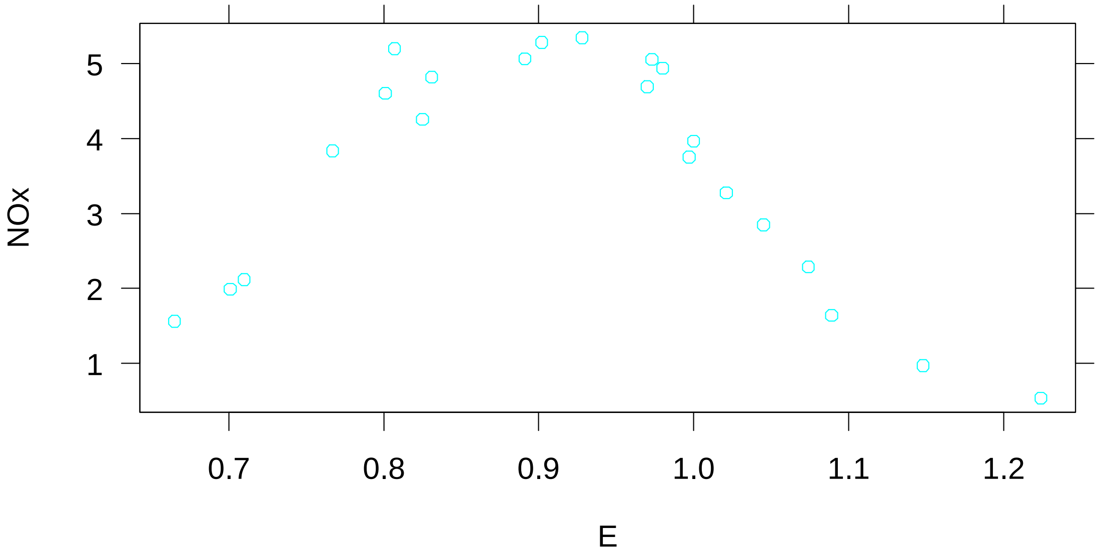


Figure 6.2

Chapter 7

General Display Functions

Each *general display function* draws a particular type of graph. For example, `dotplot()` makes dot plots, `wireframe()` makes 3-D wireframe displays, `histogram()` makes histograms, and `xyplot()` makes x-y plots. This chapter describes a collection of general display functions.

7.1 A Data Set: fuel.frame

The data frame `fuel.frame` contains five variables that measure characteristics of 60 automobile models:

```
> names(fuel.frame)
[1] "Weight"  "Disp."   "Mileage" "Fuel"    "Type"
> dim(fuel.frame)
[1] 60  5
```

The variables are weight, displacement of the engine, fuel consumption in miles per gallon, fuel consumption in gallons per mile, and a classification into type of vehicle. The first four variables are numeric. The fifth variable is a factor:

```
> table(fuel.frame$Type)
 Compact Large Medium Small Sporty Van
      15     3     13    13      9   7
```

7.2 xyplot()

We have already seen `xyplot()` in action in many of our previous examples. This function is a basic graphical method—graphing one set of numerical values on a vertical scale against another set of numerical values on a horizontal scale.

Figure 7.1 is a scatterplot of mileage against weight:

```
xyplot(Mileage ~ Weight, data = fuel.frame, aspect = 1)
```

The variable on the left of the ~ goes on the vertical, or y, axis and the variable on the right goes on the horizontal, or x, axis.

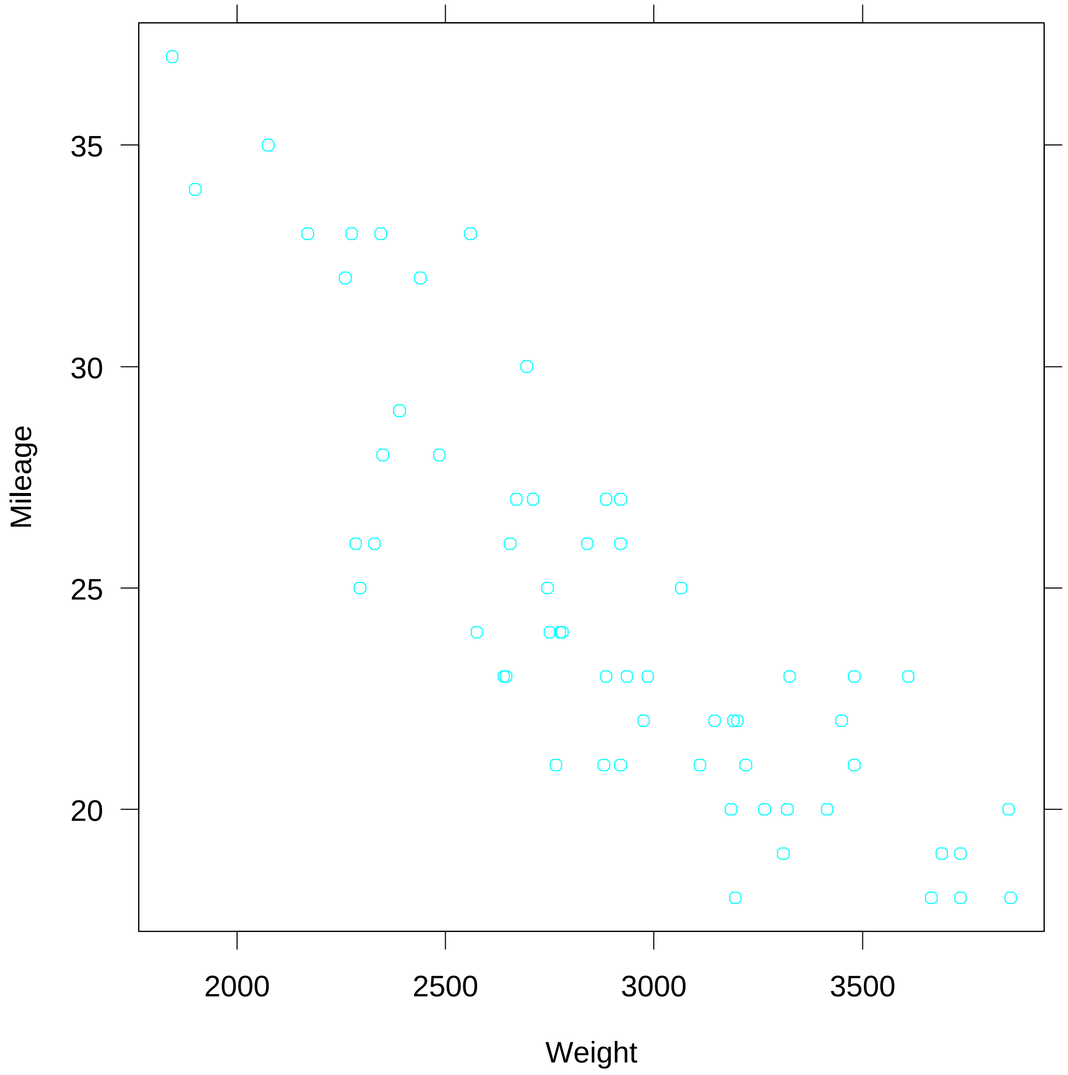


Figure 7.1

7.3 bwplot()

The box and whisker plot, or box plot, is a very clever invention of John Tukey that is widely used for comparing the distributions of several datasets.

Figure 7.2 is a box plot of mileage classified by vehicle type:

```
bwplot(Type ~ Mileage, data = fuel.frame, aspect = 1)
```

The factor `Type` is on the left of the formula because it goes on the vertical axis and the numeric vector `Mileage` is on the right because it goes on the horizontal axis.

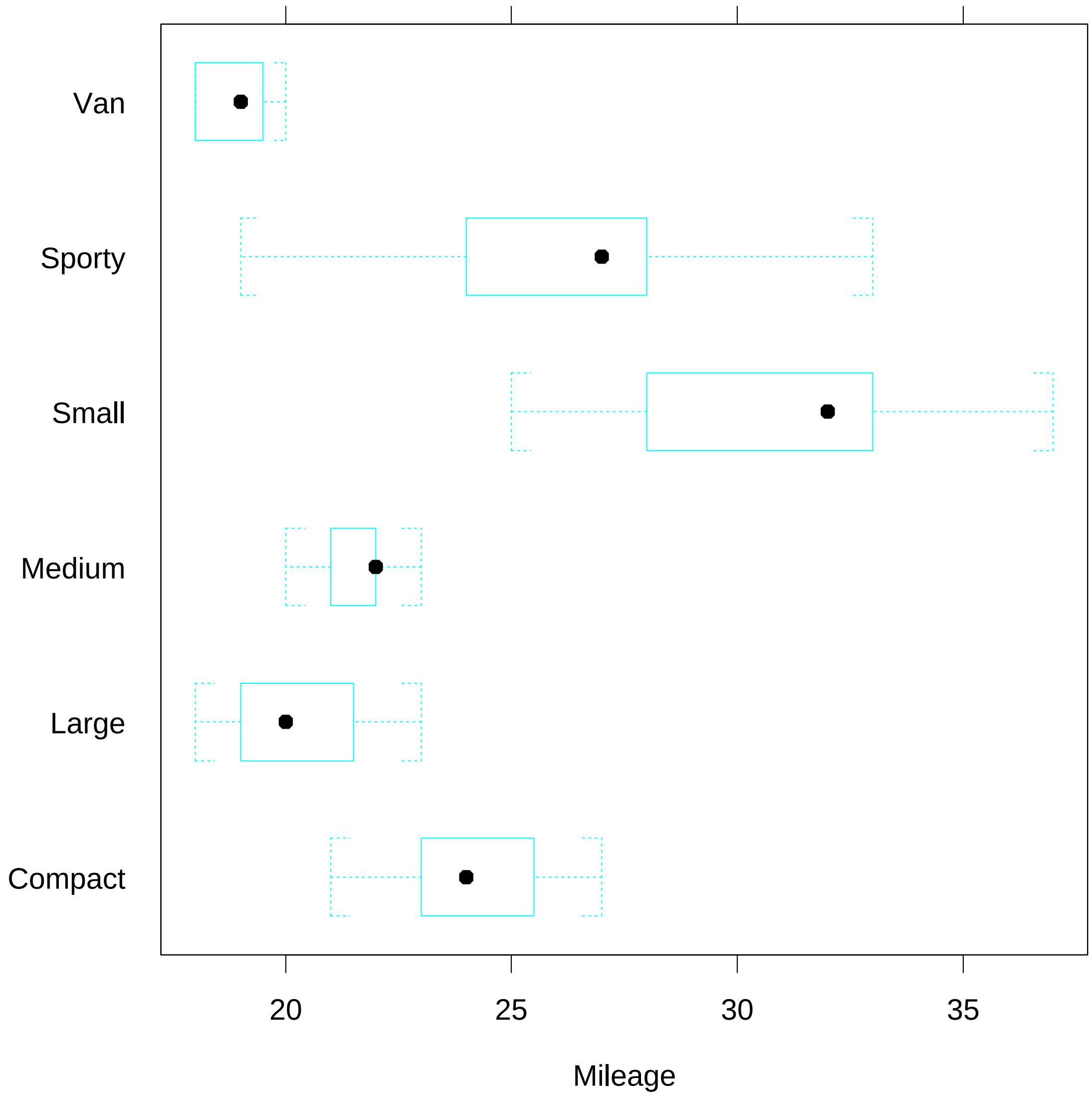


Figure 7.2

7.4 stripplot()

A strip plot, sometimes called a one-dimensional scatterplot, is similar to a box plot in general layout but the individual data points are shown instead of the box plot summary.

Figure 7.3 is a stripplot:

```
stripplot(Type ~ Mileage, data = fuel.frame, aspect = 1,
  jitter = T)
```

Setting `jitter = TRUE` causes some random noise to be added vertically to the points to alleviate the overlap of the plotting symbols. When `jitter = FALSE`, the default, the points for each level lie on a horizontal line.

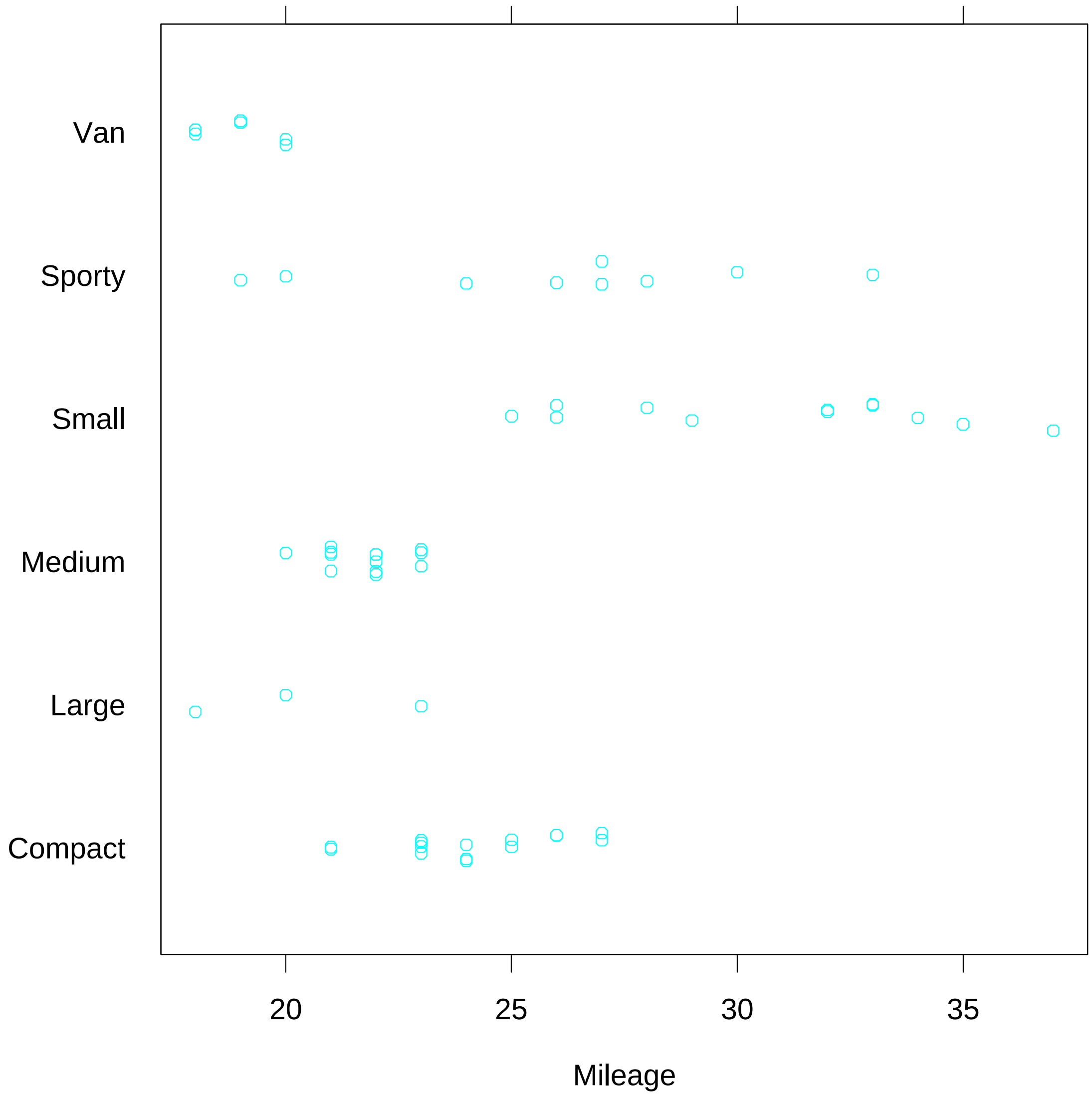


Figure 7.3

7.5 qq()

The quantile-quantile plot, or q-q plot, is an extremely powerful tool for comparing the distributions of two sets of data. The idea is quite simple; quantiles of one dataset are graphed against corresponding quantiles of the other dataset.

The variable `fuel.frame$Type` has five levels:

```
> table(fuel.frame$Type)
 Compact Large Medium Small Sporty Van
      15     3     13    13      9   7
```

Figure 7.4 is a q-q plot comparing the quantiles of mileage for compact cars with the corresponding quantiles for small cars:

```
qq(Type ~ Mileage, data = fuel.frame, aspect = 1,
  subset = (Type == "Compact")|(Type == "Small"))
```

The factor on the left side of the formula must have two levels. The default labels for the two scales are the names of the levels.

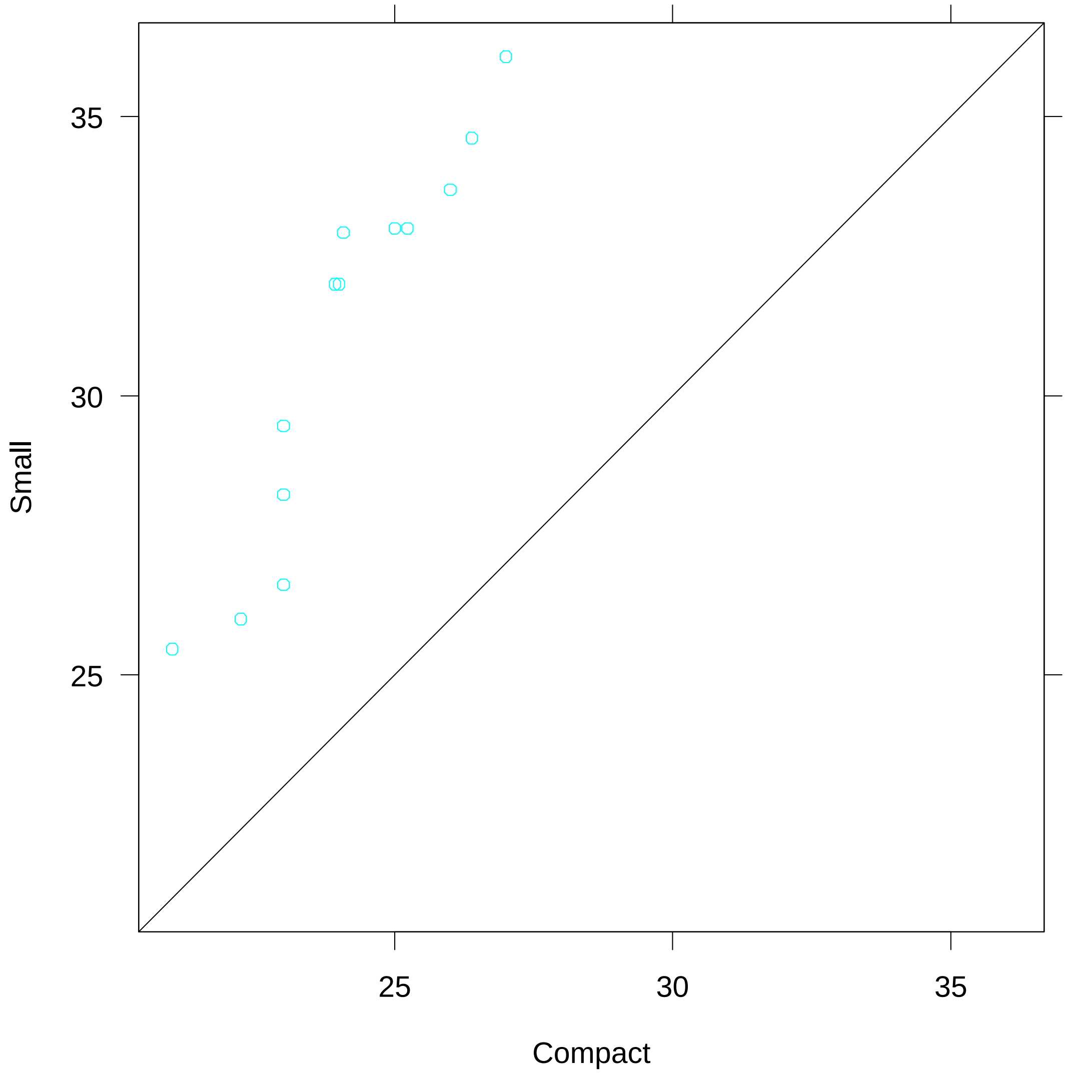


Figure 7.4

7.6 dotplot()

The dot plot, which displays data with labels, provides highly accurate visual decodings, typically far more accurate than other methods for displaying labeled data.

Let us compute the mean mileage for each vehicle type:

```
> mileage.means <- tapply(fuel.frame$Mileage,
+ fuel.frame$Type, mean)
> mileage.means
  Compact    Large   Medium Small Sporty      Van
 24.13333 20.33333 21.76923    31     26 18.85714
```

Figure 7.5 is a dotplot of the log base 2 means:

```
dotplot(names(mileage.means) ~ log(mileage.means, base=2),
  aspect = 1, cex = 1.25)
```

The argument `cex` is passed to the panel function to change the size of the dot of the dot plot in this case; more on this in chapter 12.

Notice that the vehicle types in figure 7.5 are ordered, from bottom to top, by the order of the elements of the vector `mileage.means`. So to change the order on the graph we simply change the order of the vector elements. For example, if you wanted the graph to show the values from smallest to largest going from bottom to top, you could redefine `mileage.means`:

```
mileage.means <- sort(mileage.means)
```

and then make the plot.

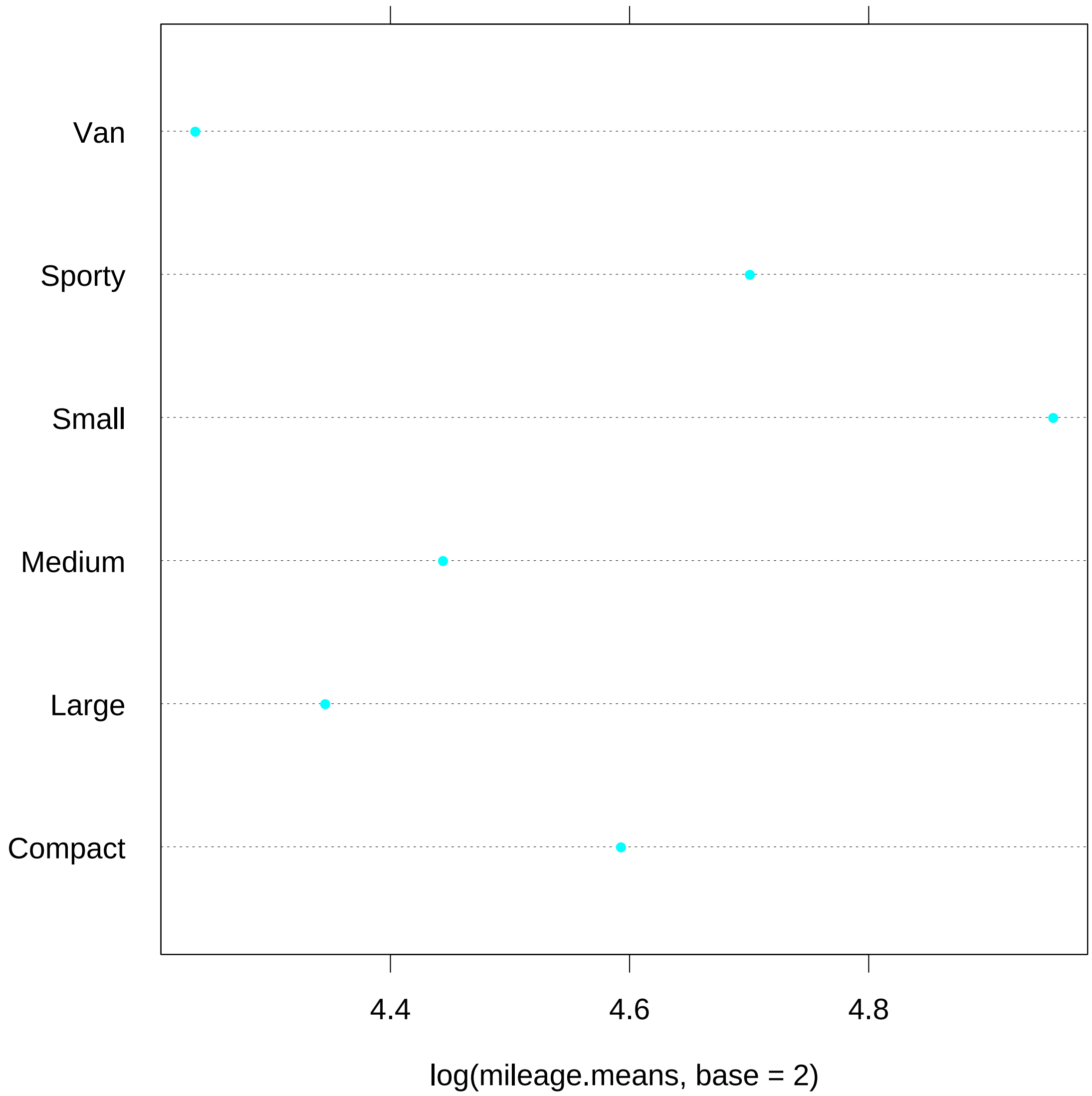

Van
Sporty
Small
Medium
Large
Compact
4.4
4.6
4.8
log(mileage.means, base = 2)

Figure 7.5

7.7 barchart()

Overall, dot plots are a more effective display method than bar charts, avoiding some of the perceptual problems of bar charts. Still, there are circumstances where bar charts are harmless.

Figure 7.6 is a bar chart of the mileage means (without logs):

```
barchart(names(mileage.means)~mileage.means, aspect = 1)
```

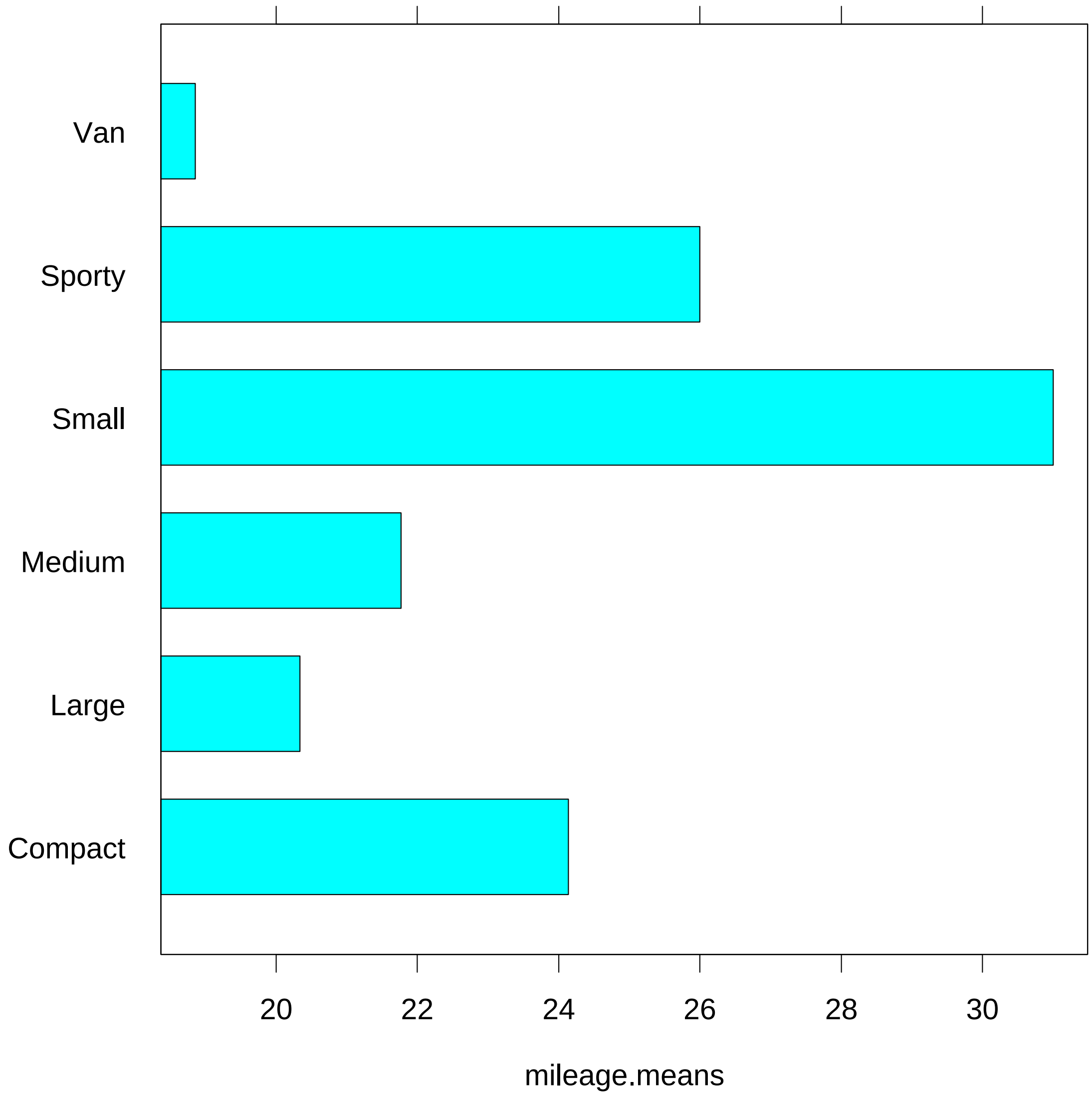

Van
Sporty
Small
Medium
Large
Compact
20
22
24
26
28
30
mileage.means

Figure 7.6

7.8 piechart()

Pie charts have severe perceptual problems. Experiments in graphical perception have shown that compared with dot plots, they convey information far less reliably. But if you want to display some data, and perceiving the information is not so important, then a pie chart is fine.

Figure 7.7 is a pie chart of the mileage means:

```
piechart(names(mileage.means)~mileage.means, aspect = .5)
```

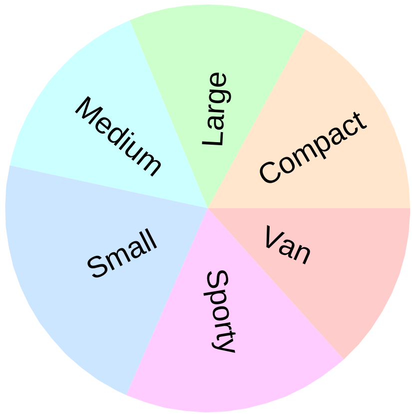


Figure 7.7

7.9 qqmath()

Normal probability plots, or normal q-q plots, are the single most powerful tool for determining if the distribution of a set of measurements is well approximated by the normal distribution.

Figure 7.8 is a normal probability plot of the mileages for small cars:

```
qqmath(~Mileage,  data = fuel.frame, aspect = 1,
  subset = (Type == "Small"))
```

That is, the ordered data are graphed against quantiles of the standard normal distribution.

Note that the formula for `qqmath()` is used in a way unlike any of the previous examples. Only one data object appears in the formula, to the right of the ~, because this graphical method utilizes only one data object.

`qqmath()` can also make probability plots for other distributions. It has an argument `distribution` whose input is any function that computes quantiles. The default is `qnorm`. If we used

```
qqmath(~Mileage,  data = fuel.frame, aspect = 1,
  subset = (Type == "Small"),
  distribution = qexp)
```

the result would be an exponential probability plot. Note that the name of the function appears as the default label on the horizontal scale of the plot.

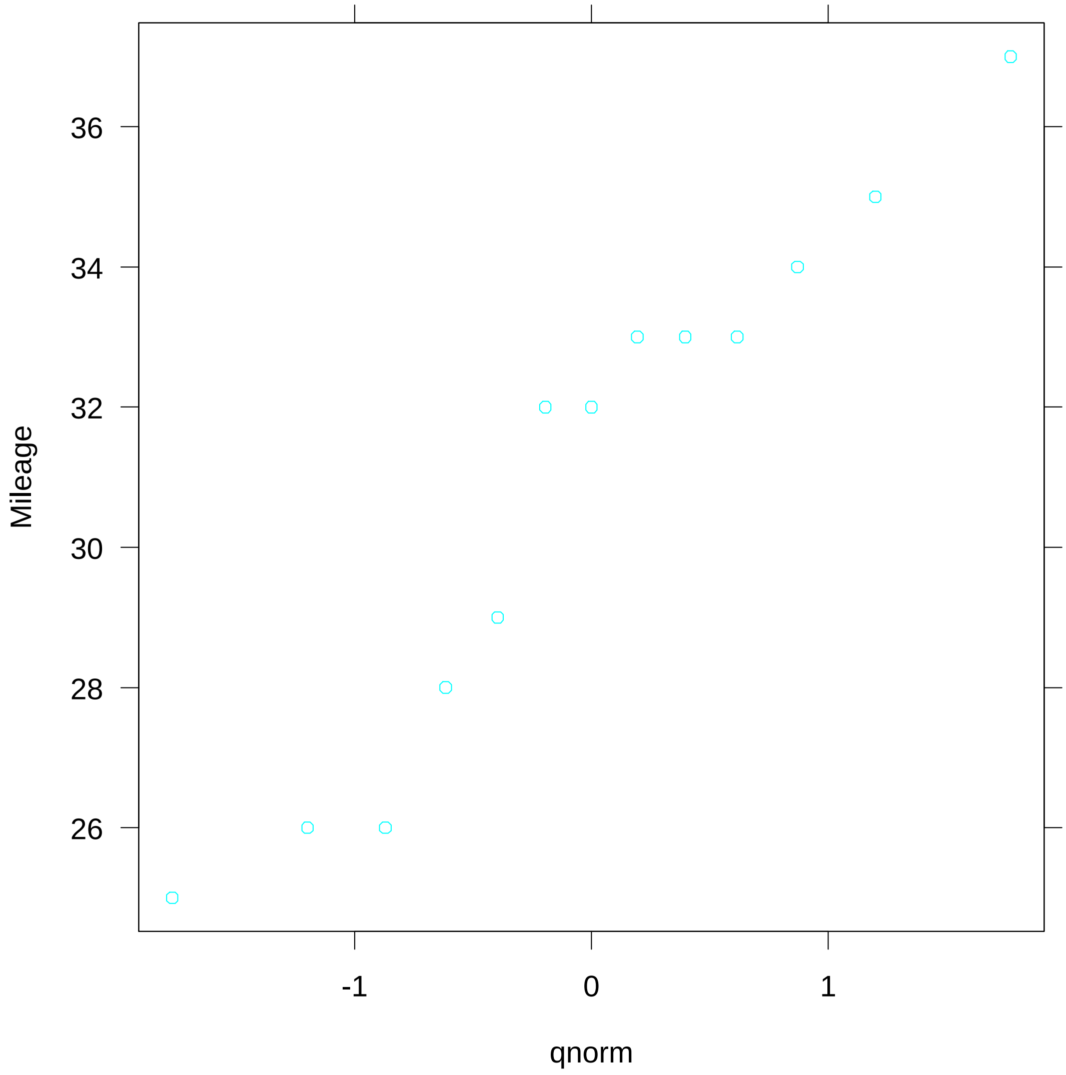


Figure 7.8

7.10 histogram()

A histogram can be useful for showing the distribution of a single set of data, but two or more histograms are typically not nearly as powerful as a box plot or q-q plot for comparing data distributions.

Figure 7.9 is a histogram of mileage:

```
histogram(~Mileage, data = fuel.frame, aspect = 1,
  nint = 10)
```

The argument `nint` determines the number of intervals. The histogram algorithm chooses the intervals to make the bar widths be simple numbers while trying to make the number of intervals as close to `nint` as possible.

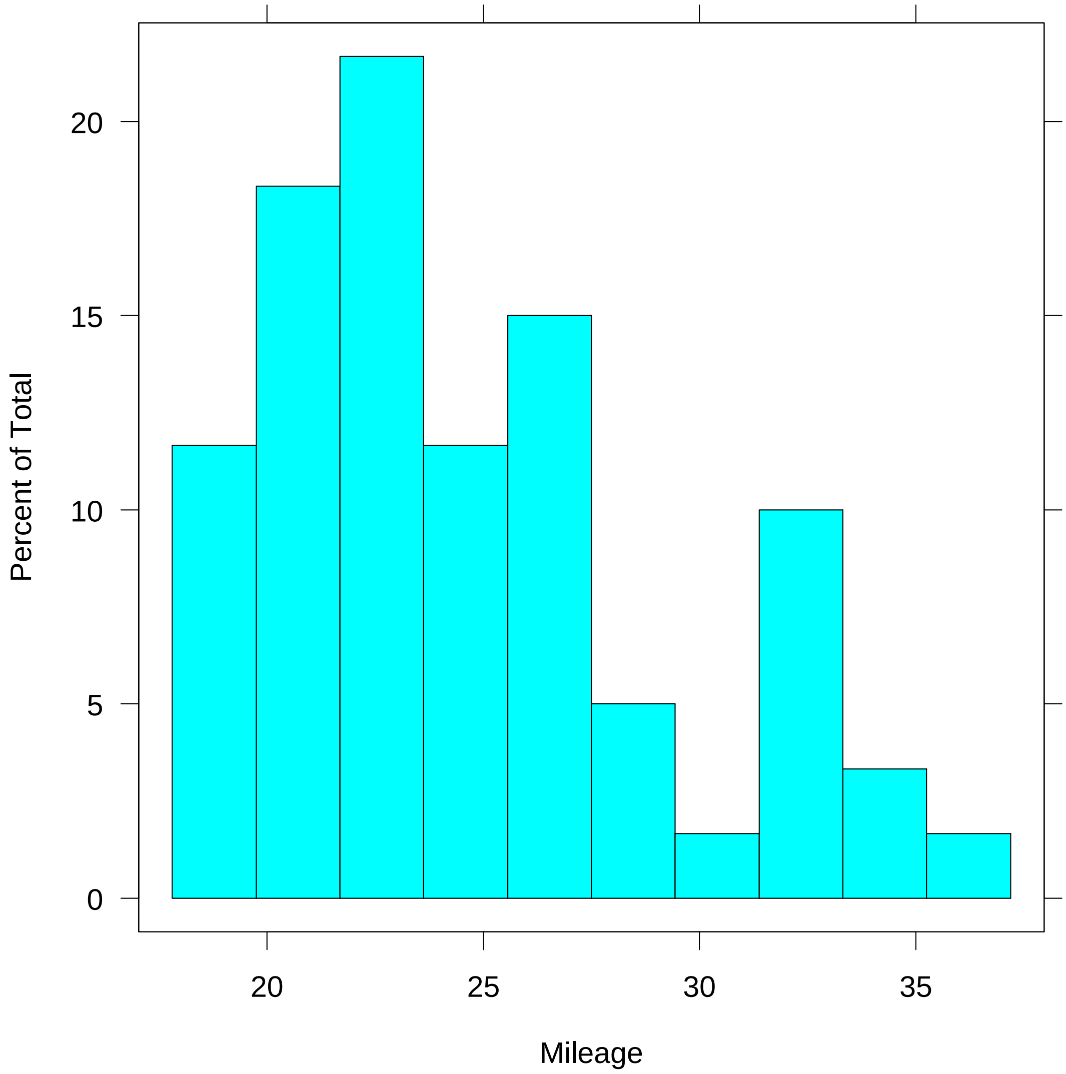


Figure 7.9

7.11 densityplot()

Like histograms, density plots can be of help in understanding the distribution of a single set of data, but box plots and q-q plots typically give more incisive comparisons of distributions.

Figure 7.10 is a density plot of mileage:

```
densityplot( ~ Mileage, data = fuel.frame, aspect = "xy",
  width = 5)
```

The argument `width` controls the width of the smoothing window in the same units as the data, mpg here; as the width increases, the smoothness increases.

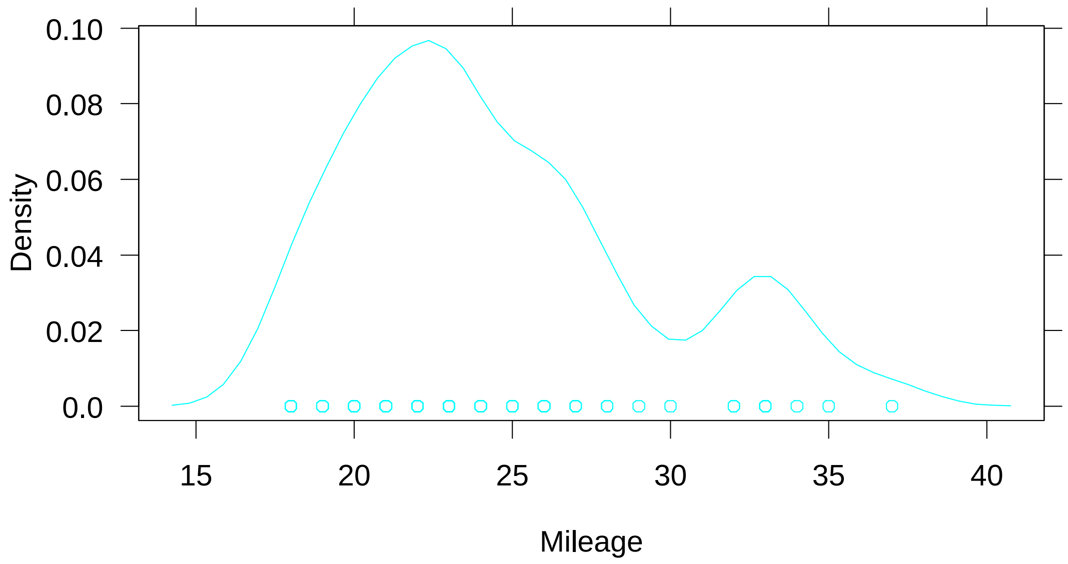


Figure 7.10

7.12 splom()

The scatterplot matrix is an exceedingly powerful tool for displaying measurements of three or more variables.

Figure 7.11 is a scatterplot matrix of the variables in `fuel.frame`:

```
splom( ~ fuel.frame)
```

Note that the factor `Type` has been converted to a numeric variable and plotted just like the other variables, which are numeric. The six levels of `Type` simply take the values 1 to 6 in this conversion.

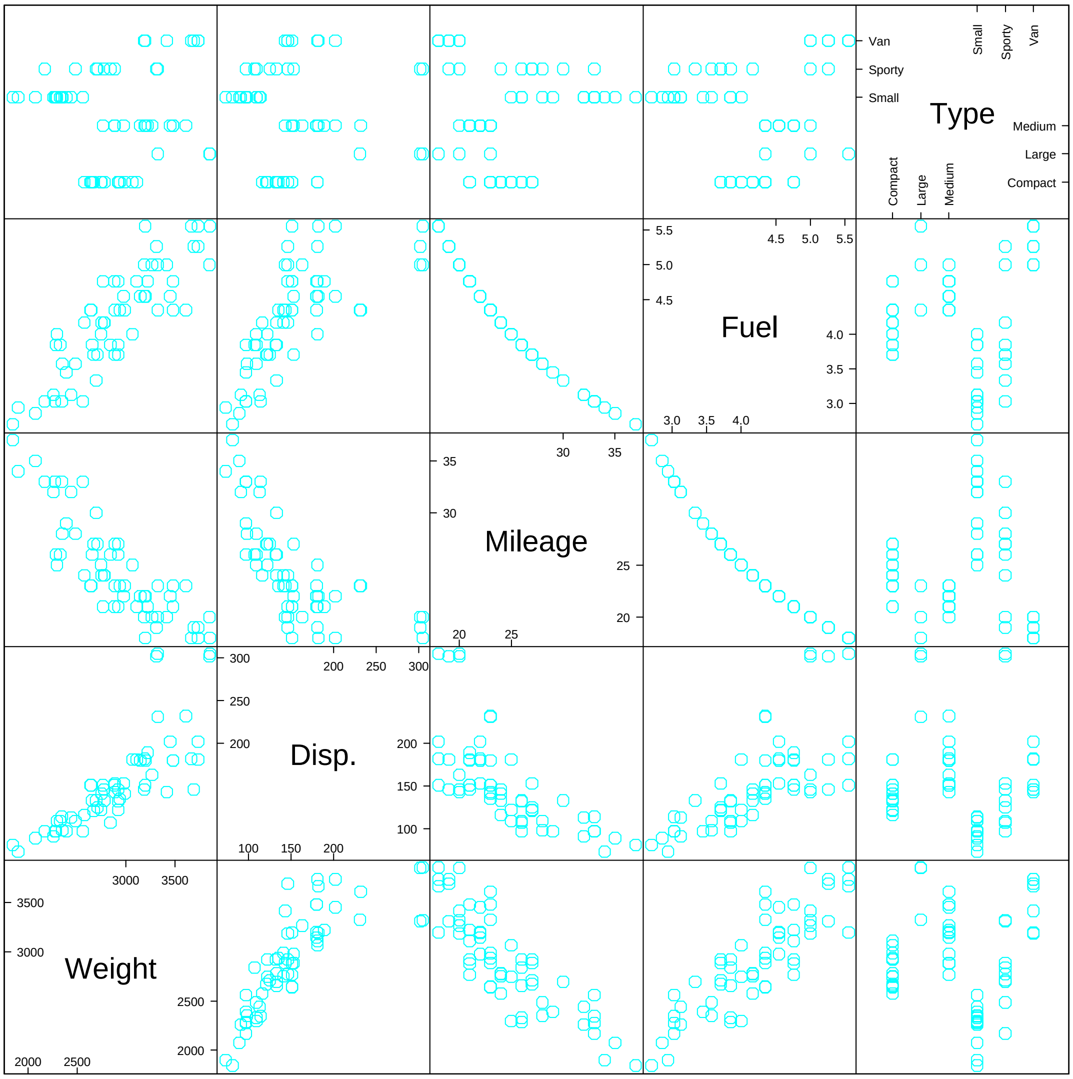

Type
Van
Sporty
Small
Medium
Large
Compact
Fuel
Mileage
Disp.
Weight

Figure 7.11

7.13 parallel()

Parallel coordinates are an interesting method, but it is unclear at the time of this writing whether they have the power to uncover structure that is not more readily apparent using other graphical methods.

Figure 7.12 is a parallel coordinates display of the variables in `fuel.frame`:

```
parallel( ~ fuel.frame)
```

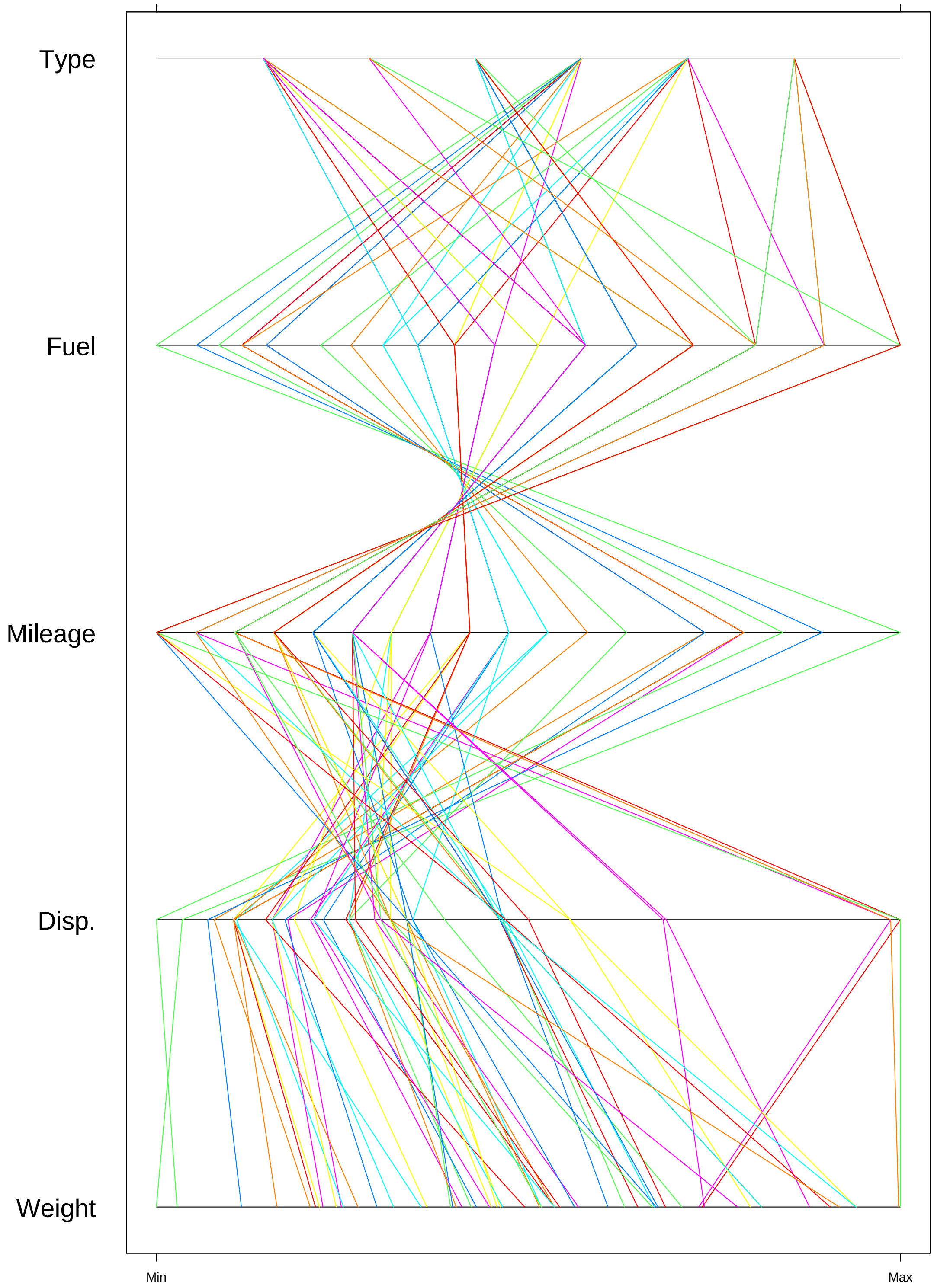


Figure 7.12

7.14 A Data Set: gauss

To further illustrate the general display routines, we will compute a function of two variables over a grid.

```
datax <- rep(seq(-1.5, 1.5, length = 50), 50)
datay <- rep(seq(-1.5, 1.5, length = 50), rep(50, 50))
dataz <- exp(-(datax^2 + datay^2 + datax*datay))
gauss <- data.frame(datax, datay, dataz)
```

Thus `dataz` is the exponential of a quadratic function defined over a 50 by 50 grid; in other words, the surface is proportional to a bivariate normal density.

7.15 contourplot()

Contour plots are helpful displays for studying a function, $f(x, y)$, when we have no need to study the conditional dependence of f on x given y or of f on y given x. Conditional dependence is revealed far better by multipanel conditioning.

Figure 7.13 is a contour plot of the gaussian surface:

```
contourplot(dataz ~ datax * datay, data = gauss, aspect = 1,
  at = seq(.1, .9, by = .2))
```

The argument `at` specifies the values as which the contours are to be computed and drawn. If the argument is not specified, reasonable default values are chosen.

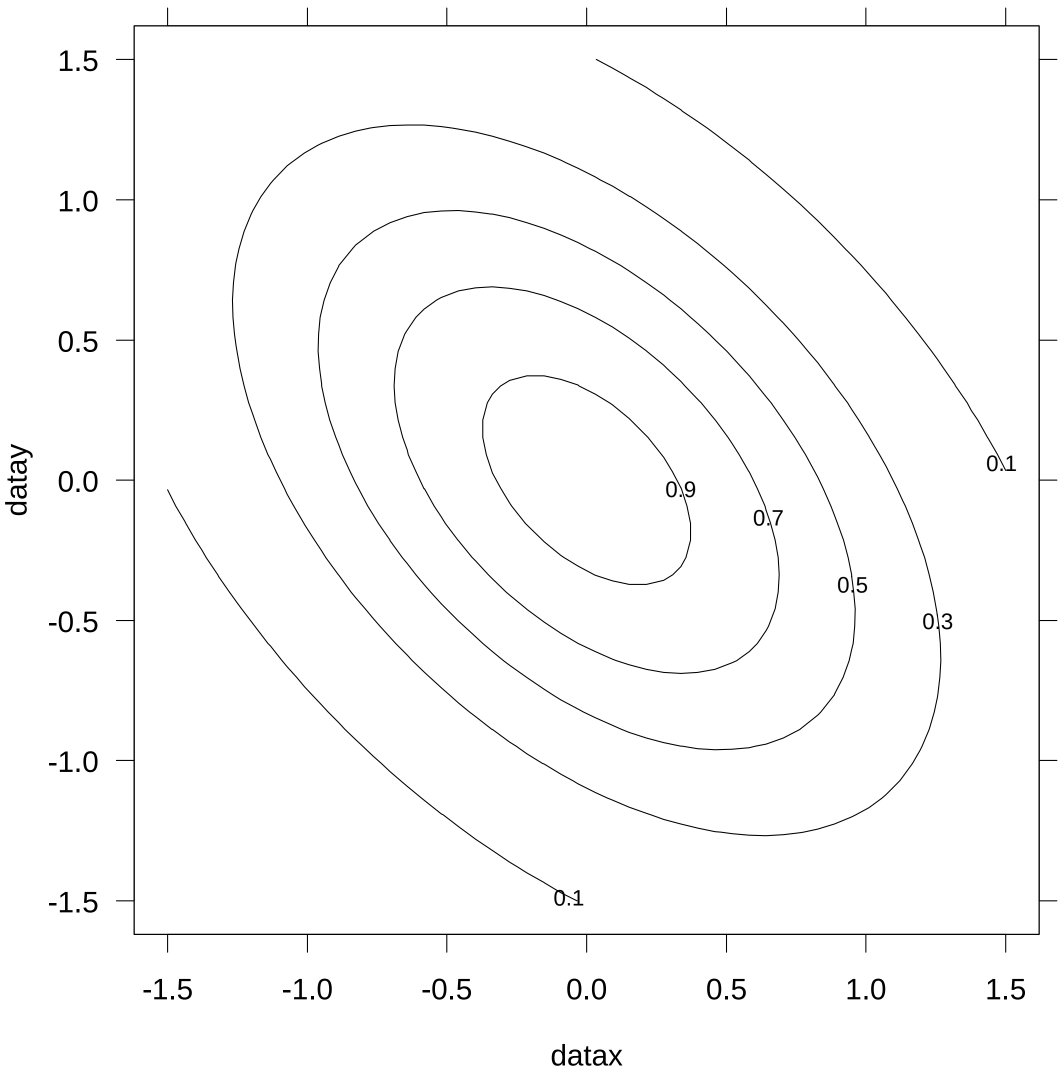

1.5
1.0
0.5
0.0
-0.5
-1.0
-1.5
datay
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
datax
0.1
0.3
0.5
0.7
0.9
0.1

Figure 7.13

7.16 levelplot()

Level plots are also helpful displays for studying a function, $f(x, y)$. They are no better than contour plots when the function is simple, but often are better when there is much fine detail, for example, many peaks and valleys.

Figure 7.14 is a level plot of the gauss surface:

```
levelplot(dataz ~ datax * datay, data = gauss, aspect = 1,
  cuts = 6)
```

The values of the surface are encoded by color, a gray scale in this case. For devices with full color, the scale goes from pure magenta to white and then to pure cyan. If the device does not have full color, a gray scale is used.

For a levelplot, the range of the function values is divided into intervals and each interval is assigned a color. A rectangle centered on each grid point is given the color of the interval containing the value of the function at the grid point. In figure 7.14 there are six intervals. The argument `cuts` specifies the number of breakpoints between intervals.

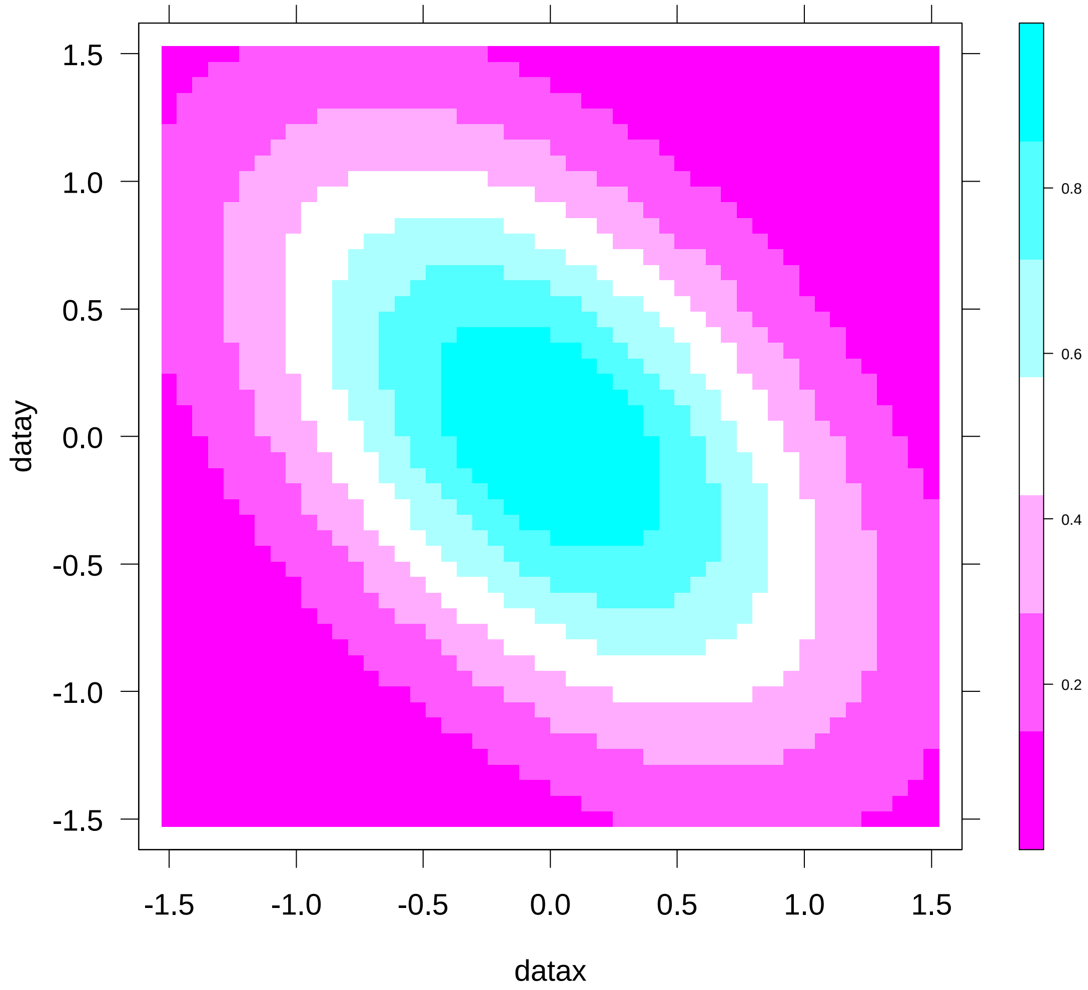


Figure 7.14

7.17 wireframe()

Wireframe displays can be quite useful for displaying $f(x, y)$ when we have no need to study conditional dependence, which is revealed far better by multipanel conditioning.

Figure 7.15 is a 3-D wireframe plot of the gauss surface:

```
wireframe(dataz ~ datax * datay, data = gauss, drape = F,
  screen = list(z = 45, x = -60, y = 0))
```

The arrows point in the direction of increasing values of the variables.

The argument `screen` is a list. The three components of the list—`x`, `y`, and `z`—refer to screen axes. The first component is horizontal and the second is vertical, both in the plane of the screen. The third component is perpendicular to the screen. The surface is rotated about these axes in the order given in the list. Here is how it worked for figure 7.15. The surface began with `datax` as the horizontal screen axis, `datay` as the vertical, and `dataz` as the perpendicular. The origin was at the lower left in the back. First, the surface was rotated 45° about the perpendicular screen axis, where a positive rotation is counterclockwise. Then, there was a $-60°$ rotation about the horizontal screen axis, where a negative rotation brings the picture at the top of the screen away from the viewer and the bottom toward the viewer. Finally, there was no rotation about the vertical screen axis; had there been one with a positive number of degrees, then the left side of the picture would have moved toward the viewer and the right away.

If `drape = T`, a color encoding is added to the surface using the same encoding method of the level plot.

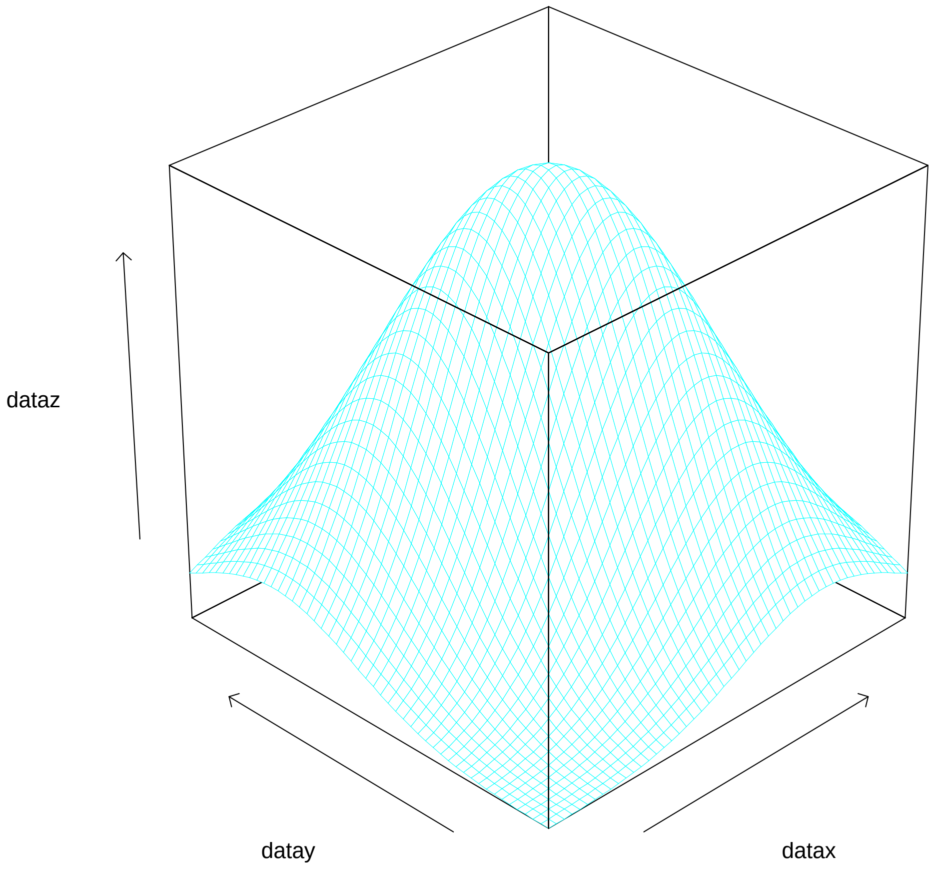


Figure 7.15

7.18 cloud()

A static 3-D plot of a scatter of points is typically not effective because the depth cues are insufficient to give a strong 3-D effect. Still, on rare occasions, such a plot can be useful, sometimes as a presentation or teaching tool.

Figure 7.16 is a 3-D scatterplot of the first three variables in the data frame `fuel.frame`:

```
cloud(Mileage ~ Weight * Disp., data = fuel.frame,
  screen = list(z = -30, x = -60, y = 0),
  xlab = "W",
  ylab = "D",
  zlab = "M")
```

The behavior of the argument `screen` is the same as that for `wireframe`. We have used three additional arguments to specify scale labels; such labeling will be discussed in chapter 10.

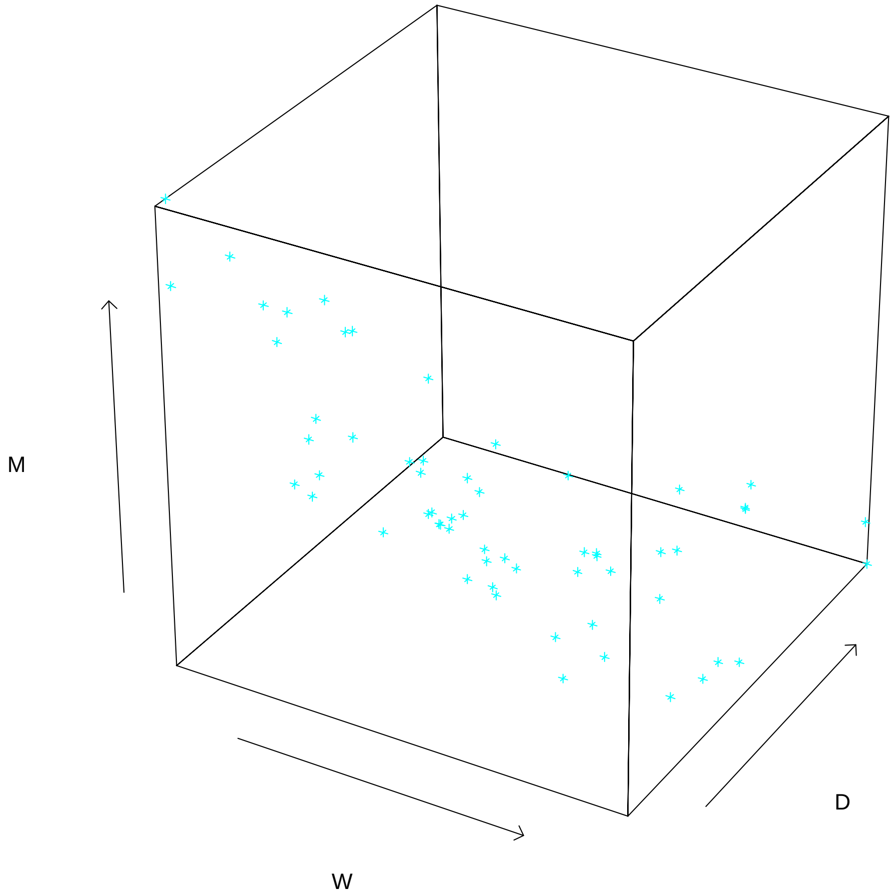


Figure 7.16

7.19 The Display Functions and Their Formulas

The following listing of the general display functions and their formulas is instructive because it shows certain conventions and consistencies in the formula mechanism:

Graph One Numerical Variable Against Another

```
xyplot(numeric1 ~ numeric2)
```

Compare the Sample Distributions of Two or More Sets of Data

```
bwplot(factor ~ numeric)
stripplot(factor ~ numeric)
qq(factor ~ numeric)
```

Graph Measurements with Labels

```
dotplot(character ~ numeric)
barchart(character ~ numeric)
piechart(character ~ numeric)
```

Graph the Sample Distribution of One Set of Data

```
qqmath(~numeric)
histogram(~numeric)
densityplot(~numeric)
```

Graph Multivariate Data

```
splom(~data.frame)
parallel(~data.frame)
```

Graph a Function of Two Variables Evaluated on a Grid

```
contourplot(numeric1 ~ numeric2 * numeric3)
levelplot(numeric1 ~ numeric2 * numeric3)
wireframe(numeric1 ~ numeric2 * numeric3)
```

Graph Three Numerical Variables

```
cloud(numeric1 ~ numeric2 * numeric3)
```

Chapter 8

Arranging Several Graphs On One Page

Several graphs, made separately by Trellis display functions, can be displayed on a single page. There is one restriction. None of the individual graphs may be a multipanel conditioning display with more than one page.

8.1 print()

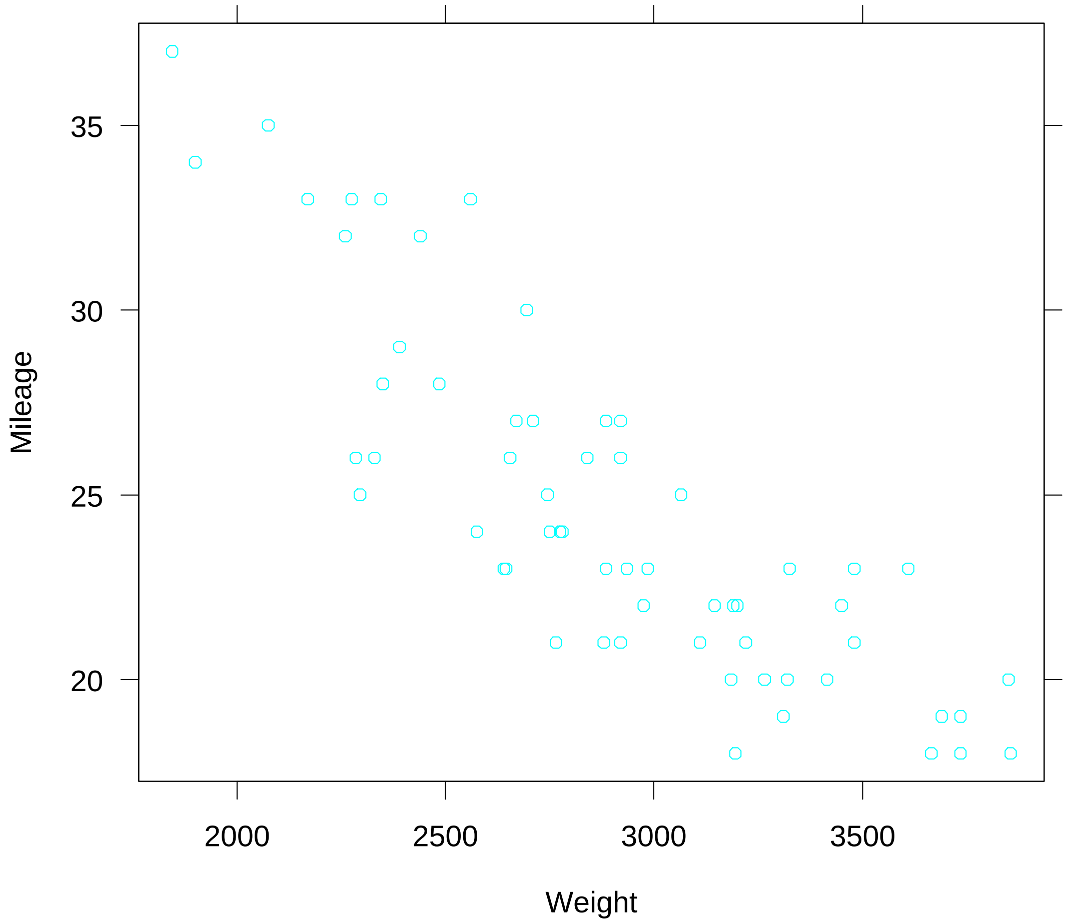
Figure 8.1 shows two graphs arranged on one page:

```
attach(fuel.frame)
box.plot <- bwplot(Type ~ Mileage)
scatter.plot <- xyplot(Mileage ~ Weight)
detach()

print(box.plot, position = c(0,0,1,.4), more = T)
print(scatter.plot, position = c(0,.35,1,1))
```

The argument `position` specifies the position of each graph on the page using a page coordinate system in which the lower left corner of the page is (0, 0) and the upper right corner is (1, 1). The *graph rectangle* is the portion of the page allocated to a graph. `position` takes a vector of four numbers; the first two numbers are the coordinates of the lower left corner of the graph rectangle, and the second two numbers are the coordinates of the upper right corner. The argument `more=` has been give a value of `T`, which says that more drawing is coming.

Notice that in the above example the graph rectangles overlap somewhat. Here is the reason. The graph contains margins (empty space) around the edges of the graph. But in arranging graphs on a page, we might well want to overlap margin space to use the page space as efficiently as possible.


35
30
25
20
Mileage
2000
2500
3000
3500
Weight

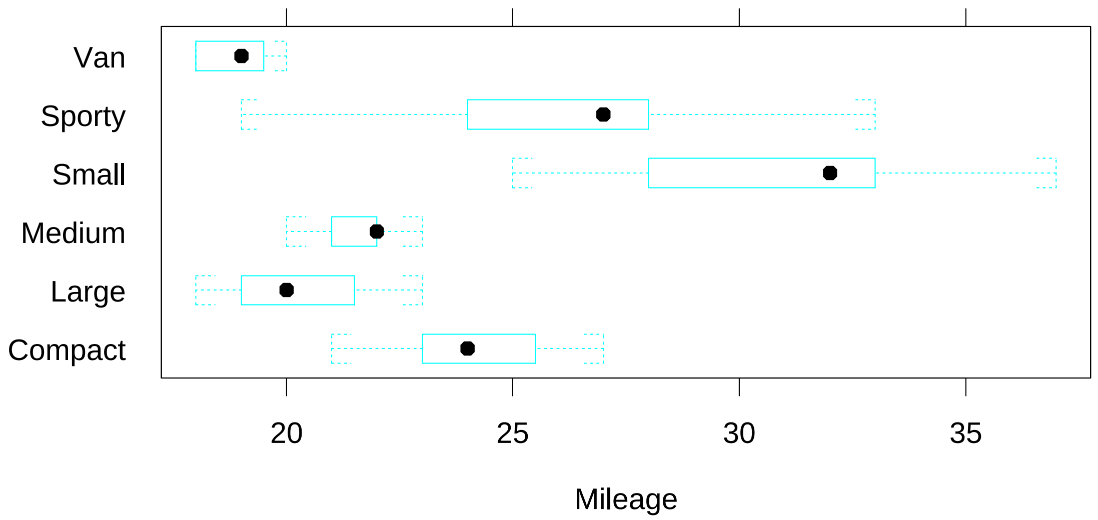

Van
Sporty
Small
Medium
Large
Compact
20
25
30
35
Mileage

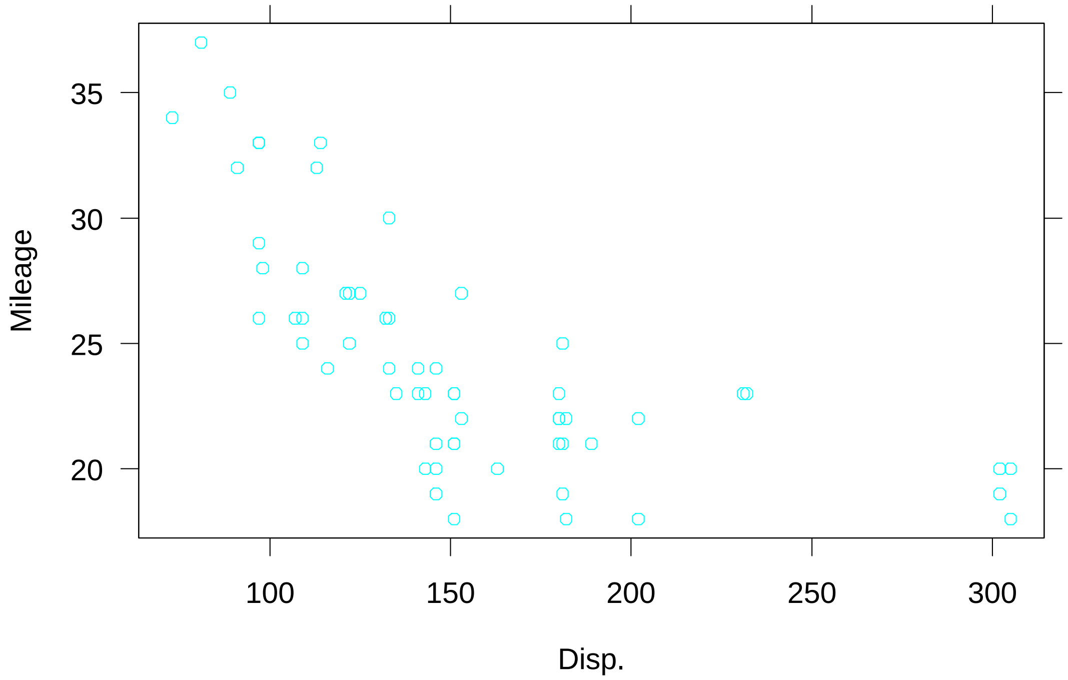
Figure 8.1

Figure 8.2 illustrates another argument, `split=`, that provides a different method for arranging the plots on the page:

```
attach(fuel.frame)
scatter.plot <- xyplot(Mileage ~ Weight)
other.plot <- xyplot(Mileage ~ Disp.)
detach()

print(scatter.plot, split = c(1,1,1,2), more = T)
print(other.plot, split = c(1,2,1,2))
```

`split=` takes a vector of four values. The last two define an array of subregions in the graphics region. In our example, the array has one column and two rows for both plots. The first two values of `split=` prescribe the subregion in which the current plot is to be drawn.


Mileage
35
30
25
20
100
150
200
250
300
Disp.

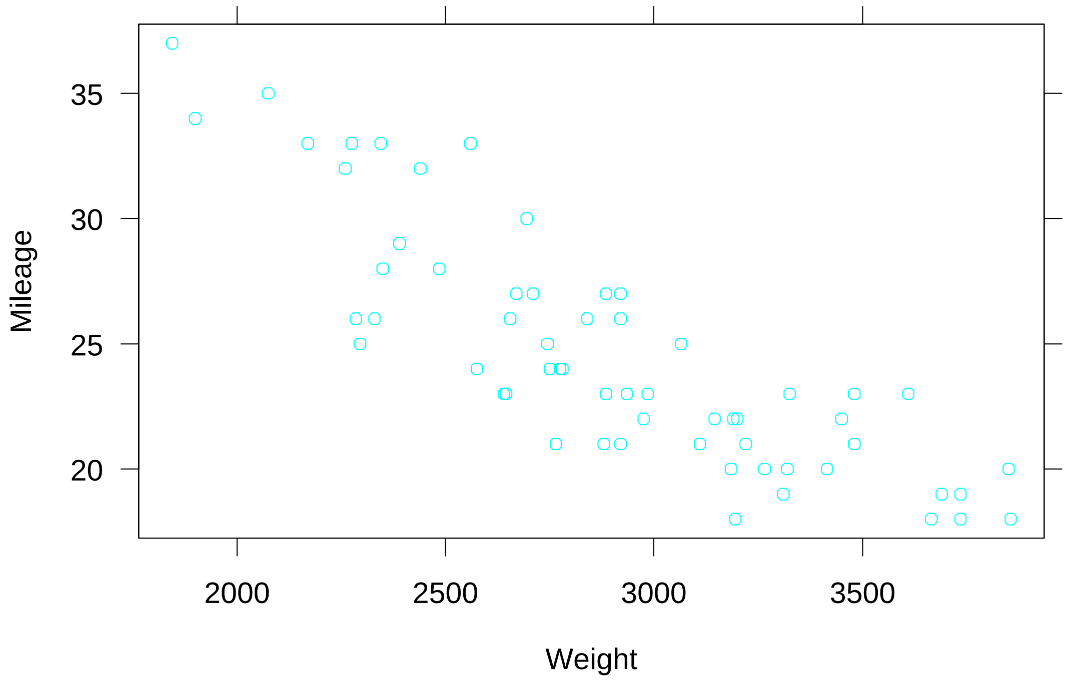

Mileage
35
30
25
20
2000
2500
3000
3500
Weight

Figure 8.2

Chapter 9

Multipanel Conditioning

9.1 A Data Set: barley

The data frame `barley` contains data from the barley experiment discussed in section 1.1.

```
> names(barley)
[1] "yield"   "variety" "year"    "site"
```

The first of these four variables is numeric, and the remaining three are factors. The experiment was run in the state of Minnesota in the 1930s. At six sites, ten varieties of barley were grown in each of two years. The data collected for the experiment are the yields in bushels/acre for all combinations of site, variety, and year, so there are $6 \times 10 \times 2 = 120$ observations.

9.2 About Multipanel Display

Figure 9.1 uses multipanel conditioning to display the barley data. Each panel displays the yields of the ten varieties for one year at one site; variety is graphed along the vertical scale and yield is graphed along the horizontal scale. For example, the lower left panel displays values of variety and yield for Grand Rapids in 1932. The *panel variables* are yield and variety and the *conditioning variables* are year and site.

9.3 formula=

Figure 9.1 was made by the following command:

```
dotplot(variety ~ yield | year * site, data = barley)
```

The `|` is read as "given". Thus the formula is read as `variety` "is graphed against" `yield` "given" `year` and `site`. Thus a simple use of `formula=` creates a complex multipanel display.

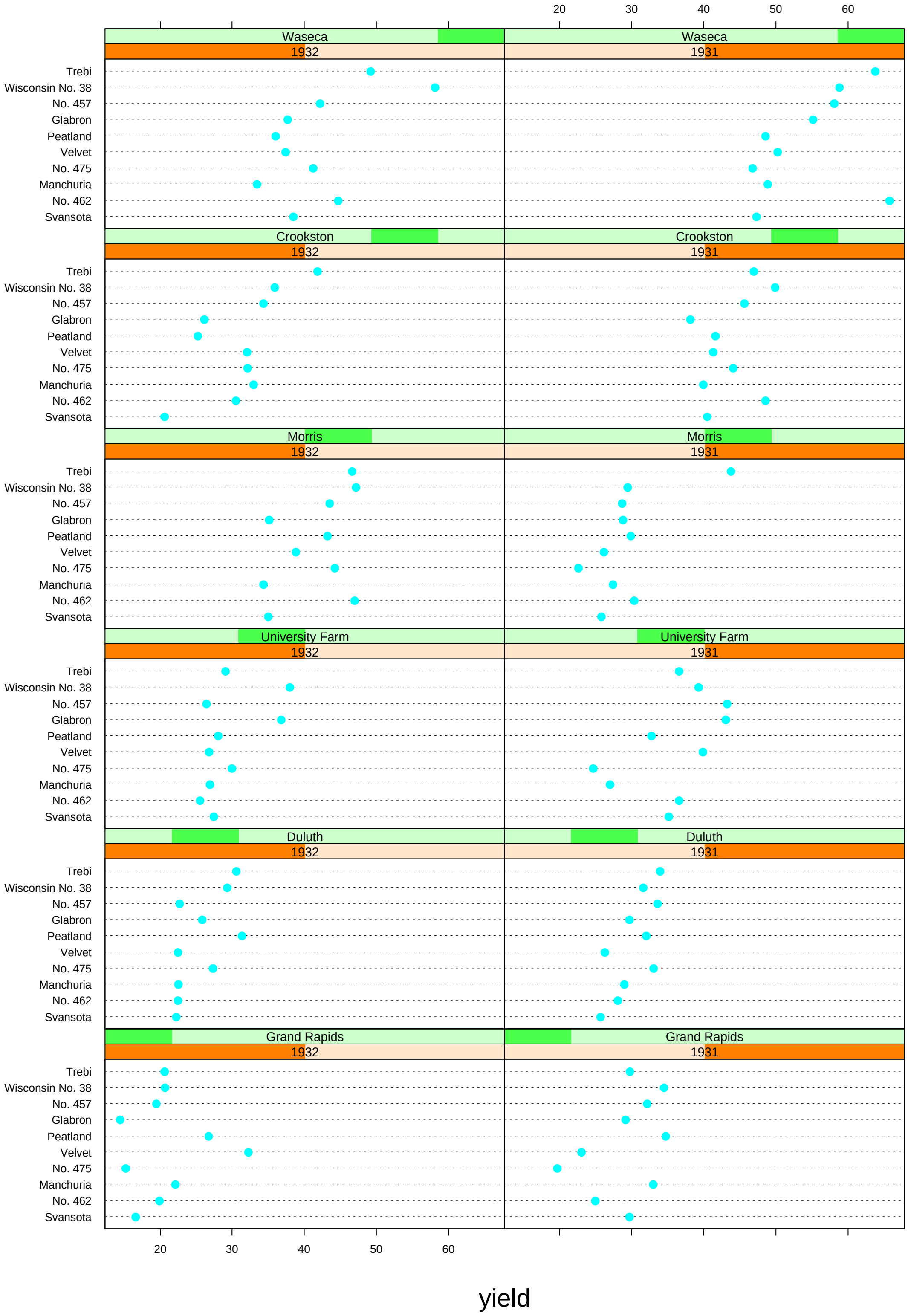

Waseca
1932
Waseca
1931
Crookston
1932
Crookston
1931
Morris
1932
Morris
1931
University Farm
1932
University Farm
1931
Duluth
1932
Duluth
1931
Grand Rapids
1932
Grand Rapids
1931
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
yield

Figure 9.1

9.4 Columns, Rows, and Pages

A multipanel conditioning display is a three-way rectangular array laid out into columns, rows, and pages. In figure 9.1 there are two columns, six rows and one page. The numbers of columns, rows, and pages are selected by an algorithm that attempts of fill up as much of the graphics region as possible subject to certain constraints. As we will see in section 9.6, there is an argument `layout=` that allows you to choose the numbers.

9.5 Packet Order and Panel Order

In the above formula, the conditioning variable `year` appeared first and `site` appeared second. This gives an explicit ordering to the conditioning variables. Each of these variables is a factor with levels:

```
> levels(barley$year)
[1] "1932" "1931"

> levels(barley$site)
[1] "Grand Rapids"    "Duluth"          "University Farm"
[4] "Morris"          "Crookston"       "Waseca"
```

The levels of each factor are ordered by their order of appearance in the `levels` attribute. As we will discuss shortly, we can control the order by making the factor an *ordered factor*.

A *packet* is information sent to a panel for display. For figure 9.1, each packet includes the values of variety and yield for a particular combination of year and site. Packets are ordered by the orderings of the conditioning variables and their levels; the levels of the first conditioning variable vary the fastest, the levels of the second conditioning variable vary the next fastest, and so forth. For figure 9.1, the order of the packets is

```
1932 Grand Rapids
1931 Grand Rapids
1932 Duluth
1931 Duluth
1932 University Farm
1931 University Farm
1932 Morris
1931 Morris
1932 Crookston
1931 Crookston
1932 Waseca
1931 Waseca.
```

The panels of a multipanel display are also ordered. The bottom left panel is panel one. From there we move fastest through the columns, next fastest through the rows, and the slowest through the pages. The panel ordering rule is like a graph, not like a table; the origin is at the lower left and as we move either from left to right or from bottom to top, the panel order increases. The following shows the panel order for figure 9.1, which has two columns, six rows, and one page:

```
11 12
 9 10
 7  8
 5  6
 3  4
 1  2
```

In Trellis Graphics, packets are assigned to panels according to the packet order and the panel order. Packet 1 goes in panel 1, packet 2 goes into panel 2 and so forth. In figure 9.1, the two orderings result in the year variable changing along the columns and the site variable changing along the rows. Note that as the levels for one of these factors increase, the darkened bars in the strip label for the factor move from left to right.

9.6 layout=

Multipanel conditioning is a powerful tool for understanding how a response depends on two or more explanatory variables. In such an analysis, it is typically important to make as many displays as necessary to have each explanatory variable appear at least once as a panel variable. In figure 9.1 variety, an explanatory variable, appears as a panel variable.

We will make a new display with site as a panel variable. The argument `layout=` specifies the numbers of columns, rows, and pages:

```
dotplot(site ~ yield | year * variety, data = barley,
  layout = c(2,5,2))
```

The result is shown in figure 9.2, the first page, and in figure 9.3, the second page.

If we do not specify `layout=`, Trellis Graphics chooses the numbers of columns, rows, and pages by a layout algorithm. The algorithm takes into account the aspect ratio, the number of packets, the number of conditioning variables, and the number of levels of each conditioning variable. It chooses the numbers to maximize the size of the graph within the graphics region.

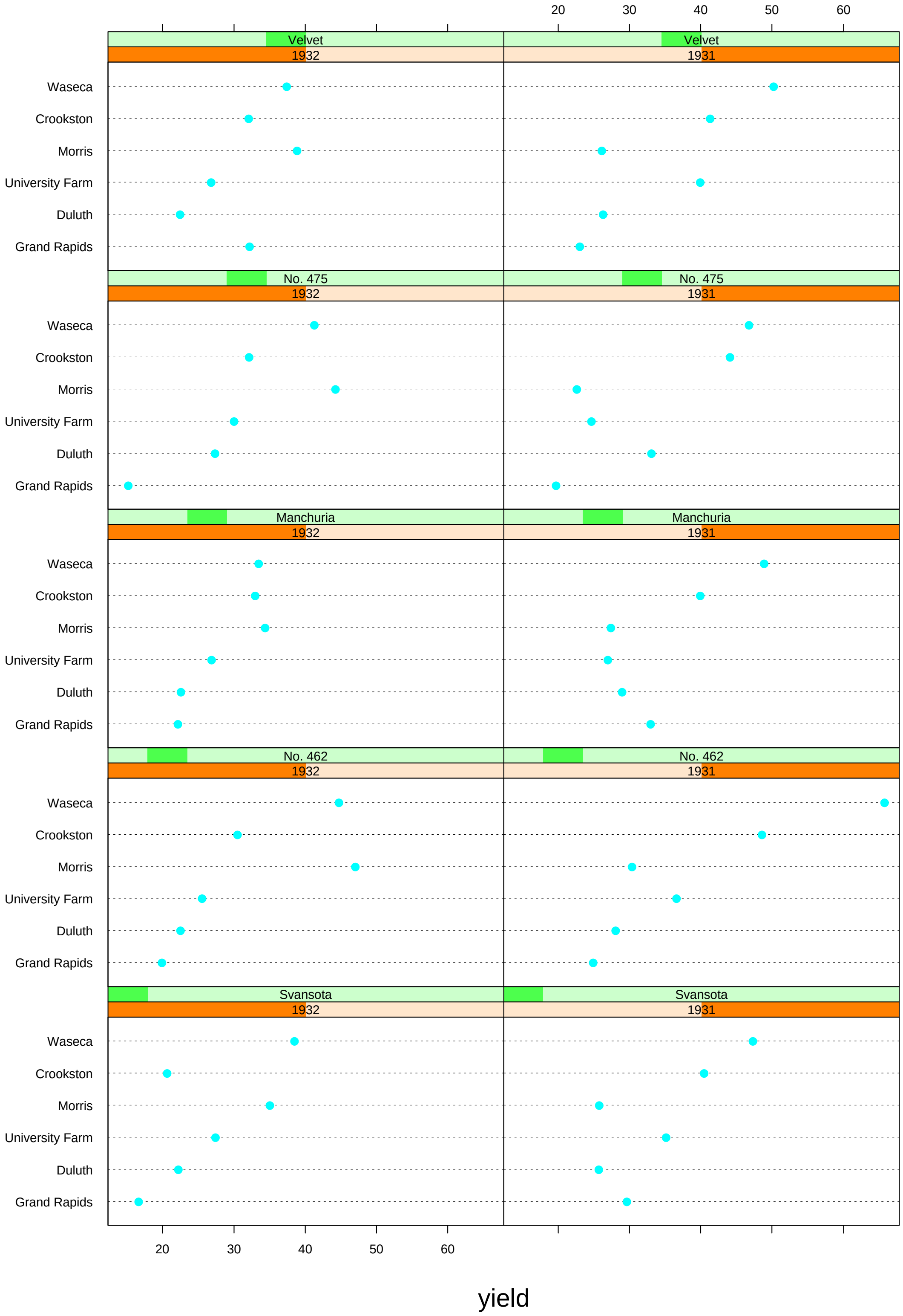

Velvet
1932
1931
No. 475
Manchuria
No. 462
Svansota
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
20
30
40
50
60
yield

Figure 9.2

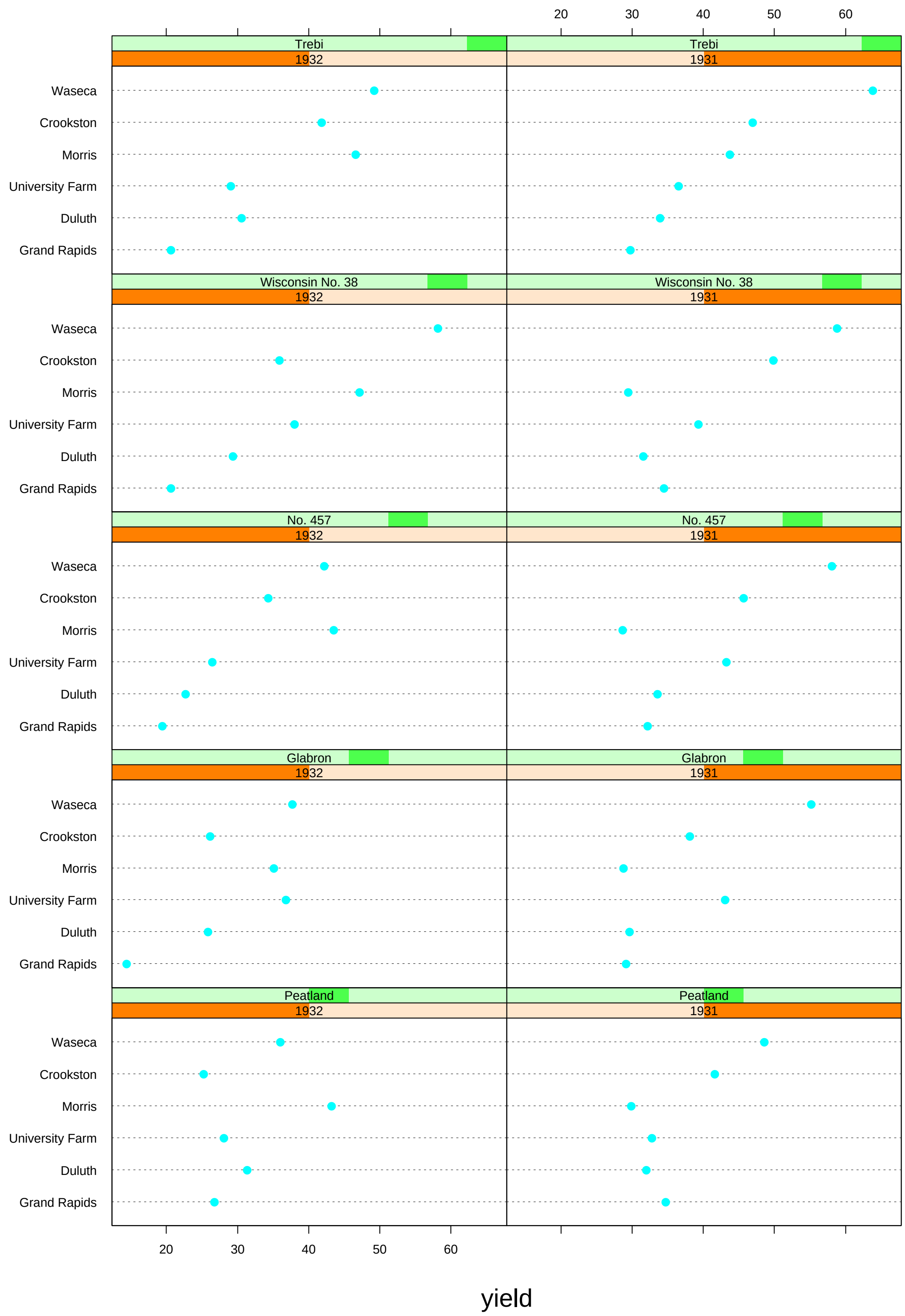

Trebi
1932
Trebi
1931
Wisconsin No. 38
1932
Wisconsin No. 38
1931
No. 457
1932
No. 457
1931
Glabron
1932
Glabron
1931
Peatland
1932
Peatland
1931
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
20
30
40
50
60
yield

Figure 9.3

9.7 Main-Effects Ordering: reorder.factor()

For the barley data, the explanatory variables are categorical. The dataset for each is a factor. (Since there are only two years, the year variable is treated as a factor rather than a numeric vector.) For each factor, consider the median yield for each level. For example, for variety, the level medians are

```
> variety.medians <- tapply(barley$yield,
+ barley$variety, median)

> variety.medians
 Svansota No. 462 Manchuria  No. 475 Velvet Peatland
    28.55   30.45  30.96667 31.06667  32.15 32.38334
 Glabron  No. 457 Wisconsin No. 38 Trebi
    32.4 33.96666            36.95  39.2
```

The barley displays in figures 9.1 to 9.3 use an important display method: *main-effects ordering of levels*. This greatly enhances our ability to perceive effects. Consider figure 9.1. On each panel, the varieties are ordered from bottom to top by the variety medians; Svansota has the smallest median and Trebi has the largest. The site panels have been ordered from bottom to top by the site medians; Grand Rapids has the smallest median and Waseca has the largest. Finally, the year panels are ordered from left to right by the year medians; 1932 has the smaller median and 1931 has the larger.

This median ordering is achieved by making the dataset for each explanatory variable an ordered factor, where the levels are ordered by the medians. For example, suppose `variety` started out as a factor without the median ordering. We get the ordered factor through the following:

```
barley$variety <- ordered(barley$variety,
  levels = names(sort(variety.medians)))
```

Main effects ordering is so important and is carried out so often that Trellis Graphics includes a function `reorder.factor()` to carry it out. Here, it is used to reorder `variety`:

```
barley$variety <- reorder.factor(barley$variety,
 barley$yield, median)
```

The first argument is the factor to be reordered, the second is the data on whose main effects the reordering is based, and the third argument is the function to be applied to the second argument to compute main effects.

9.8 Controlling the Pages of a Multipage Display

If a multipage display is sent to a screen device, the default behavior is for the pages to be drawn in succession; in other words, a page is overwritten by the drawing of its successor. This gives you little time to look at any but the last page. You can control the page flow by

```
par(ask = TRUE)
```

S-PLUS queries you before each page is drawn; hit return to go to the next page.

The problem with this method is that you cannot go backward to look at an earlier page. Another solution, however allows it. Simply specify `postscript` as the device, and then use a PostScript screen reader such as Ghostview to look at the output. Such readers allow easy movement through PostScript pages.

9.9 Summary: How to Lay Out a Multipanel Display

To lay out a multipanel display in a certain way you specify the following:

- An ordering of the conditioning variables by the order you enter them in the argument `formula=`
- An ordering of the levels of each factor, possibly by creating an ordered factor
- The number of columns, rows, and pages through the argument `layout=`.

9.10 A Data Set: ethanol

The data frame `ethanol` contains three variables from an industrial experiment with 88 runs:

```
> names(ethanol)
[1] "NOx" "C"   "E"
> dim(ethanol)
[1] 88  3
```

The concentrations of oxides of nitrogen (NOx) in the exhaust of an engine were measured for different settings of compression ratio (C) and equivalence ratio (E). These measurements were part of the same experiment that produced the measurements in the data frame `gas` introduced in section 5.1.

9.11 Conditioning On Discrete Values of a Numeric Variable

For the barley data, the explanatory variables are factors, so it is natural to condition on the levels of each factor. This is not the case for the ethanol data; both explanatory variables, C and E, are numeric. Suppose for the ethanol data, that we want to graph NOx against E given C. The variable C has five unique values; in other words, the variable, while numeric, is discrete:

```
> table(ethanol$C)
 7.5   9 12 15 18
  22 17 14 19 16
```

It makes sense then to condition on the unique values of C. Figure 9.4 does this:

```
xyplot(NOx ~ E | C, data = ethanol, aspect = 1/2)
```

When a numeric variable is used as a conditioning variable in the argument `formula=`, then conditioning is automatically carried out on the sorted unique values. In other words, the levels of the variable in such a case are the unique values. The order of the levels is from smallest to largest. For C, the first level is 7.5, the second is 9, and so forth. Thus the first packet includes values of NOx and E for C = 7.5, the second packet includes the values for C = 9, and so forth. As before, the packets fill the panels according to the packet order and the panel order. In figure 9.4, the values of C, which are indicated by the thin darkened bars in the strip labels, increase from bottom to top.

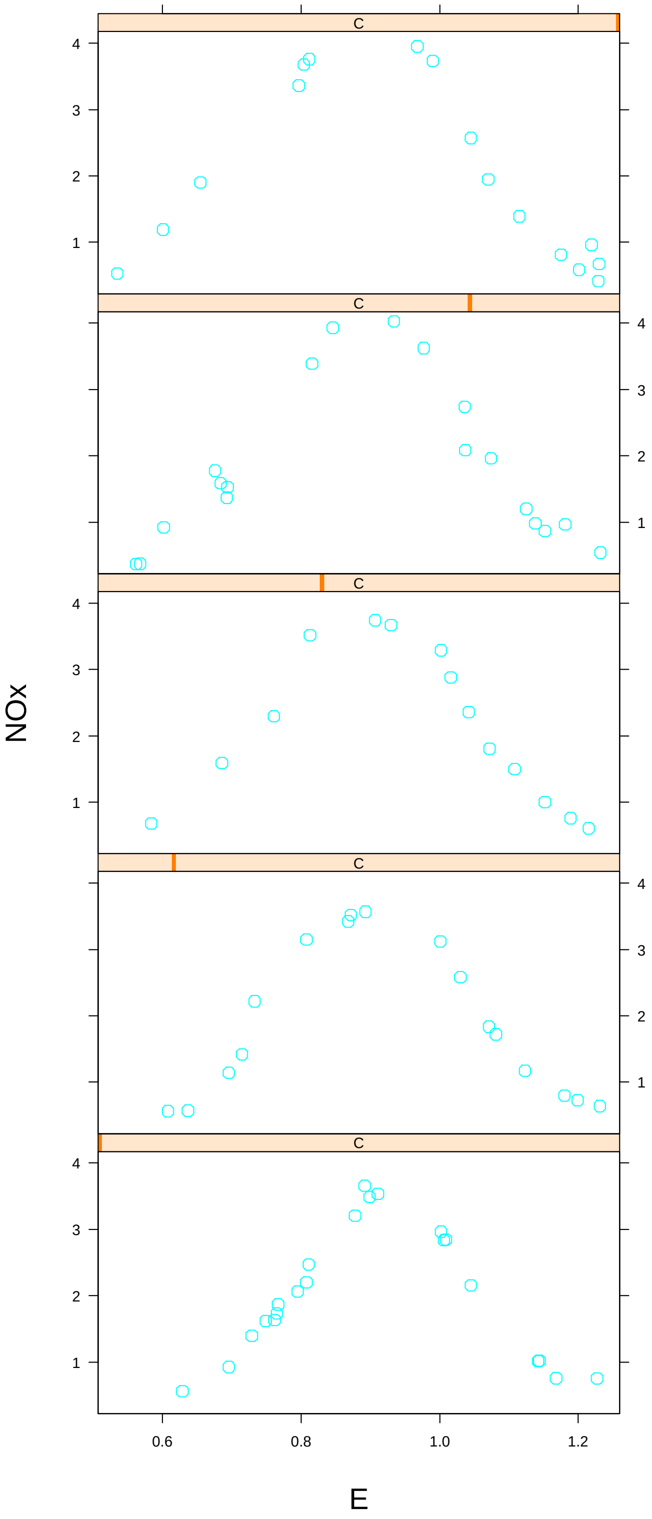


Figure 9.4

9.12 Conditioning On Intervals of a Numeric Variable

For the ethanol data we graphed NOx against E given C in figure 9.4. We would like to see NOx against C given E as well. But E varies in a nearly continuous way; there are 83 unique values out of total of 88 values. Clearly we cannot condition on single values.

Instead, we condition on intervals. This is done in figure 9.5. On each panel, NOx is graphed against C for E in an interval. The intervals, which are portrayed by the darkened bars in the strip, are ordered from low to high, so as we go left to right and bottom to top through the panels, the intervals go from low to high. The intervals overlap. The next section describes how they were created and the expression that produced the graph.

9.13 equal.count()

The nine intervals in figure 9.5 were produced by the *equal count algorithm*:

```
GIVEN.E <- equal.count(ethanol$E, number = 9,
  overlap = 1/4)
```

There are two inputs to the algorithm, the number of intervals and a target fraction of points to be shared by each pair of successive intervals. In figure 9.5, the inputs are 9 and 1/4. The algorithm picks interval endpoints that are values of the data; the left endpoint of the lowest interval is the minimum of the data, and the right endpoint of the highest interval is the maximum of the data. The endpoints are chosen to make the counts of points in the intervals as nearly equal as possible, and the fractions of points shared by successive intervals as close to the target fraction as possible.

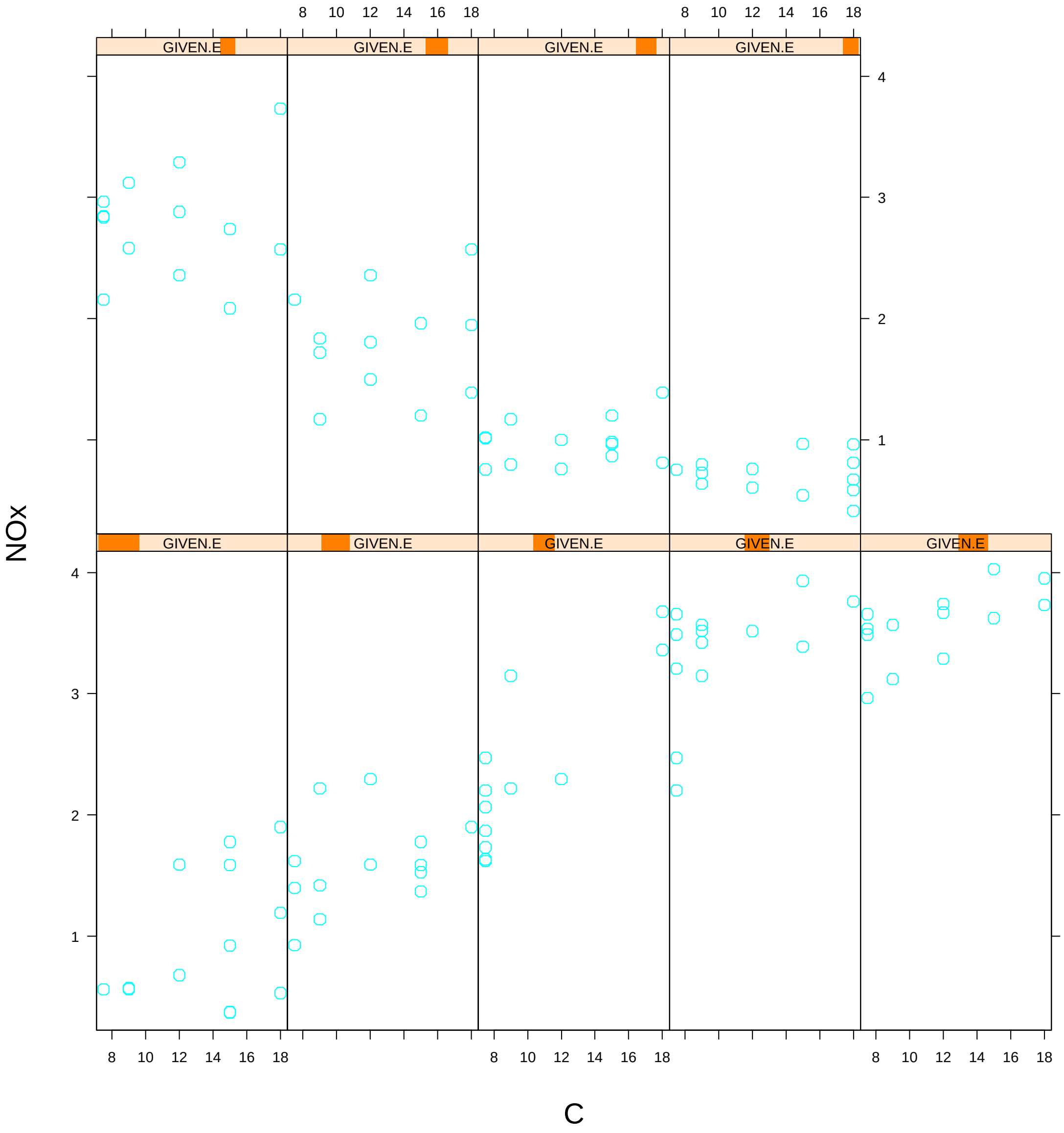


Figure 9.5

The command that produced figure 9.5 is

```
xyplot(NOx ~ C | GIVEN.E, data = ethanol, aspect = 2.5)
```

The aspect ratio was chosen to be 2.5 to approximately bank the underlying pattern of the points to 45°. Notice that the automatic layout algorithm chose five columns and two rows.

9.14 Shingles: shingle()

The result of `equal.count()` is an object of class `shingle`. The class is named "shingle" because of the overlap, like shingles on a roof. First, a shingle contains the numerical values of the variable and can be treated as an ordinary numeric variable:

```
> range(GIVEN.E)
[1] 0.535 1.232
> range(ethanol$E)
[1] 0.535 1.232
```

Second, a shingle has the intervals attached as an attribute. There is a plot method, a special Trellis function, that displays the intervals. Figure 9.6 shows the intervals of `GIVEN.E`:

```
plot(GIVEN.E)
```

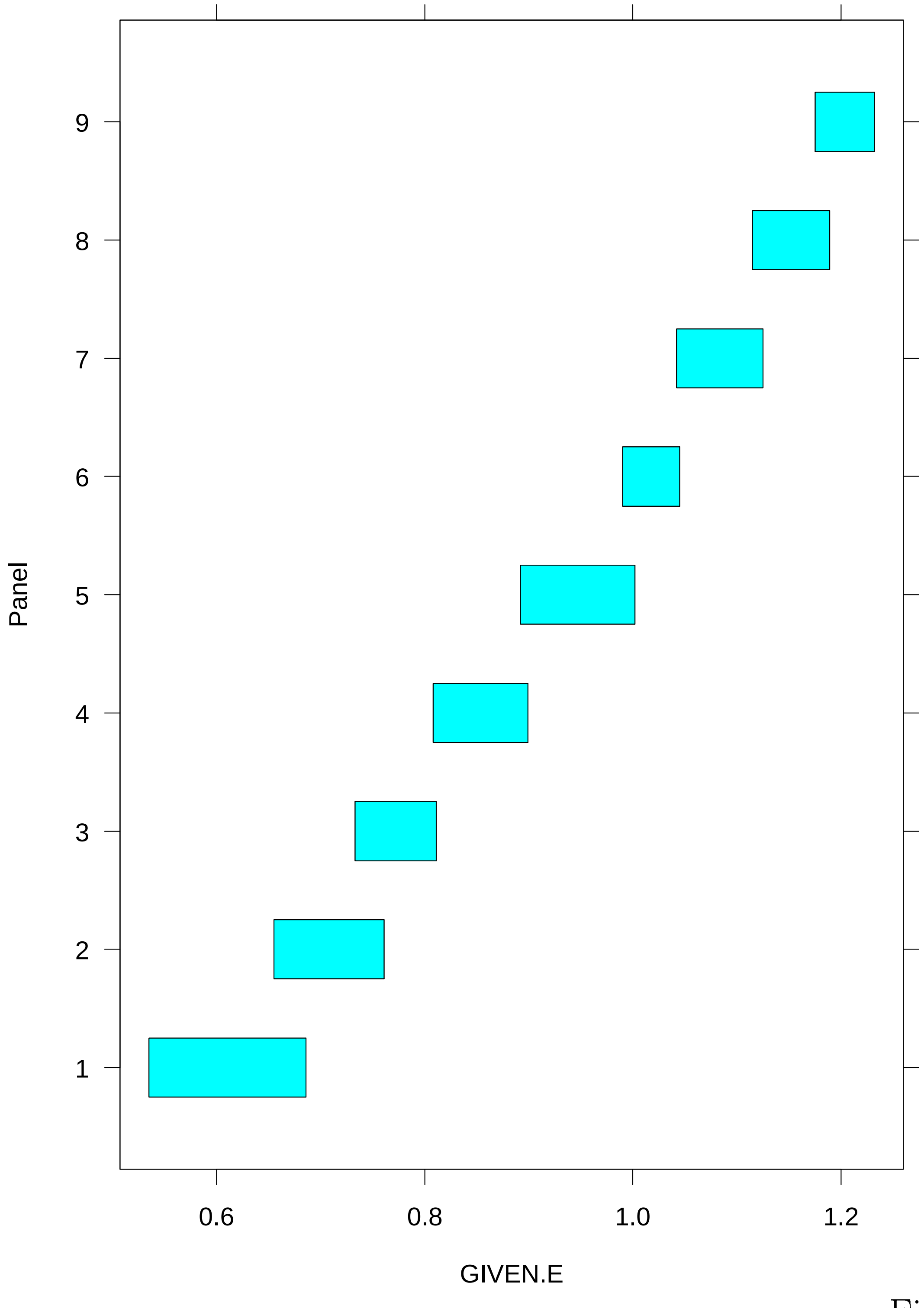


Figure 9.6

You can use the function levels() to extract the intervals from the shingle:

```
> levels(GIVEN.E)
   min   max
 0.535 0.686
 0.655 0.761
 0.733 0.811
 0.808 0.899
 0.892 1.002
 0.990 1.045
 1.042 1.125
 1.115 1.189
 1.175 1.232
```

A shingle can be specified directly by the function shingle(). For example, the following creates 5 intervals of equal width and no overlap for the variable ethanol$E:

```
> endpoints <- seq(min(ethanol$E), max(ethanol$E),
+ length = 6)
> GIVEN.E <- shingle(ethanol$E, intervals =
+     cbind(endpoints[-6],endpoints[-1]))
> levels(GIVEN.E)
    min    max
 0.5350 0.6744
 0.6744 0.8138
 0.8138 0.9532
 0.9532 1.0926
 1.0926 1.2320
```

The argument intervals= is a two-column matrix; the first column is the left endpoints of the intervals and the right column is the right endpoints of the intervals.

Chapter 10

Scales and Labels

The general display functions presented in chapter 7 have arguments that specify the scales and labels of graphs. These arguments are discussed in this chapter.

10.1 xlab=, ylab=, main=, sub=

Figure 10.1 is a scatterplot of NOx against E for the `gas` data, which were introduced in section 5.1:

```
xyplot(NOx ~ E, data = gas, aspect = 1/2)
```

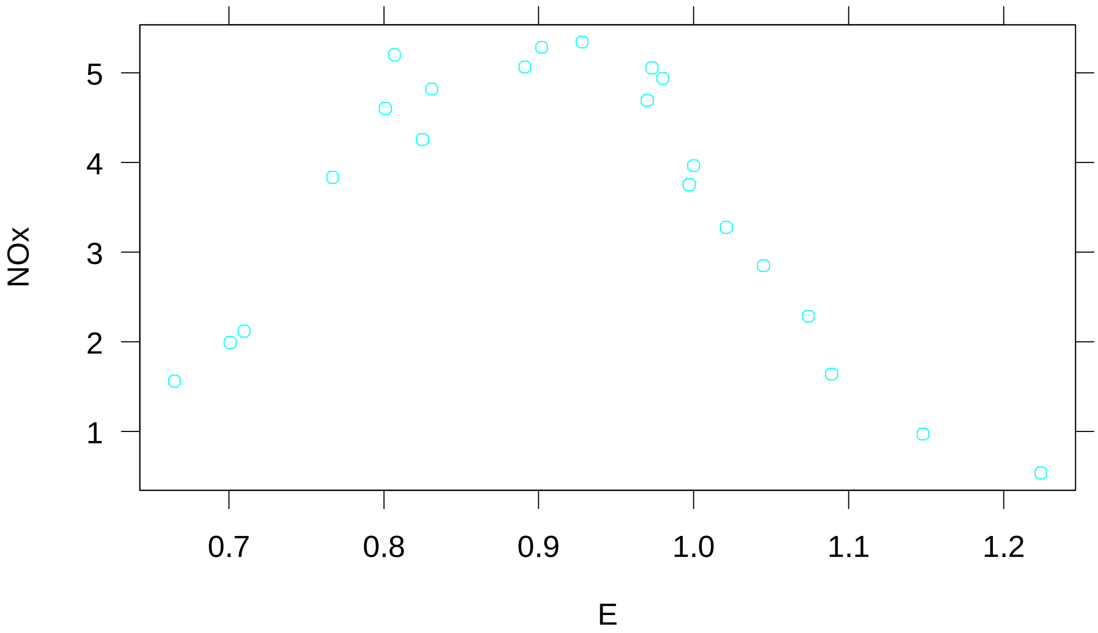


Figure 10.1

In figure 10.1, the label for the horizontal, or x, scale, and the label for the vertical, or y, scale are taken from the names used in the argument `formula=`. We can specify these scale labels as well as a main title at the top and a subtitle at the bottom. This is illustrated in figure 10.2:

```
xyplot(NOx ~ E, data = gas, aspect = 1/2,
  xlab = "Equivalence Ratio",
  ylab = "Oxides of Nitrogen",
  main = "Air Pollution",
  sub = "Single-Cylinder Engine")
```

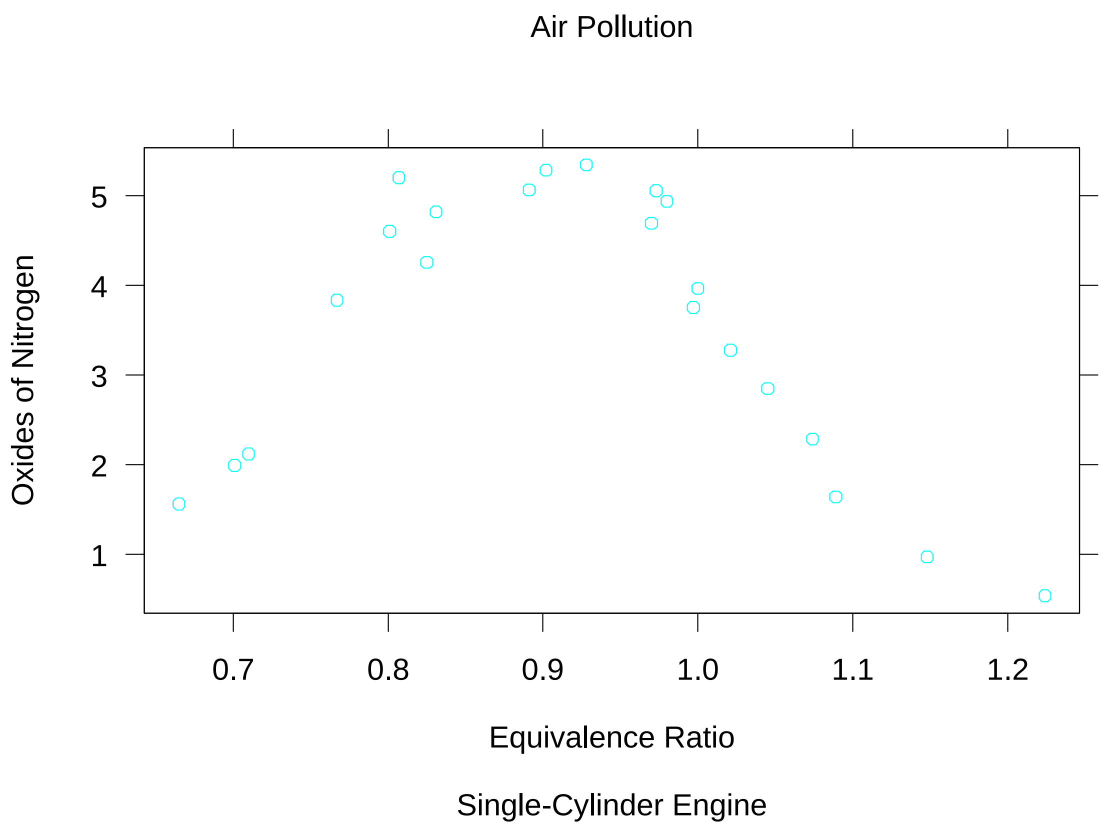

Air Pollution
Oxides of Nitrogen
1
2
3
4
5
0.7
0.8
0.9
1.0
1.1
1.2
Equivalence Ratio
Single-Cylinder Engine

Figure 10.2

Each of these four label arguments can also be a list. The first component of the list is a new character string for the text of the label. The other components specify the size, font, and color of the text. The component `cex` specifies the size; `font`, a positive integer, specifies the font; and `col`, a positive integer, specifies the color. Figure 10.3 changes the sizes of the title and subtitle:

```
xyplot(NOx ~ E, data = gas, aspect = 1/2,
  xlab = "Equivalence Ratio",
  ylab = "Oxides of Nitrogen",
  main = list("Air Pollution", cex = 2),
  sub = list("Single-Cylinder Engine", cex = 1.25))
```

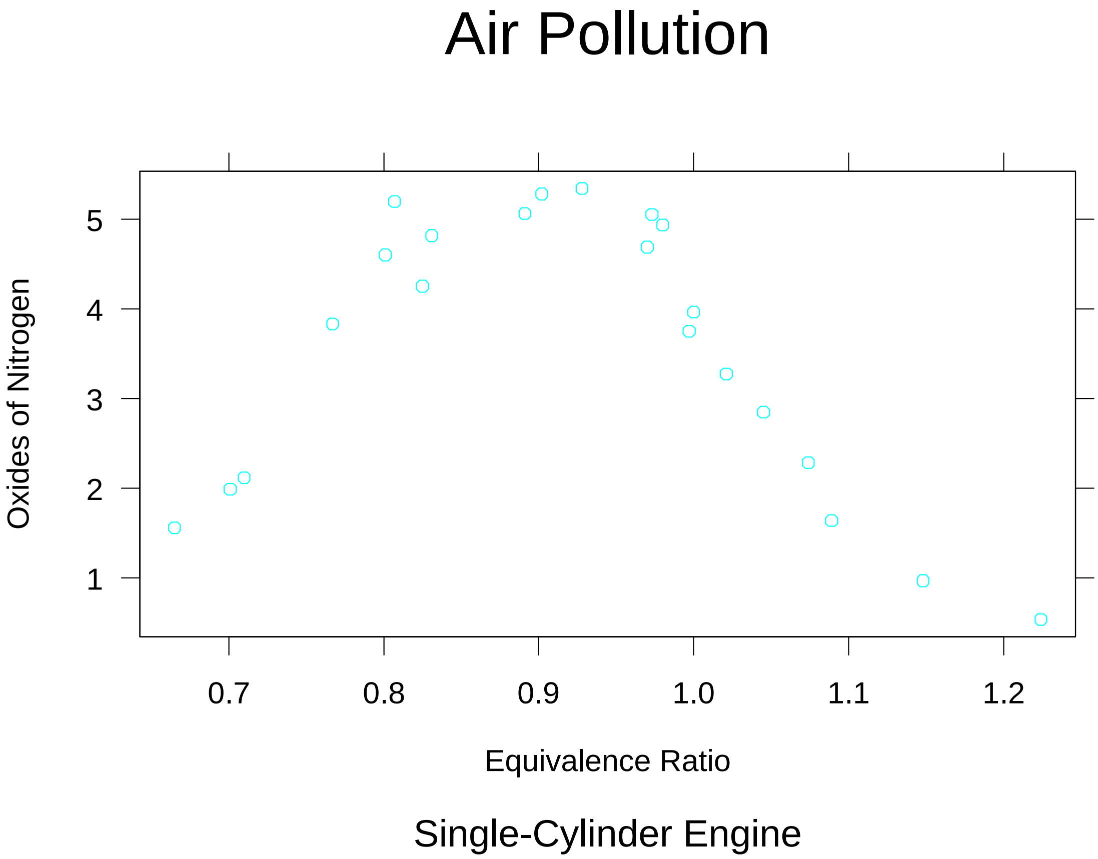

Air Pollution
Oxides of Nitrogen
1
2
3
4
5
0.7
0.8
0.9
1.0
1.1
1.2
Equivalence Ratio
Single-Cylinder Engine

Figure 10.3

10.2 xlim=, ylim=

In Trellis, the upper value of the scale line for a numeric variable is the maximum of the data to be plotted plus 4% of the range of the data. Similarly, the lower value of the scale line for a numeric variable is the minimum of the data to be plotted minus 4% of the range of the data. The 4% helps prevent the data values from running into the edge of the plot.

We can alter the extremes of the horizontal scale line by the argument `xlim=`, a vector of two values. The first value replaces the minimum of the data in the above procedure, and the second value replaces the maximum. Similarly, we can alter the vertical scale by `ylim=`.

In figures 10.1 to 10.3, NOx is graphed along the vertical scale. The limits of this variable are

```
> range(gas$NOx)
[1] 0.537 5.344
```

In figure 10.4, the values 0 and 6 have been included in the vertical scale:

```
xyplot(NOx ~ E, data = gas, aspect = 1/2, ylim = c(0, 6))
```

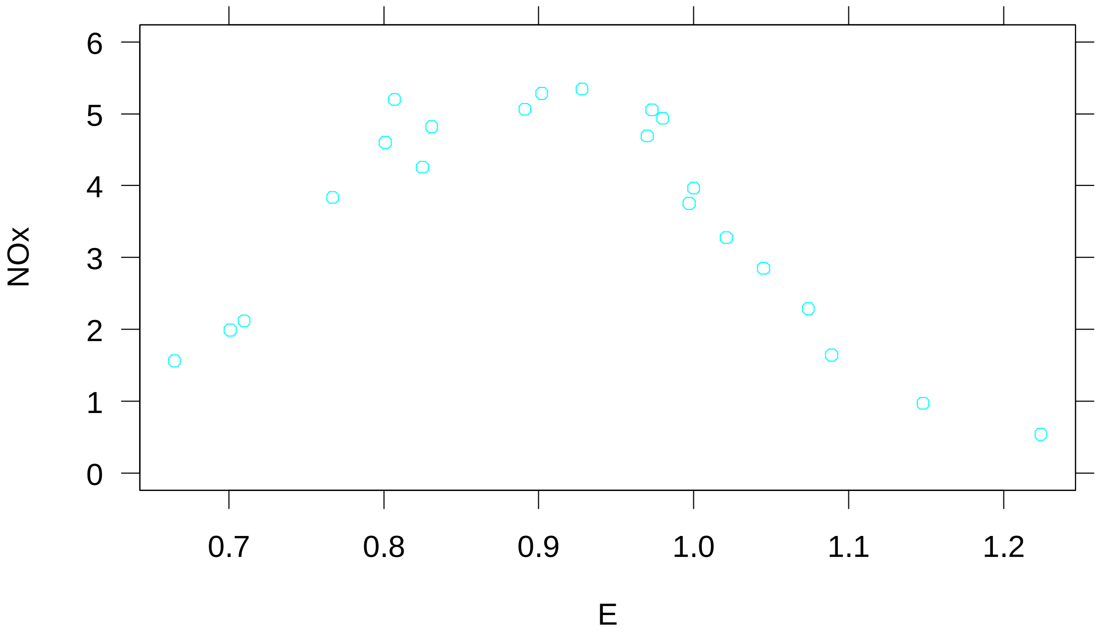


Figure 10.4

10.3 scales=, pscales=

The argument `scales=` affects tick marks and tick mark labels. In figure 10.4 there are seven tick marks and tick mark labels along the vertical scale and six along the horizontal. In figure 10.5, `scales=` is used to reduce the number of ticks and increase the size of the tick labels:

```
xyplot(NOx ~ E, data = gas, aspect = 1/2, ylim = c(0, 6),
  scales = list(cex = 2, tick.number = 4))
```

The argument `scales=` is a list. The list component `cex` affects the size. The list component `tick.number` affects the number, but it is just a suggestion; an algorithm goes off and tries to find tick values that are pretty, while trying to come as close as possible to the specified number.

We can also specify the tick marks and labels separately for each scale. The specification

```
scales = list(cex = 2,
  x = list(tick.number = 4),
  y = list(tick.number = 10))
```

changes `cex` on both scales, but `tick.number` has been set to 4 for the horizontal, or x, scale, and has been set to 10 for the vertical, or y, scale. Thus the rule is this: specifications for the horizontal scale appear in `scales=` as a component `x` that is itself a list, specifications for the vertical scale appear in `scales=` as a component `y` that is a list, and specifications for both scales appear as remaining components of `scales=`.

There is an exception to the behavior of `scales=`. The two 3-D general display functions `wireframe()` and `cloud()` currently do not accept changes to each scale separately; in other words, components `x` `y`, and `z` cannot be used. The general display function `piechart()` has no tick marks and labels, so `scales=` does not apply at all. The general display function `splom` has many scales, so the same delicate control is not available, but more limited control is available through the argument `pscales=`.

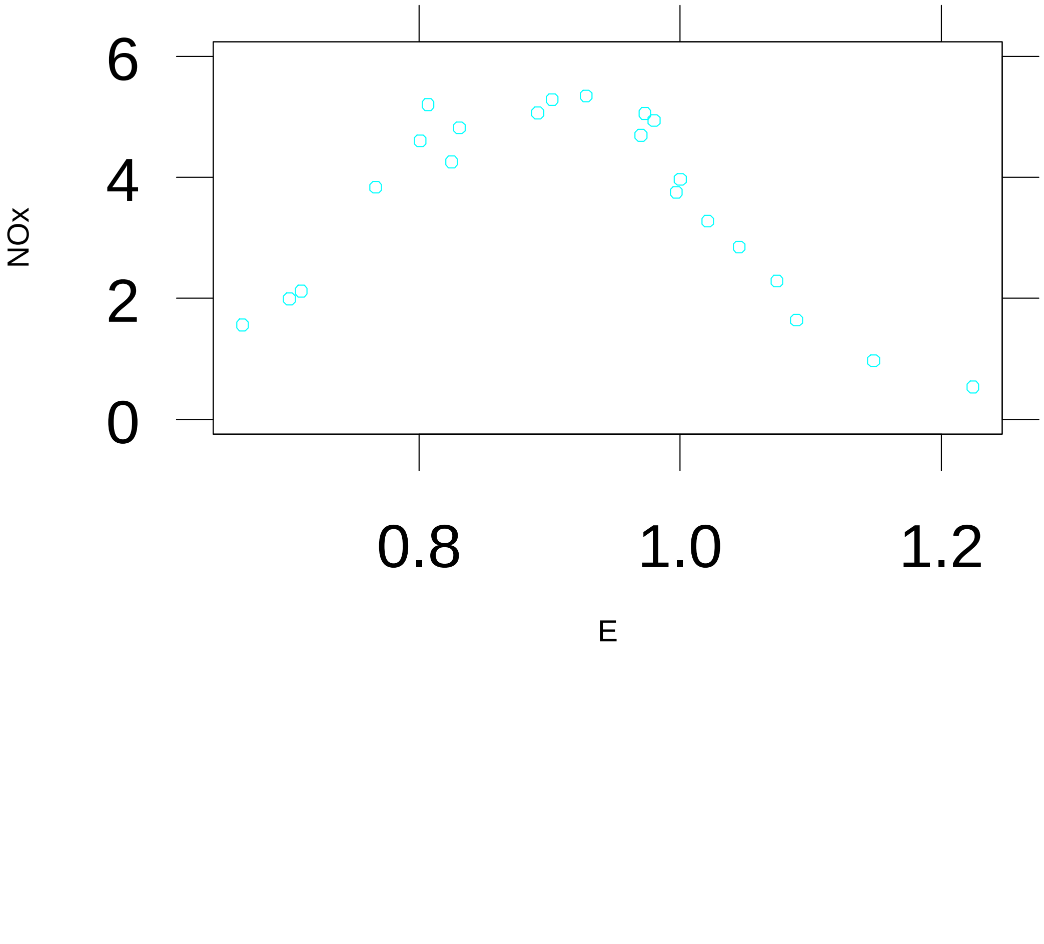


Figure 10.5

10.4 3-D Display: aspect=

The aspect ratio, the height of a panel data region divided by the width, is controlled by `aspect=`. This argument was introduced in chapter 6 for 2-D displays. The behavior of `aspect=` for the two 3-D general display functions, `wireframe()` and `cloud()`, is somewhat different. Since there are three axes, we must specify two aspect ratios to specify the shape of the 3-D box around the data. Suppose the formula and the aspect arguments are

```
formula = z ~ x * y, aspect = c(1, 2)
```

Then the ratio of the length of the y-axis to the length of the x-axis is 1, and the ratio of the length of the z-axis to the length of the x-axis is 2.

10.5 Changing the Text in Strip Labels

The default text in the strip label for a numeric conditioning variable is the name of the variable. This is illustrated in figure 10.6, which displays the ethanol data introduced in section 9.10:

```
xyplot(NOx ~ E | C, data = ethanol)
```

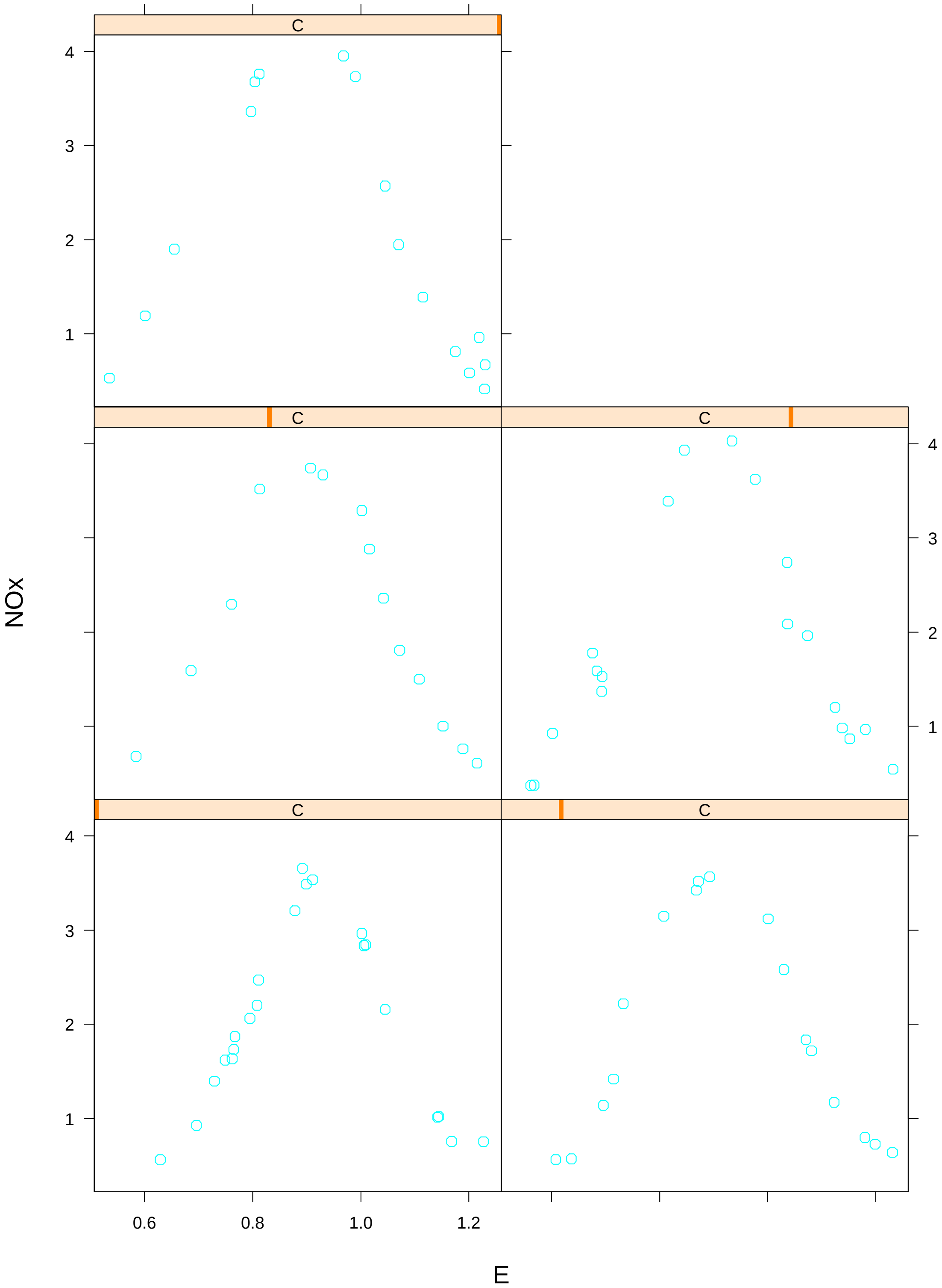


Figure 10.6

The default text in the strip label of a factor conditioning variable is the name of the factor level for the panel. This is illustrated in figure 10.7, which displays the barley data introduced in section 9.1.

```
dotplot(variety ~ yield | year * site, data = barley)
```

The name of the factor, for example, `site`, does not appear because seeing the names of the levels is typically enough to convey the name of the factor.

Thus the text comes from the names given to variables and factor levels in the datasets that are plotted. If we want to change the text we can change the names. For example, if we want to change the long label "University Farm" to "U. Farm" then we can change the names of the levels of the factor `site`:

```
> levels(barley$site)
[1] "Grand Rapids"    "Duluth"          "University Farm"
[4] "Morris"          "Crookston"       "Waseca"

> levels(barley$site)[3] <- "U. Farm"
```

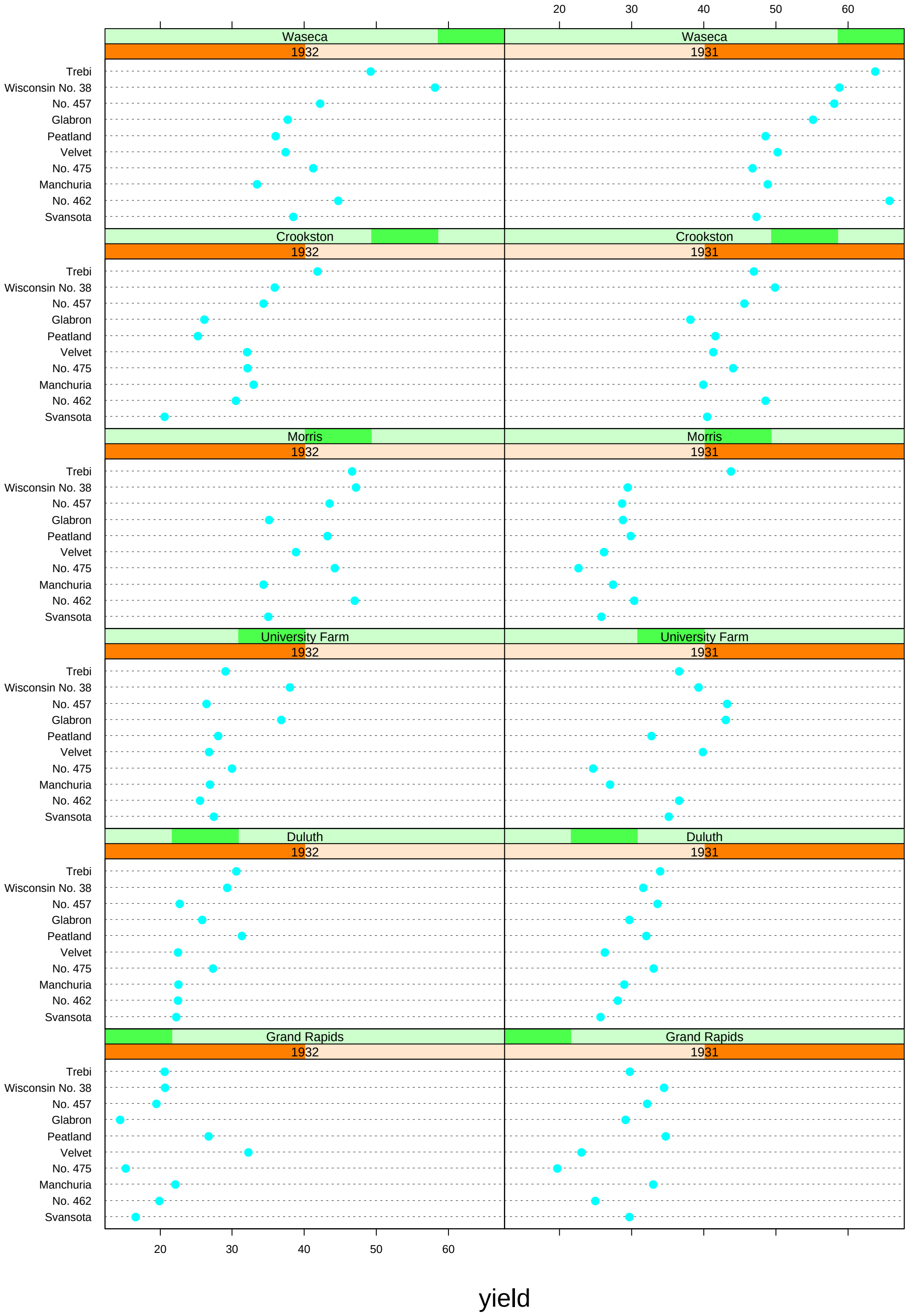

Waseca
1932
1931
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
yield

Figure 10.7

10.6 Strip Label Text Size: par.strip.text=

The size, font, and color of the text in the strip labels can by changed by the argument `par.strip.text=`, a list whose components are the parameters `cex` for size, `font` for the font, and `col` for the color. For example, we can make huge strip labels by

```
par.strip.text = list(cex = 2)
```

10.7 Programming Strip Labels: strip=

The argument `strip=` allows very delicate control of what is put in the strip labels. One usage is to remove the strip labels altogether:

```
strip = F
```

Another is to control the inclusion of names of conditioning variables in strip labels. This is illustrated in Figure 10.8:

```
dotplot(variety ~ yield | year * site, data = barley,
  strip = function(...)
    strip.default(..., strip.names = c(T,T))
  )
```

The argument `strip.names=` takes a logical vector of length two. The first element tells whether or not the names of factors should be included along with the names of the levels of the factor, and the second element tells whether or not the names of shingles should be included. The default is `c(F,T)`.

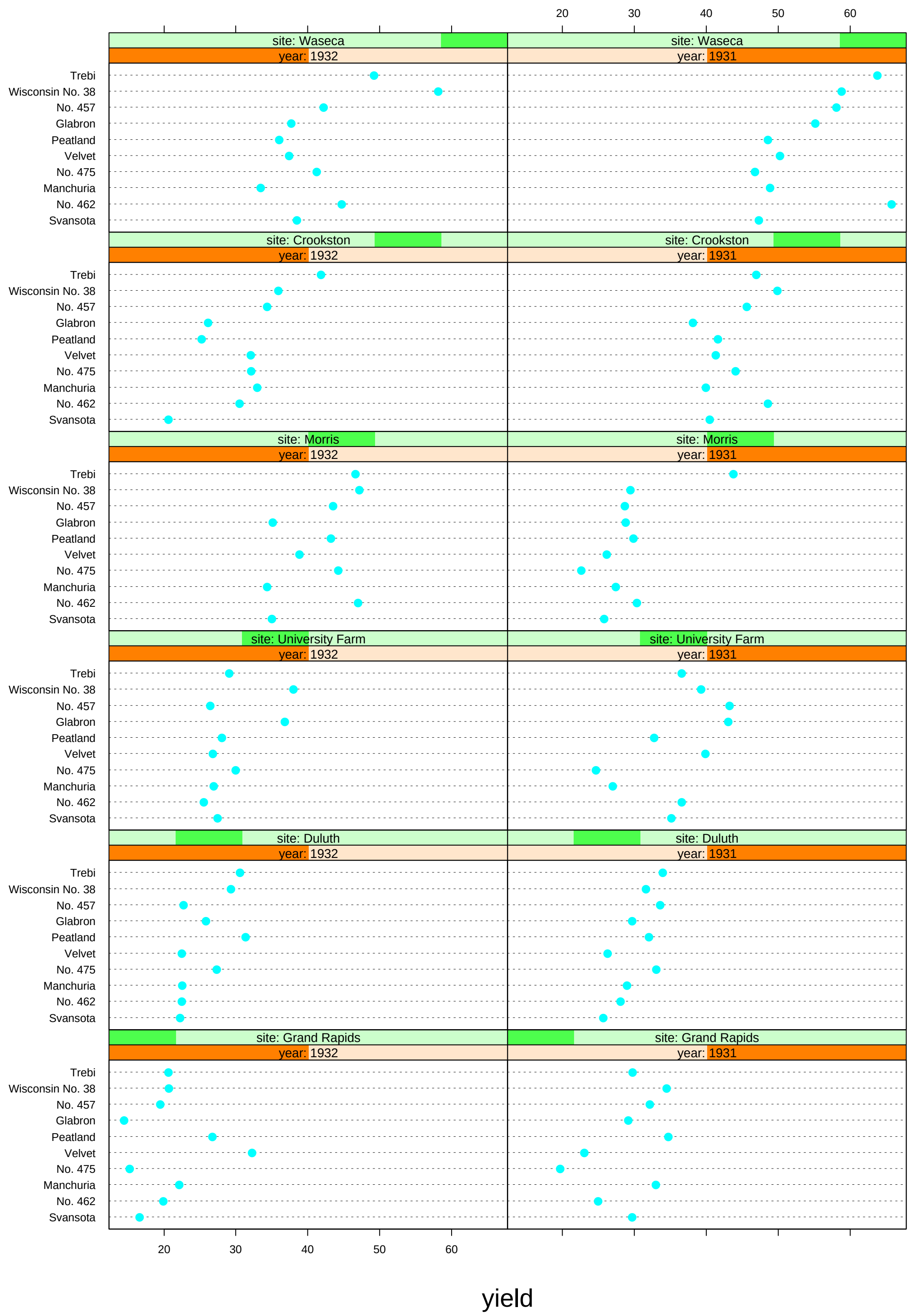

site: Waseca
year: 1932
year: 1931
site: Crookston
site: Morris
site: University Farm
site: Duluth
site: Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
yield

Figure 10.8

Chapter 11

Devices

11.1 Three Kick Methods

You can send Trellis graphs to a printer directly or to a file for later printing. But when you issue a command to do this, the sending does not happen immediately. You need to give the system a kick. There are three ways to kick: (1) send another graph; (2) turn off the device with the command `dev.off()`; (3) end your S-PLUS session with `q()`.

11.2 trellis.device()

The function `trellis.device()` specifies a device and enables Trellis Graphics to tailor rendering details such as color, symbols, and line types to the specified device. We saw in section 3.1 that it can be used to specify screen devices. As we will see, it can be used to specify devices for sending directly to a printer or for sending to a file for later printing.

11.3 Sending to a Printer or a File

On UNIX, the command

```
trellis.device(postscript, onefile = FALSE)
```

sets up a PostScript device for direct sending to the printer. A graph goes to the printer when you kick the system. Adding `color = TRUE` to the argument list specifies color PostScript.

On UNIX, the command

```
trellis.device(postscript,
  onefile = FALSE,
  print.it = FALSE,
  file = "greatgraph.ps")
```

sets up a PostScript device for sending to the file greatgraph.ps. The file writing is completed after you kick. But with this device specification, if you issue two commands to draw two separate graphs, the first will overwrite the second. Again, adding `color = TRUE` to the argument list specifies color PostScript.

On Windows, you can specify various types of printers. The command

```
trellis.device(win.printer, printer.type = "postscript")
```

specifies a PostScript printer for direct sending. A graph goes to the printer when you kick the system. Adding `color = TRUE` to the argument list specifies color PostScript. For PCL printers use:

```
trellis.device(win.printer, printer.type = "pcl")
```

However, while you can get color printing on PCL printers by changing arguments to Trellis functions, there is not yet an argument `color` to customize PCL for color printing.

On Windows, the command

```
trellis.device(win.printer,
  printer.type = "postscript",
  format = "printer",
  file = "graph.ps")
```

writes PostScript to the file graph.ps, after the kick. Similarly,

```
trellis.device(win.printer,
  printer.type = "pcl",
  format="printer",
  file = "graph.pcl")
```

does the same for PCL. Note that if you issue two commands to draw two separate graphs without changing the device in any way, the first will overwrite the second.

You can also create a Windows metafile that can be inserted into documents:

```
trellis.device(win.printer,
  format = "placeable metafile",
  file = "graph.wmf")
```

On Windows, you can print hardcopies by using the S-PLUS File–Print menu, but this typically produces an undesirable graph because Trellis Graphics cannot customize the rendering to your hard copy device.

11.4 Devices for this *Manual*

The graphs for this *Manual* were produced on UNIX using the `postscript` device. The device used for the black and white graphs was

```
trellis.device(postscript)
```

and the device used for color graphs was

```
trellis.device(postscript, color = T)
```

11.5 Multiple Devices: dev.list(), dev.cur(), dev.set()

S-PLUS allows you to run multiple devices. A common usage is to have a screen device and a hardcopy device, the first for experimenting and the second for sending what you hope will be a finished product.

Suppose you are on UNIX. Then

```
trellis.device(motif)
trellis.device(postscript)
```

sets up a screen and a hardcopy device. Only one device is current, and that one receives your graphics commands. For our example, `postscript` is current since it was set up last. You can change the current device:

```
> dev.set(which = 2)
 motif
     2
```

Now `motif` is current. You can show the current device:

```
> dev.cur()
  motif
     2
```

You can see the list of all active devices:

```
> dev.list()
 motif postscript
     2          3
```

Finally, as we have seen, `dev.off()` turns off the current device and shows the new current device:

```
> dev.off()
 postscript
          3
```

On Windows, you can use these functions, but you can also use the Tools-Graphics Device menu to list, select, and close graphics devices, including Trellis devices. (You cannot open a Trellis device from this menu, but you can manipulate it once it is open.)

Chapter 12

Panel Functions

The data region of a panel on a Trellis display is the rectangular region where the data are plotted. A *panel function* has the sole responsibility for drawing in the data regions produced by a general display function. The panel function is given as an argument of the general display function. The other arguments of the general display function manage the superstructure of the graph—scales, labels, boxes around the data region, and keys. The panel function manages the symbols, lines and so forth that encode the data in the data region.

Every general display function has a default panel function. In all examples given so far in this *Manual*, the default panel function has been doing the drawing.

12.1 How to Change the Rendering in the Data Region

You can change what is drawn in the data region by one of two mechanisms. First, a default panel function has arguments. You can change the rendering by using these arguments; in fact, you can give them to the general display function, which will pass them along to the panel function. Second, you can write your own panel function.

12.2 Passing Arguments to a Default Panel Function

The name of the default panel function for a general display function is "`panel.`" followed by the name of the general function. For example the default panel function for `xyplot()` is `panel.xyplot()`. You can use S-PLUS online help to see the arguments of a default panel function. For example, `?panel.xyplot` tells you about the panel function for `xyplot`.

You can give an argument to a panel function by giving it to the general display function; the general display function passes it on to the panel function. In Figure 12.1, `xyplot()` passed `pch` to `panel.xyplot` to specify a "+" as the plotting symbol:

```
xyplot(NOx ~ E, data = gas, aspect = 1/2, pch = "+")
```

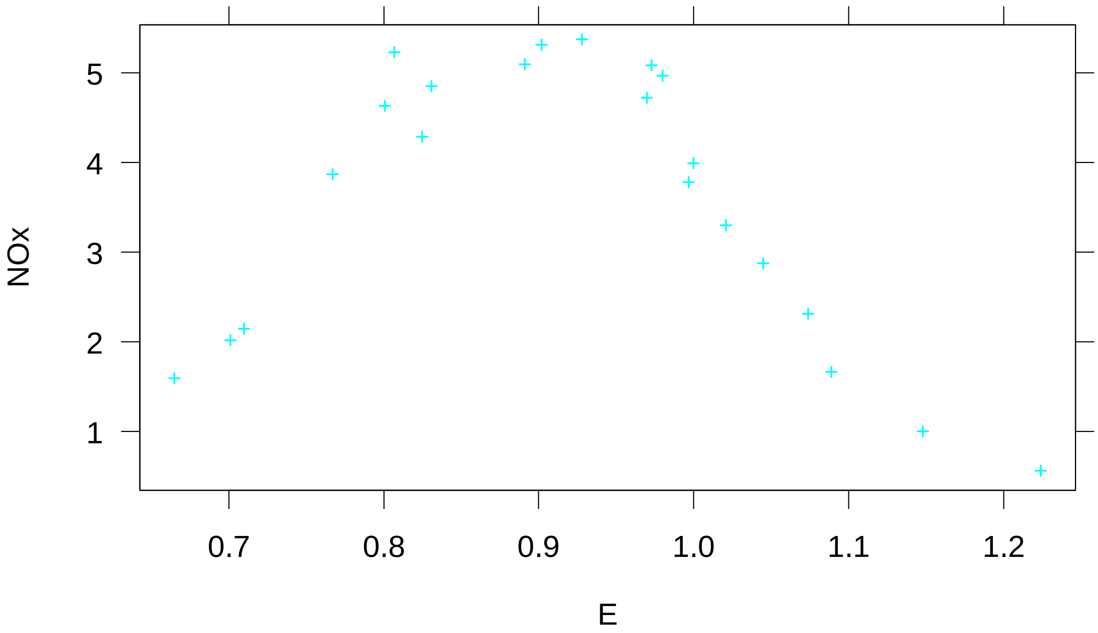


Figure 12.1

12.3 Writing A Panel Function: panel=

If you write your own panel function, you give it to the general display function as the argument `panel=`. For example, if you have your own panel function `mypanel()`, you specify

```
panel = mypanel
```

A panel function is always a function of at least two arguments; the first two are named `x` and `y`. Suppose, for the gas data, that you want to use `xyplot()` to graph NOx against E and use a "+" as the plotting symbol for all observations except that for which NOx is a maximum, in which case you want to use "M". There is no provision for `xyplot()` to do this, so you must write your own.

First, let us write the panel function:

```
panel.special <- function(x,y){
  biggest <- y == max(y)
  points(x[!biggest], y[!biggest], pch = "+")
  points(x[biggest], y[biggest], pch = "M")
}
```

The function `points()` is a core graphics function. It graphs individual points on a graph. Its first argument `x` contains the coordinates of the points along the horizontal scale, and its second argument `y` contains the coordinates of the points along the vertical scale. The third argument `pch` gives the symbol used to display the points.

Figure 12.2 shows the result of giving `panel.special()` to `xyplot()`.

```
xyplot(NOx ~ E, data = gas, aspect = 1/2,
  panel = panel.special)
```

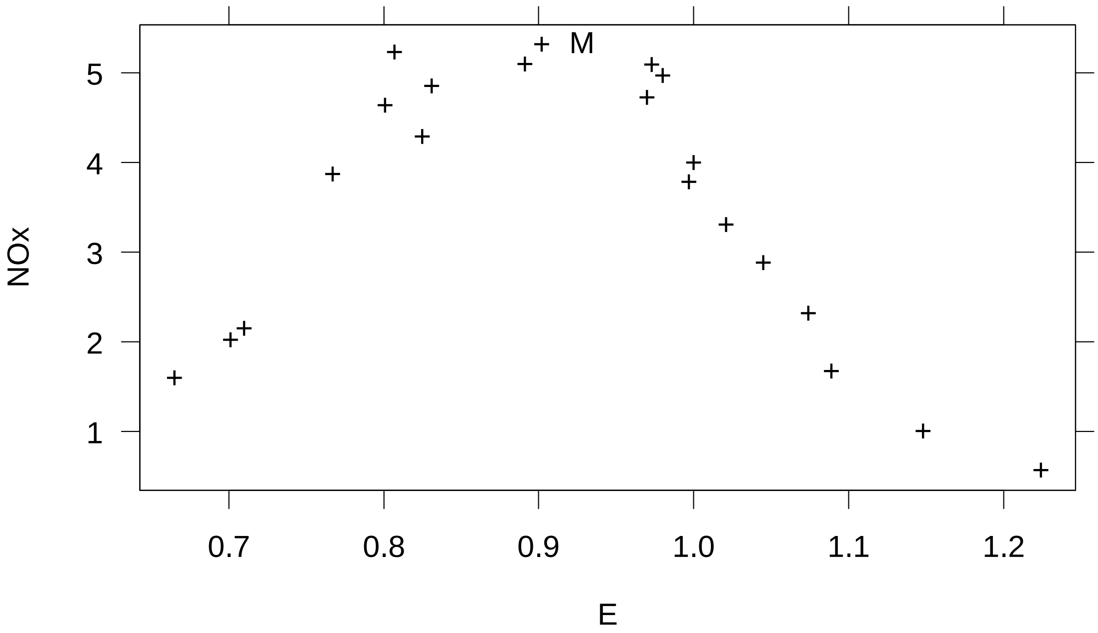


Figure 12.2

The panel function for figure 12.2 also could have been defined as part of the `xyplot()` command:

```
xyplot(NOx ~ E, data = gas,
  aspect = 1/2,
  panel = function(x,y){
    biggest <- y == max(y)
    points(x[!biggest], y[!biggest], pch = "+")
    points(x[biggest],  y[biggest],  pch = "M")
    }
  )
```

12.4 A Panel Function for a Multipanel Display

In most cases, a panel function that is used for a single panel display can be used for a multipanel display as well. In figure 12.3 the panel function `panel.special()`, just used in figure 12.2, is used to show the maximum value of NOx on each panel of a multipanel display of the ethanol data:

```
xyplot(NOx ~ E | C, data = ethanol, aspect = 1/2,
  panel = panel.special)
```

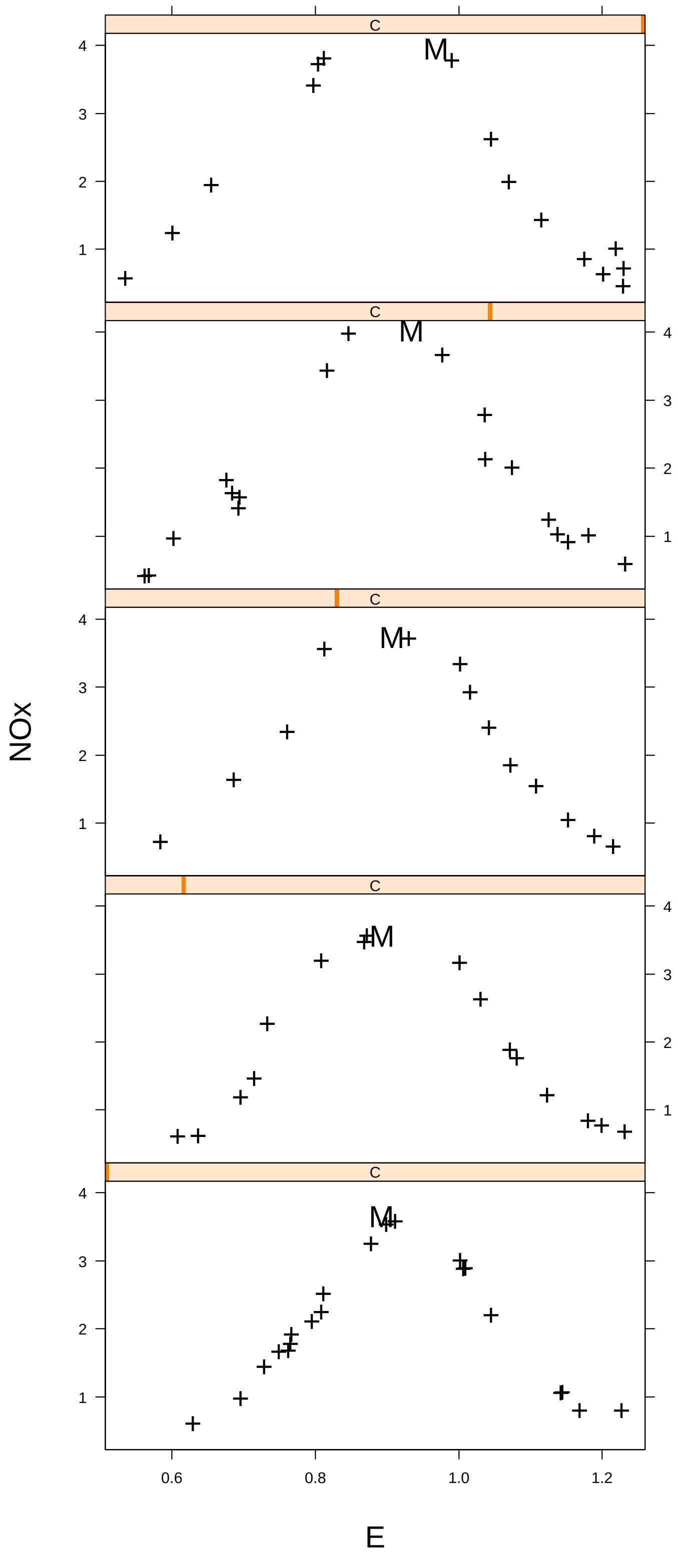


Figure 12.3

12.5 Special Panel Functions

Even if you write your own panel function you might want to use the default panel function as part of it. This is often true when you want to augment a standard Trellis panel. Also, Trellis Graphics provides some special purpose panel functions. One of them is `panel.loess()`. It adds smooth curves to scatterplots.

Figure 12.4 adds smooth curves to a multipanel display of the ethanol data:

```
xyplot(NOx ~ C | GIVEN.E, data = ethanol, aspect = 2.5,
  panel = function(x, y){
    panel.xyplot(x, y)
    panel.loess(x, y, span = 1)
    }
  )
```

The default panel function `panel.xyplot()` draws the points of the scatterplot on each panel. The special panel function `panel.loess()` computes and draws the smooth curves; the argument `span`, the smoothing parameter, has been specified.

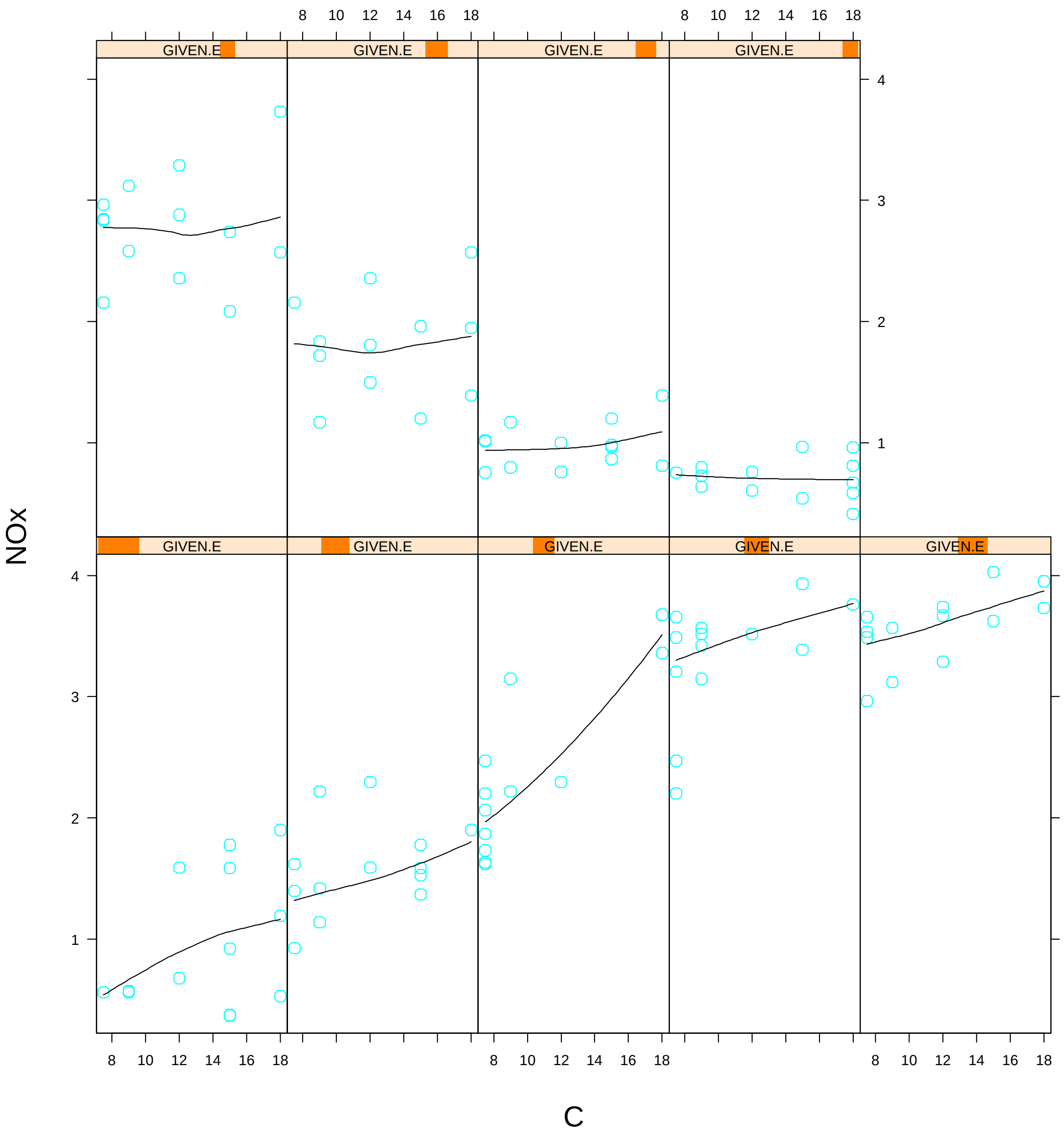


Figure 12.4

12.6 subscripts=

If you request it, another component of the packet sent to each panel is the subscripts that tell which original observations make up the the packet. Knowing these subscripts is helpful for getting the values of other variables that might be needed for rendering on the panel. In such a case the panel function argument `subscripts=` contains the subscripts. In figure 12.5 the observation numbers have been added to the graph of NOx against E given C:

```
xyplot(NOx ~ E | C, data = ethanol, aspect = 1/2,
  panel = function(x, y, subscripts)
    text(x, y, subscripts, cex = .75)
)
```

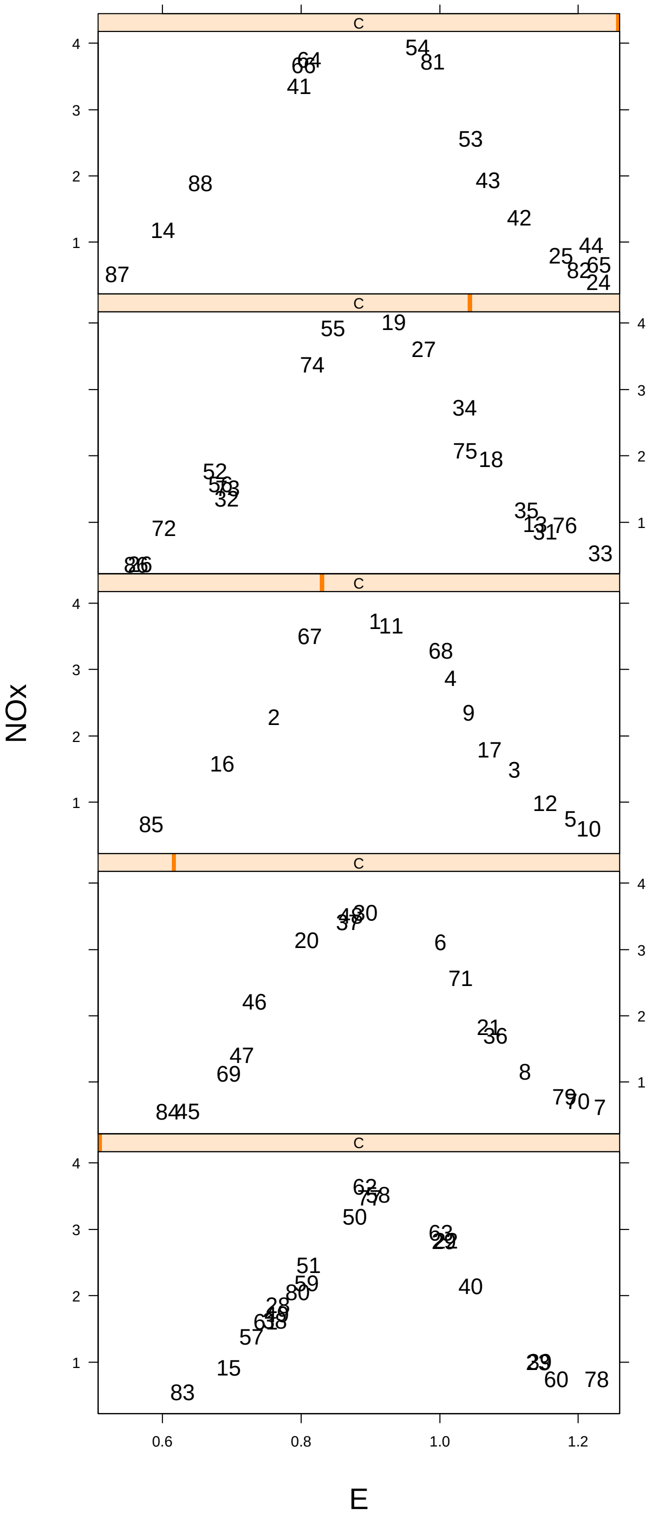


Figure 12.5

12.7 Commonly-Used Core S-PLUS Graphics Functions and Parameters

The core graphics functions commonly used in writing panel functions are

```
points()
lines()
text()
segments()
polygon()
```

You can use the S-PLUS online help to see what they do. The core parameters commonly used in writing panel functions are

```
col
lty
pch
lwd
cex
```

Use `?par` for their definitions.

Chapter 13

Panel Functions and the Trellis Settings

Trellis Graphics, as we have discussed, is implemented using S-PLUS core graphics, which has controllable graphical parameters that determine the characteristics of plotted objects. For example, if we want to use a symbol to show points on a scatterplot, graphical parameters determine the type, size, font, and color of the symbol.

In Trellis Graphics, the default panel functions for the general display functions select graphical parameters to render plotted elements as effectively as possible. But because the most desirable choices for one graphics device can be different from those for another device, the default graphical parameters are device dependent. These parameters are contained in lists which we will refer to as the "Trellis settings." When `trellis.device()` sets up a graphics device, the Trellis settings are established for that device and are saved on a special data structure.

When you write your own panel functions, you may want to make use of the Trellis settings to provide good performance across different devices.

Three functions enable you to access, display, and change the settings for the current device. `trellis.par.get()` lets you get settings for use in a panel function. `show.settings()` shows graphically the values of the settings. `trellis.par.set()` lets you change the settings for the current device.

13.1 trellis.par.get()

Here is the panel function `panel.xyplot()`:

```
function(x, y, type = "p", cex = plot.symbol$cex,
  pch = plot.symbol$pch, font = plot.symbol$font,
  lwd = plot.line$lwd, lty = plot.line$lty,
  col = if(type == "l") plot.line$col
        else plot.symbol$col, ...)
{
if(type == "l") {
  plot.line <- trellis.par.get("plot.line")
  lines(x, y, lwd = lwd, lty = lty, col = col,
    type = type, ...) }
else {
  plot.symbol <- trellis.par.get( "plot.symbol")
  points(x, y, pch = pch, font = font, cex = cex,
    col = col, type = type, ...) }
}
```

If the argument `type` is `"p"`, which means that point symbols are used to plot the data, then the plotting symbol is defined by the settings list `plot.symbol`; the components of this last are given to the function `points()` that draws the symbols. The list is accessed by `trellis.par.get()`.

Here is the list `plot.symbol` for the color PostScript device:

```
> trellis.device(postscript, color = T)
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol
$pch:
[1] 1

$col:
[1] 2

$cex:
[1] 0.8

$font:
[1] 1
```

The `pch` of 1 and `col` of 2 produce a cyan octagon.

If `type` is `"l"`, which means that `lines()` is used to plot the data, then the graphical parameters for the lines are in the settings list `plot.line`:

```
> trellis.device(postscript, color = T)
> plot.line <- trellis.par.get("plot.line")
> plot.line
$lwd:
[1] 1

$lty:
[1] 1

$col:
[1] 2
```

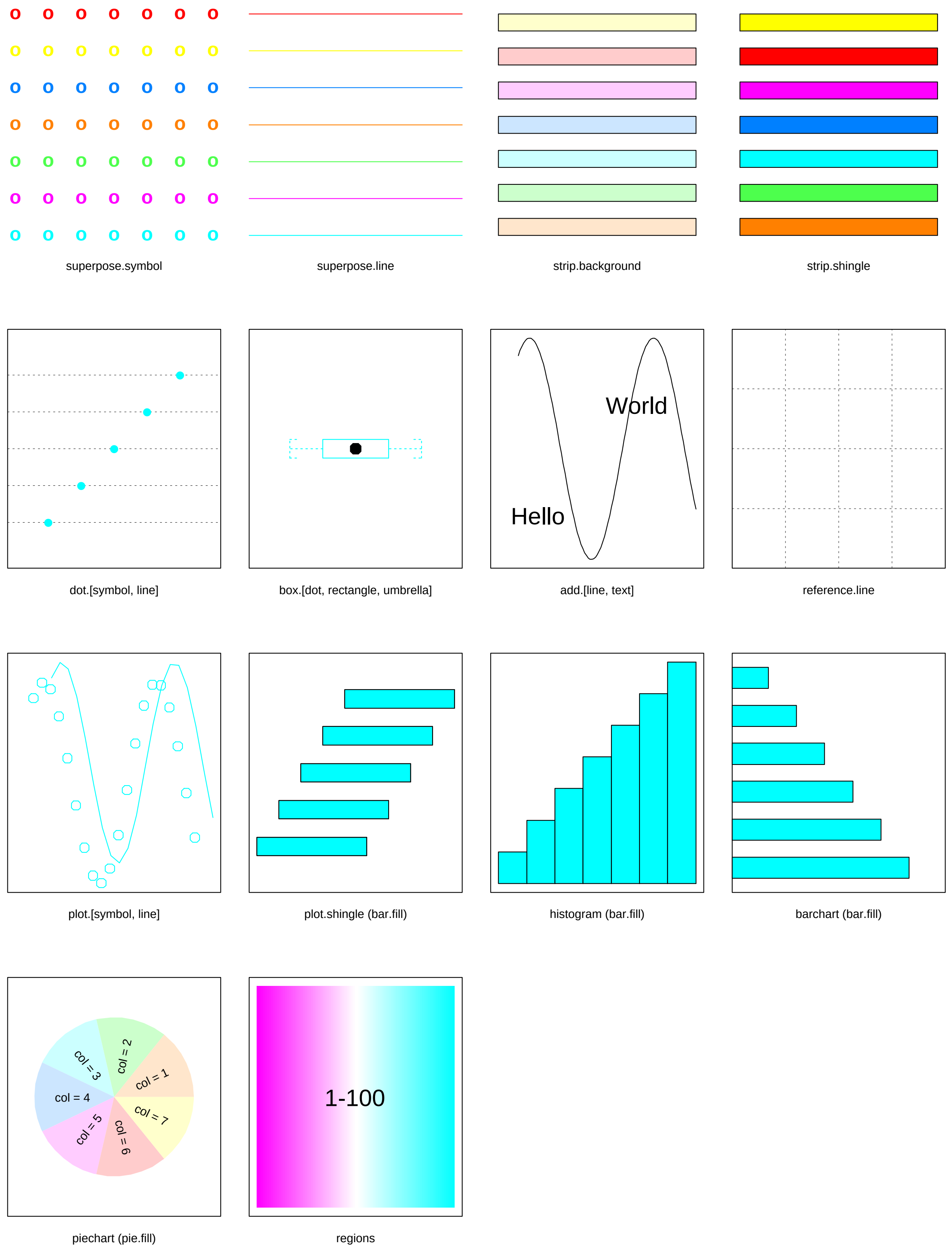
This is a cyan-colored solid line.

13.2 show.settings()

`show.settings()` displays the graphical parameters in the Trellis settings for the current device. The result for color PostScript is shown in Figure 13.1:

```
trellis.device(postscript)
show.settings()
```

Each panel displays one or more settings lists. The names of the settings appear below the panels. For example, the panel in the third row (from the top) and first column shows plotting symbols with graphical parameters `plot.symbol` and lines with graphical parameters `plot.line`, and the panel in the third row and third column shows that the panel function of the general display function `histogram()` uses the graphical parameters in `bar.fill` for the color that shades the bars of a histogram.



superpose.symbol
superpose.line
strip.background
strip.shingle
dot.[symbol, line]
box.[dot, rectangle, umbrella]
World
Hello
add.[line, text]
reference.line
plot.[symbol, line]
plot.shingle (bar.fill)
histogram (bar.fill)
barchart (bar.fill)
col = 1
col = 2
col = 3
col = 4
col = 5
col = 6
col = 7
piechart (pie.fill)
1-100
regions

Figure 13.1

13.3 trellis.par.set()

The Trellis settings for the current device can be changed:

```
> trellis.device(postscript, color = T)
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol$col
[1] 2
> plot.symbol$col <- 3
> trellis.par.set("plot.symbol", plot.symbol)
> plot.symbol <- trellis.par.get("plot.symbol")
> plot.symbol$col
[1] 3
```

`trellis.par.set()` sets an entire Trellis setting list, not just some of the components. Thus the simplest way to make a change is to get the current list, alter it, and then save the altered list. The change lasts only as long as the device continues. If the S-PLUS session is ended the altered settings are removed.

Chapter 14

Superposing Two or More Groups of Values on a Panel

One common visualization task is superposing two or more groups of values in the same data region, encoding the different groups in different ways to show the grouping. For example, we might graph leaf width against leaf length for two samples of leaves, one from maple trees and one from oaks, and use a circle as the plotting symbol for the maples and a plus for the oaks.

Superposition is achieved by the panel function `panel.superpose()`. In addition, the `key=` argument of the general display functions can be used to show the group encoding.

14.1 panel.superpose()

Superposition is illustrated in Figure 14.1 which graphs variables from the data frame `fuel.frame`. For 60 automobiles, `Mileage` is graphed against `Weight` for six types of vehicles described by the factor `Type`:

```
> table(fuel.frame$Type)
 Compact Large Medium Small Sporty Van
      15     3     13    13      9   7
```

The vehicle types are encoded by using different plotting symbols. (Nothing on the graph indicates which symbol is for which type, but the next section contains information about drawing a legend, or key.)

The panel function `panel.superpose()` carries out such a superposition, and was used to create Figure 14.1:

```
xyplot(Mileage ~ Weight,
  data = fuel.frame,
  groups = Type,
  aspect = 1,
  panel = panel.superpose)
```

The factor `Type` is given to the argument `groups` of `xyplot()`. But `groups` is also an argument of `panel.superpose()`, so `Type` is passed along to the panel function to be used to determine the plotting symbols.

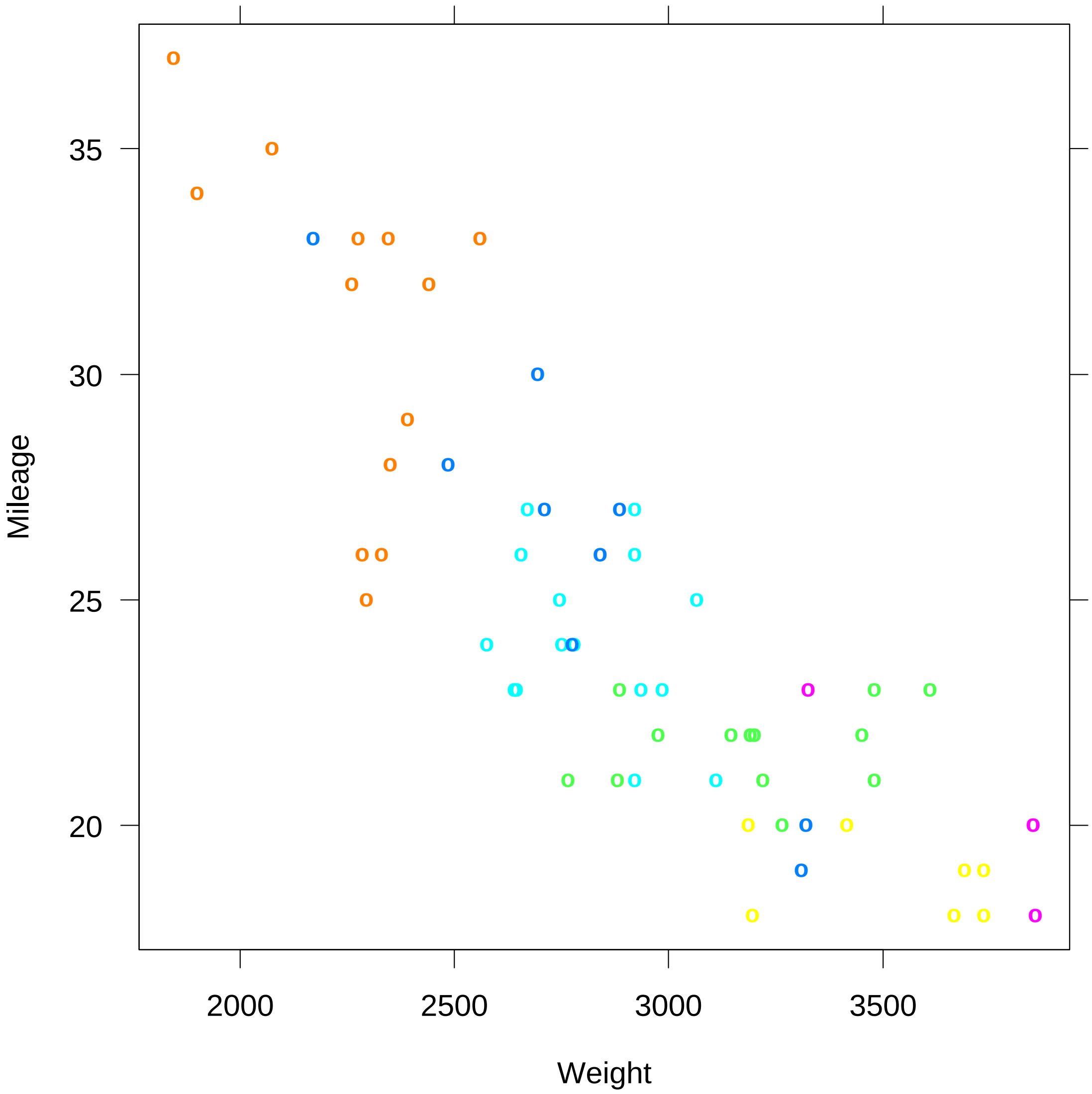


Figure 14.1

In Figure 14.1, the plotting symbols are the defaults that are set up by the trellis device function `trellis.device()`; such trellis settings were discussed in chapter 13. The specific settings used by `panel.superpose()` are discussed later in this section. The default symbols have been chosen to enhance the visual assembly of each group of points; that is, we want to effortlessly assemble the plotting symbols of a given type to form a visual gestalt or whole. If assembly can be performed efficiently then we can compare the characteristics of the data for different automobile types.

You can choose your own plotting symbols. For example, suppose that in Figure 14.1 we want to use the first letters of the vehicle types, but with "S" (for "Small") replaced by "P" (for "Peewee") to avoid duplication with "Sporty":

```
mysymbols <- c("C","L","M","P","S","V")
```

`panel.superpose()` has an argument `pch=` that can be used to specify the symbols. This is shown in Figure 14.2, which results from the expression:

```
xyplot(Mileage ~ Weight,
  data = fuel.frame,
  aspect = 1,
  groups = Type,
  pch = mysymbols,
  panel = panel.superpose
  )
```

Notice that, again, we specify an argument of the panel function — in this case `pch` — by giving it as an argument to `xyplot()`, which passes it along to the panel function.

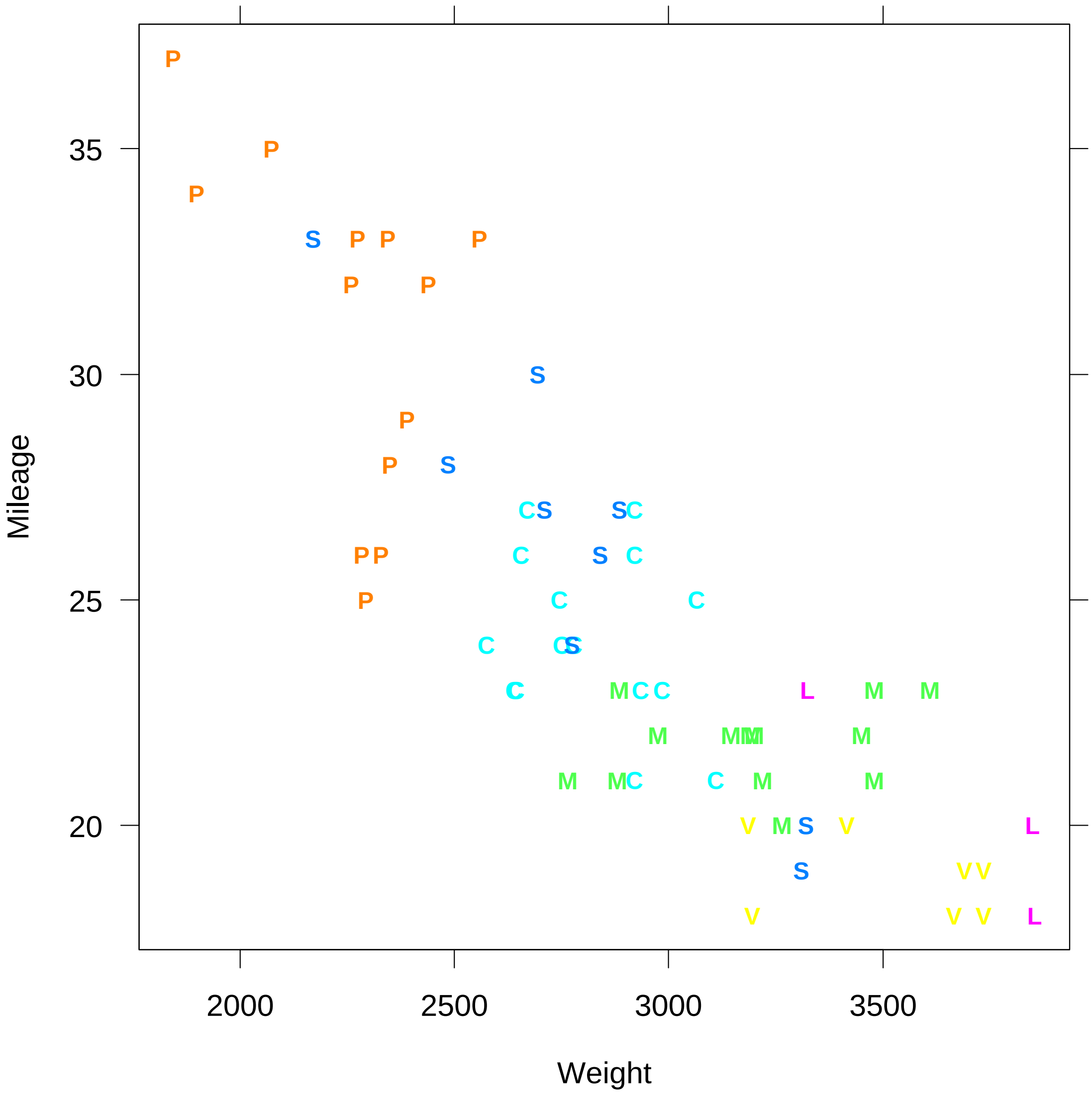


Figure 14.2

`panel.superpose()` will also superpose curves. In Figure 14.3, a line and a quadratic are superposed:

```
x <- seq(0, 1, length = 50)
linquad <-c(x, x^2)
x <- rep(x, 2)
which <- rep(c("linear","quadratic"), c(50,50))

xyplot(linquad ~ x,
  xlab = "Argument",
  ylab = "Functions",
  aspect = 1,
  groups = which,
  type = "l",
  panel = panel.superpose
  )
```

The argument `type=` controls the method of plotting. For `type="p"`, the default, the data are rendered by plotting symbols; the default has been used to produce Figures 14.1 and 14.2. For `type="l"`, the data are rendered by lines; this has been used to produce Figure 14.3.

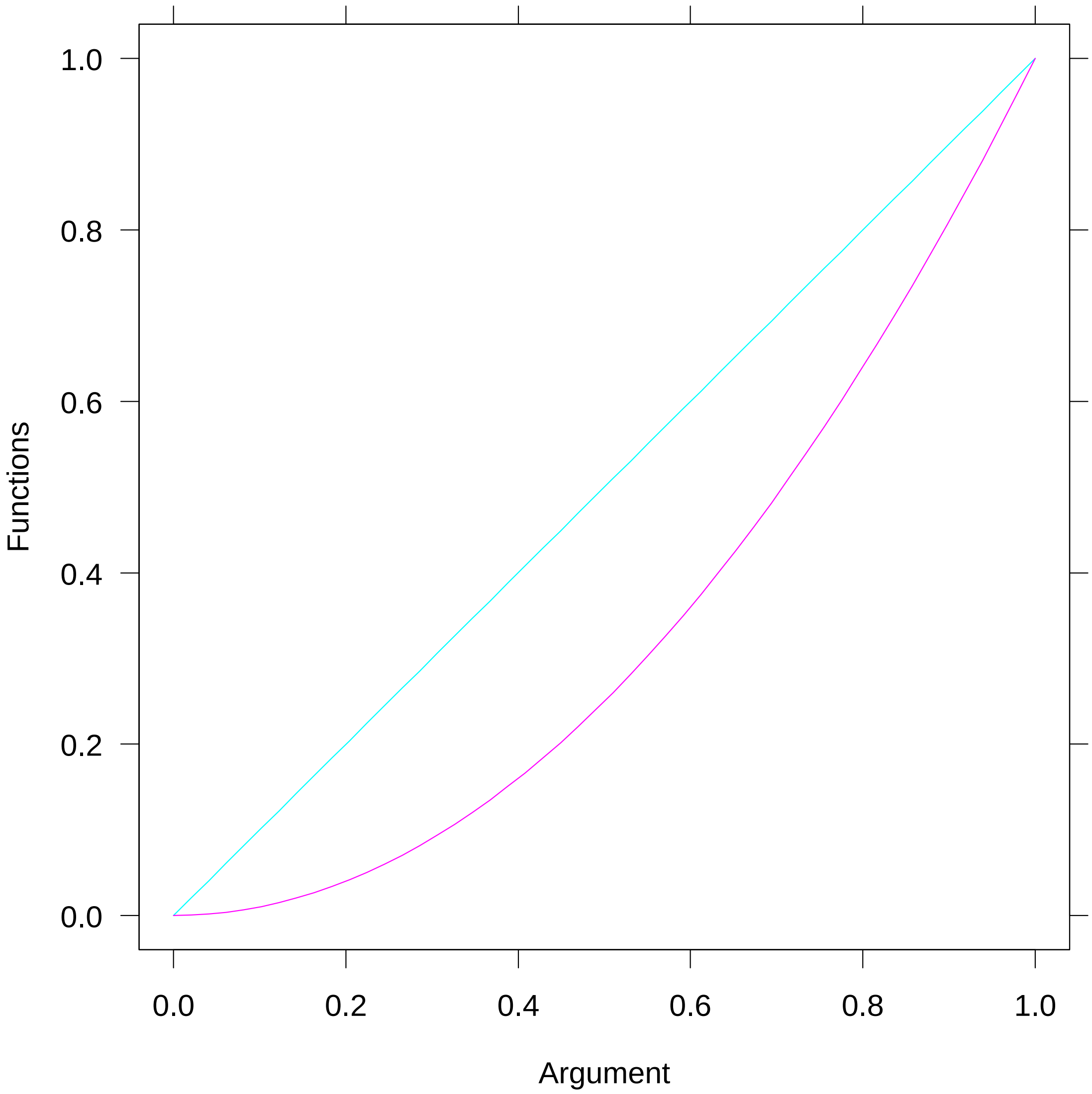


Figure 14.3

`panel.superpose()` uses the graphical parameters in the Trellis setting `superpose.symbol` for the default plotting symbols. For black and white postscript, the setting results in different symbol types:

```
> trellis.device(postscript)
> trellis.par.get("superpose.symbol")
$pch:
[1] "\001" "+"    ">"    "s"    "w"    "#"    "{"

$col:
[1] 1 1 1 1 1 1 1

$cex:
[1] 0.85 0.85 0.85 0.85 0.85 0.85 0.85

$font:
[1] 1 1 1 1 1 1 1
```

There are seven symbols, providing for up to seven groups. The symbols are shown in the first panel of the top row of Figure14.4, drawn by `trellis.settings()`. If there are two groups, the first two symbols are used; if there are three groups, the first three symbols are used; and so forth. The setting for the default line types is `superpose.line`:

```
> trellis.par.get("superpose.line")
$lwd:
[1] 1 1 1 1 1 1 1

$lty:
[1] 1 2 3 4 5 6 7

$col:
[1] 1 1 1 1 1 1 1
```

There are seven line types, shown in the second panel of the top row of Figure 14.4.

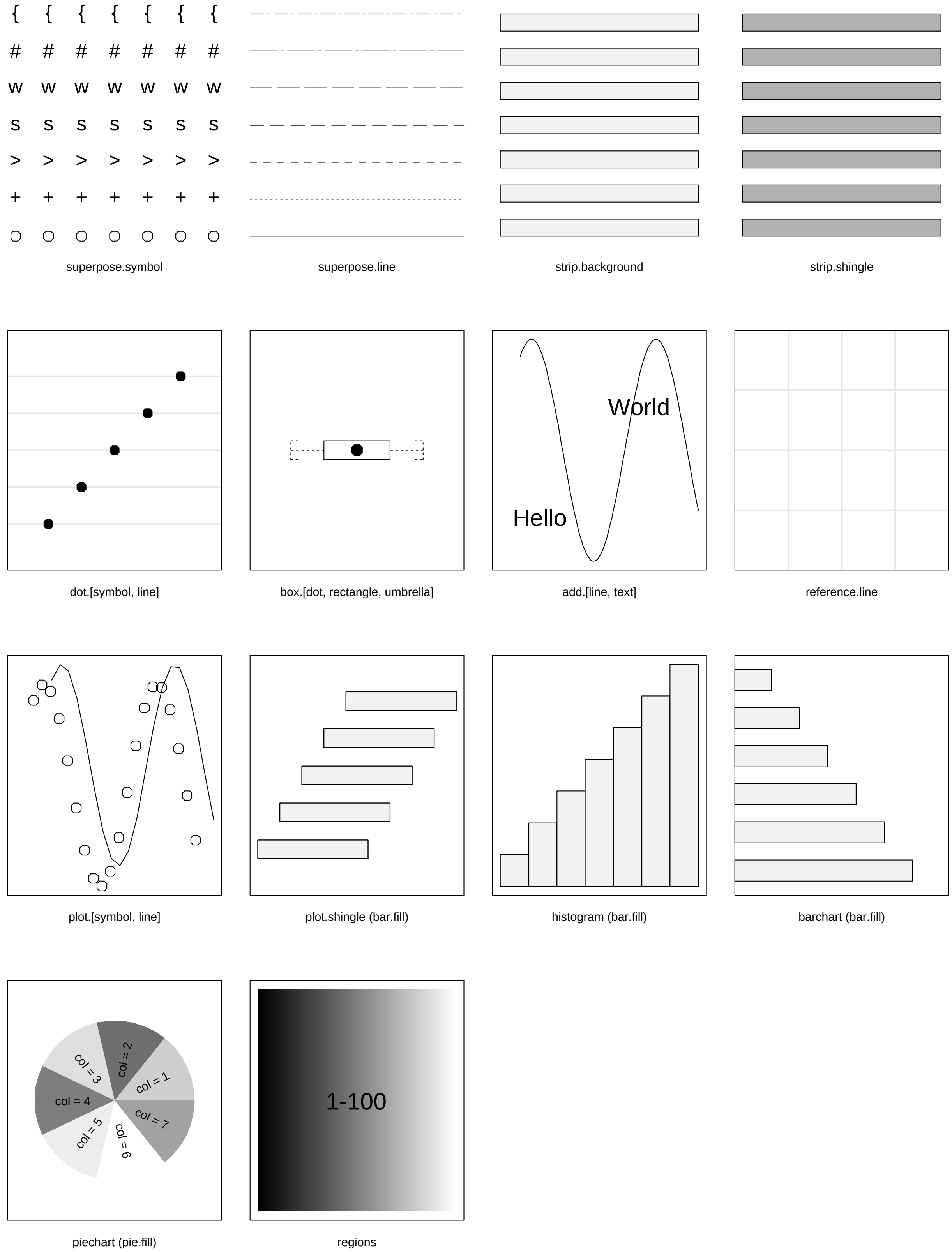

superpose.symbol
superpose.line
strip.background
strip.shingle
World
Hello
dot.[symbol, line]
box.[dot, rectangle, umbrella]
add.[line, text]
reference.line
plot.[symbol, line]
plot.shingle (bar.fill)
histogram (bar.fill)
barchart (bar.fill)
col = 1
col = 2
col = 3
col = 4
col = 5
col = 6
col = 7
1-100
piechart (pie.fill)
regions

Figure 14.4

`panel.superpose` can be used with any general display function where superposing different groups of values makes sense. For example, we can superpose datasets with `xyplot()`, as in Figures 14.1 to 14.3, or with `dotplot()`, or with many of the other general display functions. By achieving superposition through the panel function, we do not need a special superposition general display function for each type of graphical method, which makes things much simpler.

Figure 14.5 is a dot plot of the barley data discussed in chapter 1:

```
barley.plot <- dotplot(variety ~ yield | site, data = barley,
  panel = function(x, y, ...) {
    dot.line <- trellis.par.get("dot.line")
    abline(h = unique(y), lwd = dot.line$lwd,
      lty = dot.line$lty, col = dot.line$col)
    panel.superpose(x, y, ...)
    },
  groups = year,
  layout = c(1, 6), aspect = .5,
  xlab = "Barley Yield (bushels/acre)")
barley.plot
```

On each panel, data for two years are displayed, and the years, 1931 and 1932, are distinguished by different plotting symbols. The plot has been saved in the `trellis` object `barley.plot` for use later on.

For Figure 14.5, the general display function `dotplot()` has not sent the factor `variety` to the panel function to be the `y` vector for the function, but rather has sent a numeric vector of values 1 to 10 with 1 corresponding to the first of the 10 levels of the factor, with 2 corresponding to the second level, and so forth. And the display function has sent the values of `yield` as the vector `x`. The conditioning vector is `site`; thus on each panel there are 20 values of `x` and 20 values of `y`; for each level of variety, there are two values of `x` (one for 1931 and one for 1932) and two values of `y`, and there are 10 levels of variety. The plotting symbols are drawn by `panel.superpose()` at the 20 values of `x` and `y` on each panel.

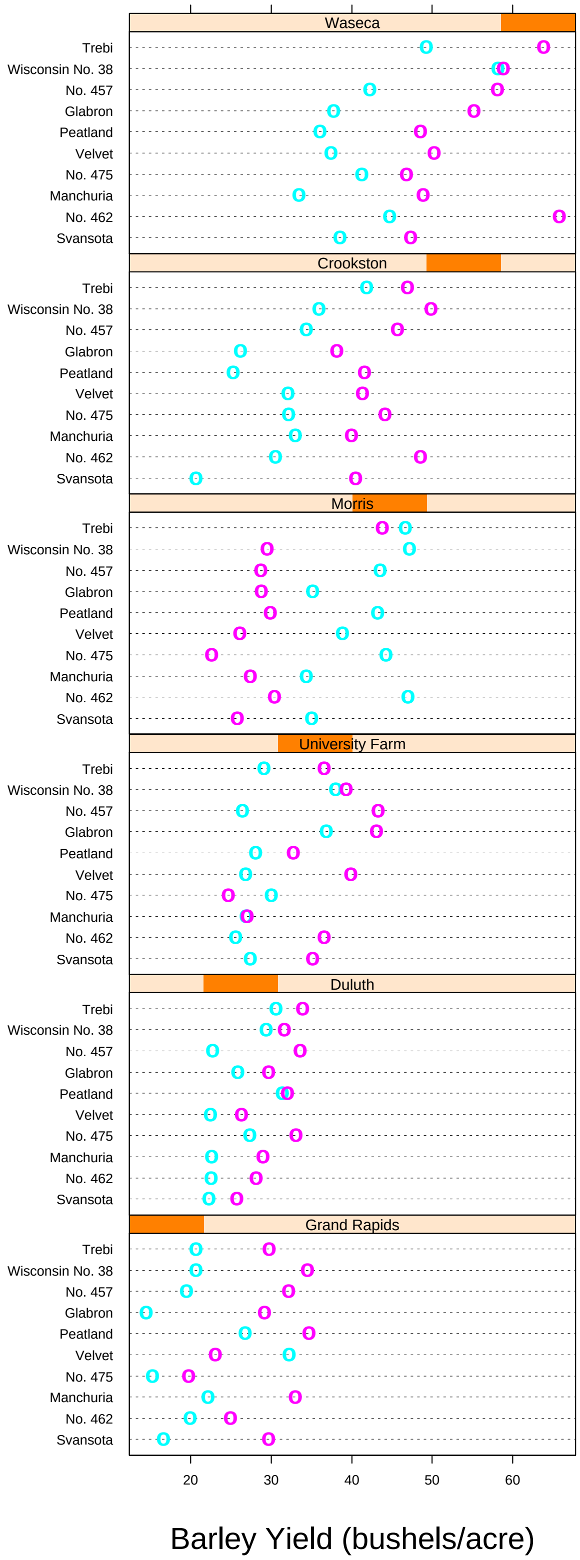

Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
Barley Yield (bushels/acre)

Figure 14.5

The panel function for this `dotplot()` example is more complicated than that for the `xyplot()` examples because, along with superposing the plotting symbols by `panel.superpose()`, the horizontal lines of the dot plot must be drawn. `abline()` draws the lines at the unique values of `y`. The characteristics of the line are specified by the Trellis setting `dot.line`.

14.2 key=

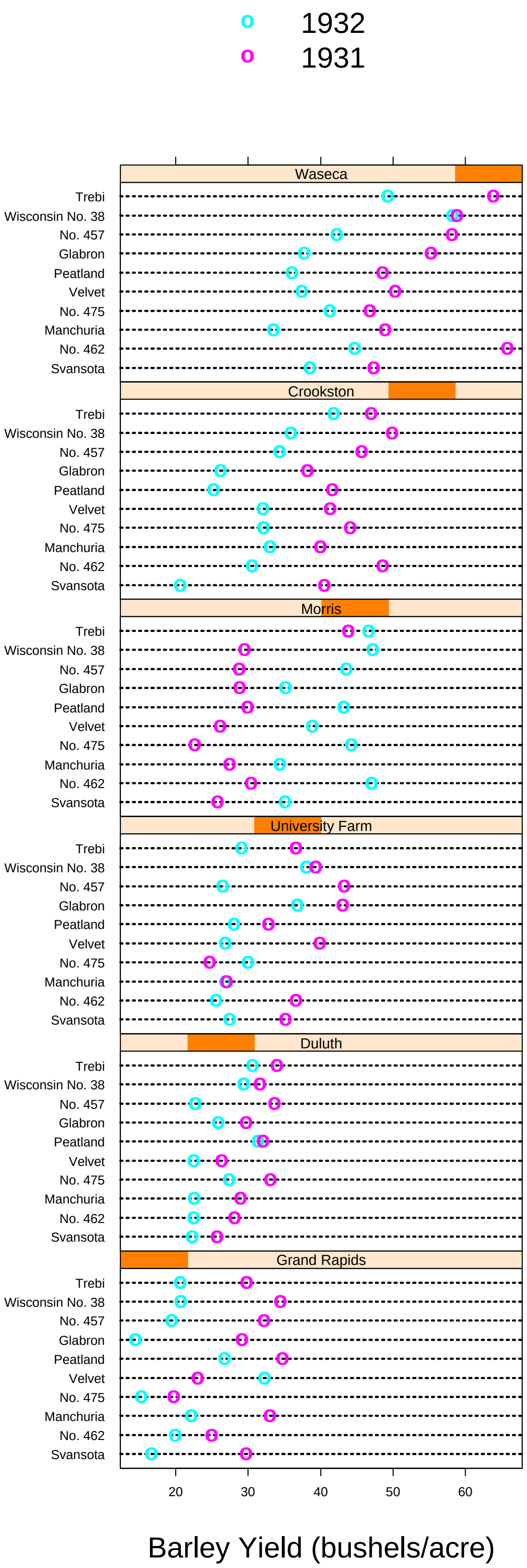
A key can be added to a Trellis display through the argument `key=` of the general display functions. The argument is a list. With one exception, the component names are the names of the arguments of the function `key()`, which actually does the drawing of the key, so the values of these components are given to the corresponding arguments of `key()`. The exception is the component `space=` which can leave extra space for a key in the margins of the display.

`key=` is easy to use yet is quite powerful; it has the capability to draw most keys used in practice and many yet to be invented.

Figure 14.6 adds a key to Figure 14.5:

```
update(barley.plot,
  key = list(
    points = Rows(superpose.symbol, 1:2),
    text = list(levels(barley$year))
    )
  )
```

The figure is drawn using `update()` to alter `barley.plot`, the object that produced Figure 14.5. The component `text` of `key=` is a list with the year names. The component `points` is a list with the graphical parameters of the two symbols used by `panel.superpose` to plot the data. These parameters are from the Trellis setting `superpose.symbol`, which `panel.superpose` uses to draw the plotting symbols.



1932
1931
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
Barley Yield (bushels/acre)

Figure 14.6

We want to give the component `points` only the parameters of the symbols used in Figure 14.6, so the function `Rows` extracts the first two elements of each component of `superpose.symbol`:

```
> trellis.device(postscript)
> Rows(trellis.par.get("superpose.symbol"), 1:2)
$pch:
[1] "o" "+"

$col:
[1] 1 1

$cex:
[1] 1 1

$font:
[1] 1 1
```

For Figure 14.6, the key has two entries, one for each year. If there had been four years there would have been four entries. Each entry has two items; as we shall see, we can specify more items if we choose. The order of the items is the order of specification in `key=`; in the above expression that draws Figure 14.6, `points` is first and `text` is second, so in the key, the symbol is the first item and then the text is the second item. Had we specified `text` first, the symbol would have followed the text in each entry.

In Figure 14.6, the two entries, by default, are drawn as an array with one column and two rows. We can change this by the argument `columns=`. In Figure 14.7, there are two columns. In addition, we have switched the order of the symbols and the text:

```
update(barley.plot,
  key = list(
    text = list(levels(barley$year)),
    points = Rows(superpose.symbol, 1:2),
    columns = 2))
```

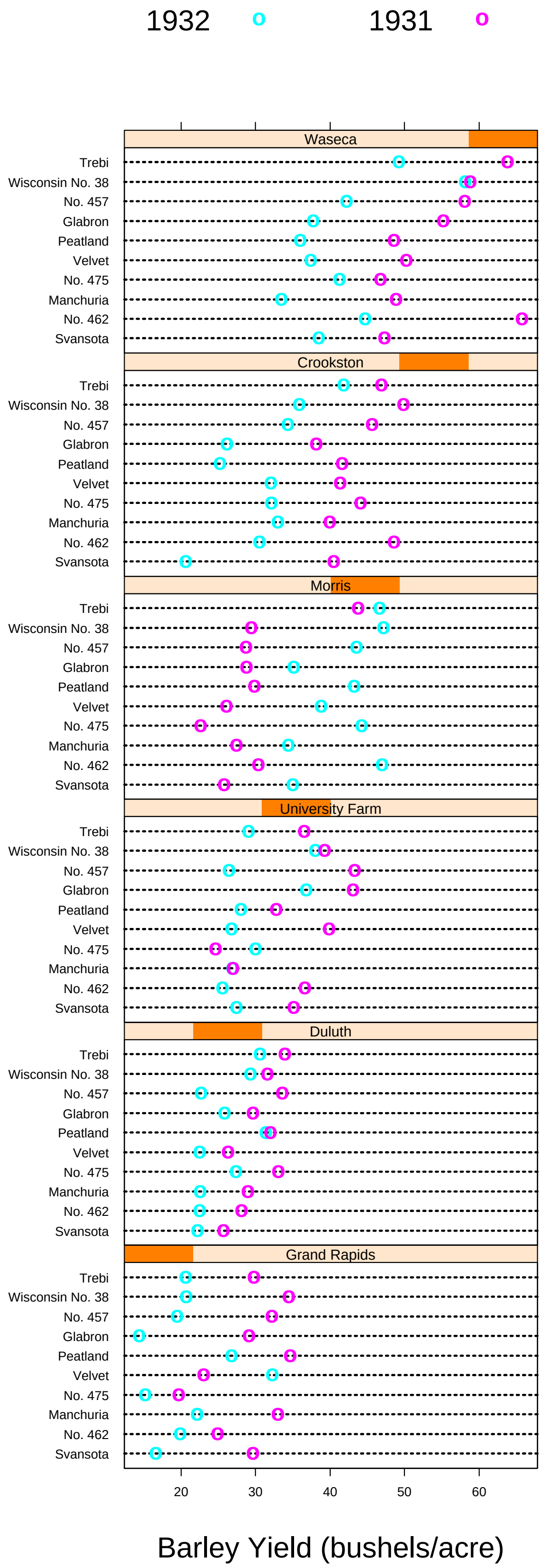

1932
1931
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
Barley Yield (bushels/acre)

Figure 14.7

The argument `space=` allocates space for the key in the margins. It takes one of four values — `"top"`, `"bottom"`, `"right"`, `"left"` — allocating the space on the side of the graph described by the value. So far, it has been allocating space at the top, which is the default, and placing the key in the allocated space. More will be said about `space=` later.

In Figure 14.7, the default location of the key seems a bit too far from the rest of the graph. The key has been repositioned in Figure 14.8, and a border has been drawn around it:

```
update(barley.plot,
  key = list(
    points = Rows(superpose.symbol, 1:2),
    text = list(levels(barley$year)),
    columns = 2,
    border = 1,
    x = .5,
    y = 1.02,
    corner = c(.5,0)
    ))
```

The argument `border=` draws a border; it takes a number that specifies the color in which the border should be drawn.

The repositioning uses two coordinate systems. The first describes locations in the rectangle that just encloses the panels of the display, but not including the tick marks; the lower left corner of this panel rectangle has coordinates $(0, 0)$, and the upper right corner has coordinates $(1, 1)$. A location in the panel rectangle is specified by the components `x` and `y`. The second coordinate system describes locations in the border rectangle of the key, which is shown when the border is drawn, as in Figure 14.7; the lower left corner of the key rectangle has coordinates $(0, 0)$, and the upper right corner has coordinates $(1, 1)$. A location in the border rectangle is specified by the component `corner`, a vector with two elements, the horizontal and vertical coordinates. The key is positioned so that the locations specified by the two coordinate systems are at the same place on the graph.

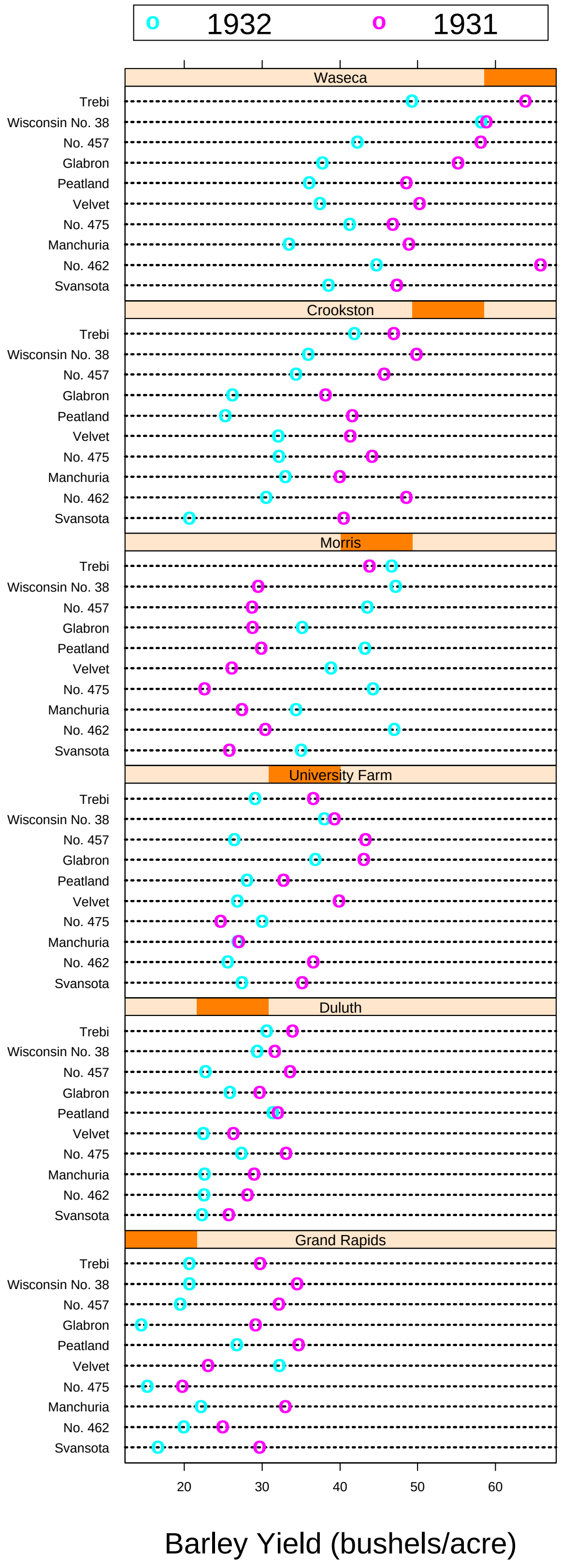

o 1932
o 1931
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
Barley Yield (bushels/acre)

Figure 14.8

Having two coordinate systems makes it far easier to get the key to a desired location quickly, often on the first try.

Notice that we specified `space=` to be `"top"` in Figure 14.8. The reason is this: As soon as we specify a value for any of the coordinate arguments `x`, `y`, or `corner`, no default space is allocated in any margin location unless we explicitly use the argument `space=`. In Figures 14.6 and 14.7, we did not use the coordinate arguments, so `space=` defaulted to `"top"`.

In Figure 14.9, space is allocated to the right.

```
update(barley.plot,
  key = list(
    points = Rows(superpose.symbol, 1:2),
    text = list(levels(barley$year)),
    space = "right"
    )
  )
```

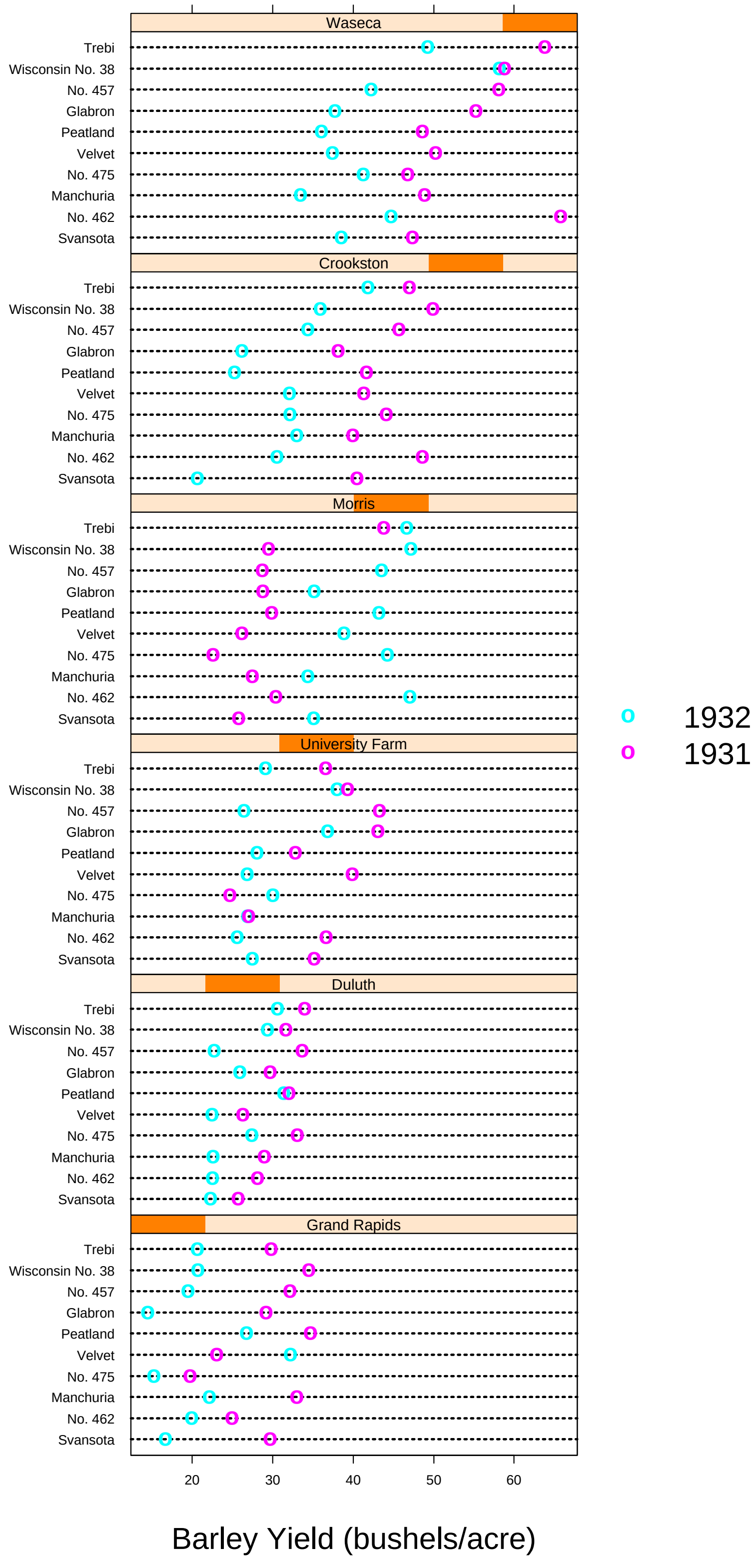

Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
Barley Yield (bushels/acre)
1932
1931

Figure 14.9

In Figure 14.10 some changes have been made to Figure 14.9. A border has been drawn, and the key is positioned by putting the upper left corner of the border rectangle at the same vertical position as the top of the panel rectangle and at a horizontal position slightly to the right of the right side of the panel rectangle.

```
update(barley.plot,
  key = list(
    text = list(levels(barley$year)),
    points = Rows(superpose.symbol, 1:2),
    space = "right",
    corner = c(0, 1),
    x = 1.05,
    y = 1,
    border = 1
    )
  )
```

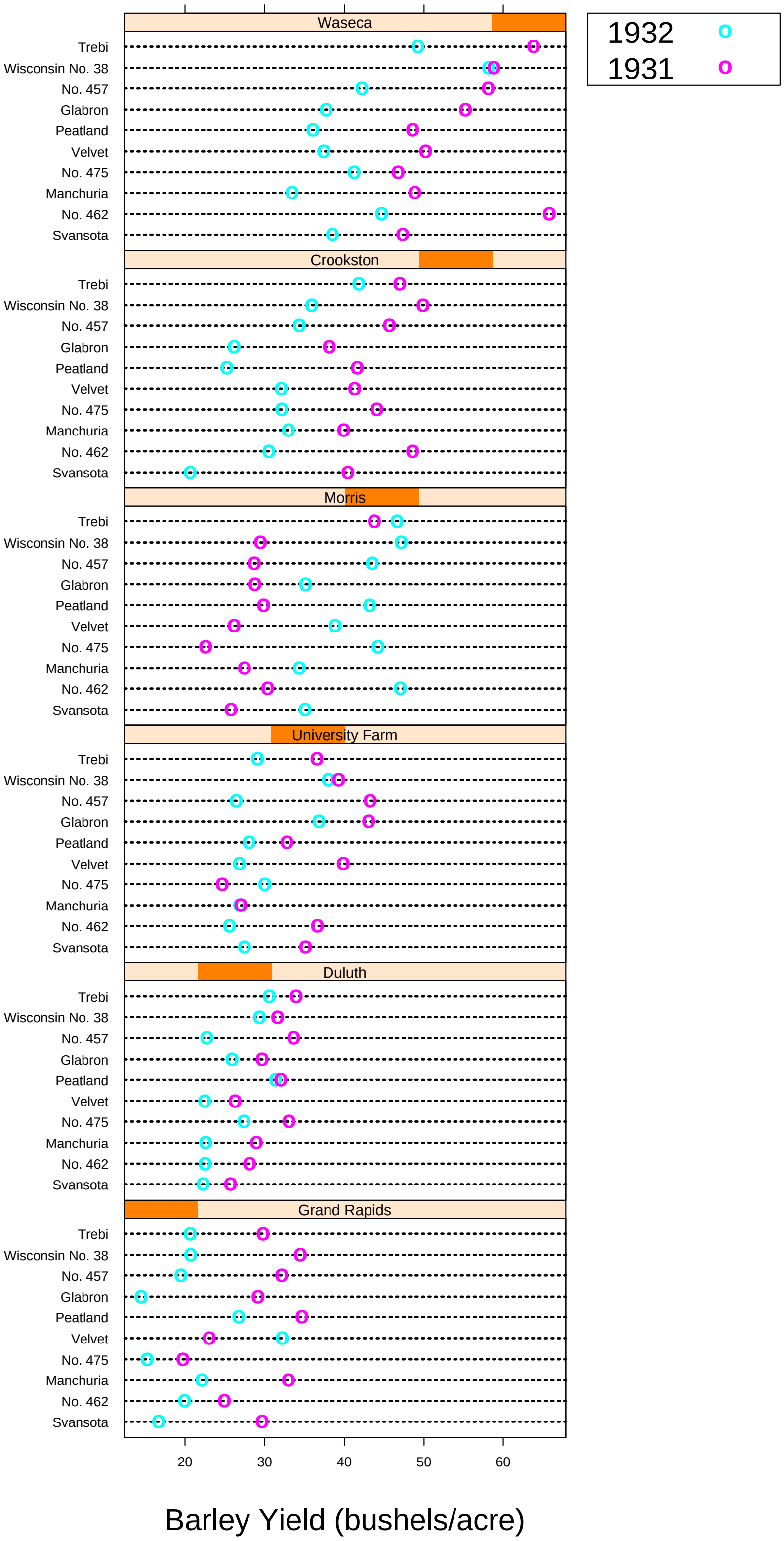

Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
Barley Yield (bushels/acre)
1932 o
1931 o

Figure 14.10

So far we have seen that components `points` and `text` can be used to create items in key entries. A third component, `lines`, draws line items. To illustrate this, let us return to Figure 14.1, the first plot in this chapter, which graphs `Mileage` against `Weight` for six types of vehicles. Figure 14.11 makes the plot again and adds two loess smooths with two different values of the smoothing parameter `span`:

```
superpose.line <- trellis.par.get("superpose.line")
superpose.line$col[3:6] <- 0
superpose.symbol <- trellis.par.get("superpose.symbol")

xyplot(Mileage ~ Weight,
  data = fuel.frame,
  groups = Type,
  aspect = 1,
  panel = function(x, y, ...){
    panel.superpose(x, y, ...)
    panel.loess(x, y, span = 1/2,
      lwd=superpose.line$lwd[1], lty=superpose.line$lty[1],
      col=superpose.line$col[1])
    panel.loess(x, y, span = 1,
      lwd=superpose.line$lwd[2], lty=superpose.line$lty[2],
      col = superpose.line$col[2])},
  key = list(
    transparent = T,
    x = .95, y = .95,
    corner = c(1,1),
    lines = c(Rows(superpose.line, 1:6),
              list(size = c(3,3,0,0,0,0))),
    text = list(c("Span = 0.5", "Span = 1.0", rep("", 4))),
    points = Rows(superpose.symbol, 1:6),
    text = list(levels(fuel.frame$Type))
    )
  )
```

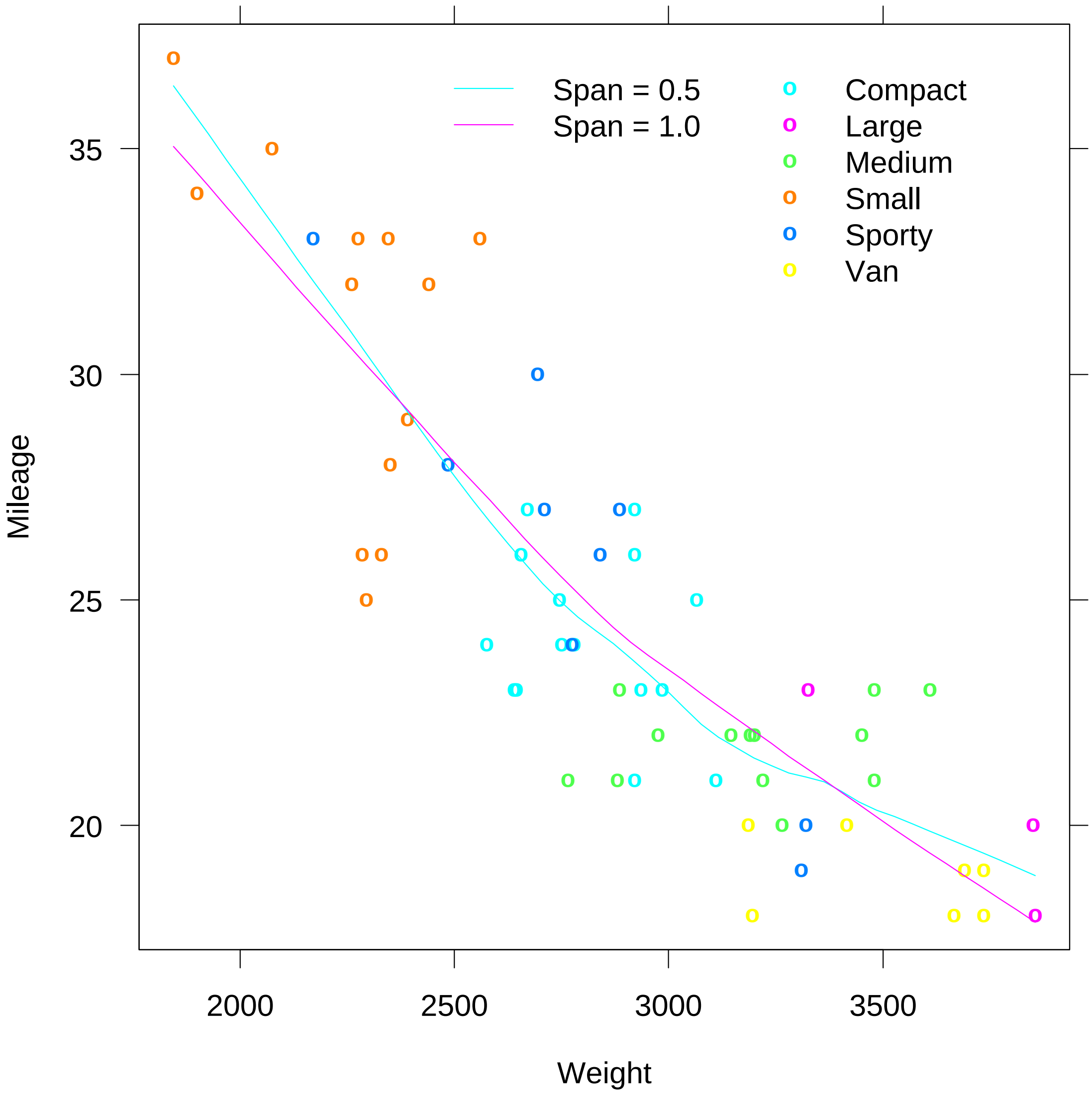

Span = 0.5
Span = 1.0
Compact
Large
Medium
Small
Sporty
Van
35
30
25
20
Mileage
2000
2500
3000
3500
Weight

Figure 14.11

Chapter 15

Data Structures

Trellis Graphics uses the S-PLUS formula language to specify the data for plotting. This requires the data to be stored in datasets that work with formulas. Roughly speaking, this means the data variables must be either from a data frame or be vectors of the same length. (This is also true of the S-PLUS modeling functions such as `lm()`.) But in S-PLUS there are many other data structures. So that Trellis functions will be easy to use, three functions convert data structures of different kinds into data frames — `make.groups()`, `as.data.frame.array()`, and `as.data.frame.ts()`.

15.1 make.groups()

The function `make.groups()` takes several vectors and constructs a data frame with two components: `data` and `which`. For example, consider payoffs of the New Jersey Pick-It lottery from three time periods. The data are stored as three vectors of values. Suppose we want to make box plots to compare the three distributions: We first convert the three vectors to a data frame:

```
> lottery <- make.groups(lottery.payoff,
+ lottery2.payoff, lottery3.payoff)
> names(lottery)
[1] "data"  "which"
> levels(lottery$which)
[1] "lottery.payoff"  "lottery2.payoff" "lottery3.payoff"
```

The `data` component is simply the combined numbers from all the `make.groups` arguments. The `which` component is a factor with 3 levels, giving the names of the original data vectors. Now we can make the box plots, which are shown in Figure 15.1: lottery3.payoff)

```
bwplot(which ~ data, data = lottery, aspect = 1)
```

15.2 as.data.frame.array()

The function `as.data.frame.array()` converts arrays into data frames. Consider the object `iris`, a 3-way array of 50 measurements of 4 variables for each of 3 varieties of irises:

```
> dim(iris)
[1] 50  4  3
```

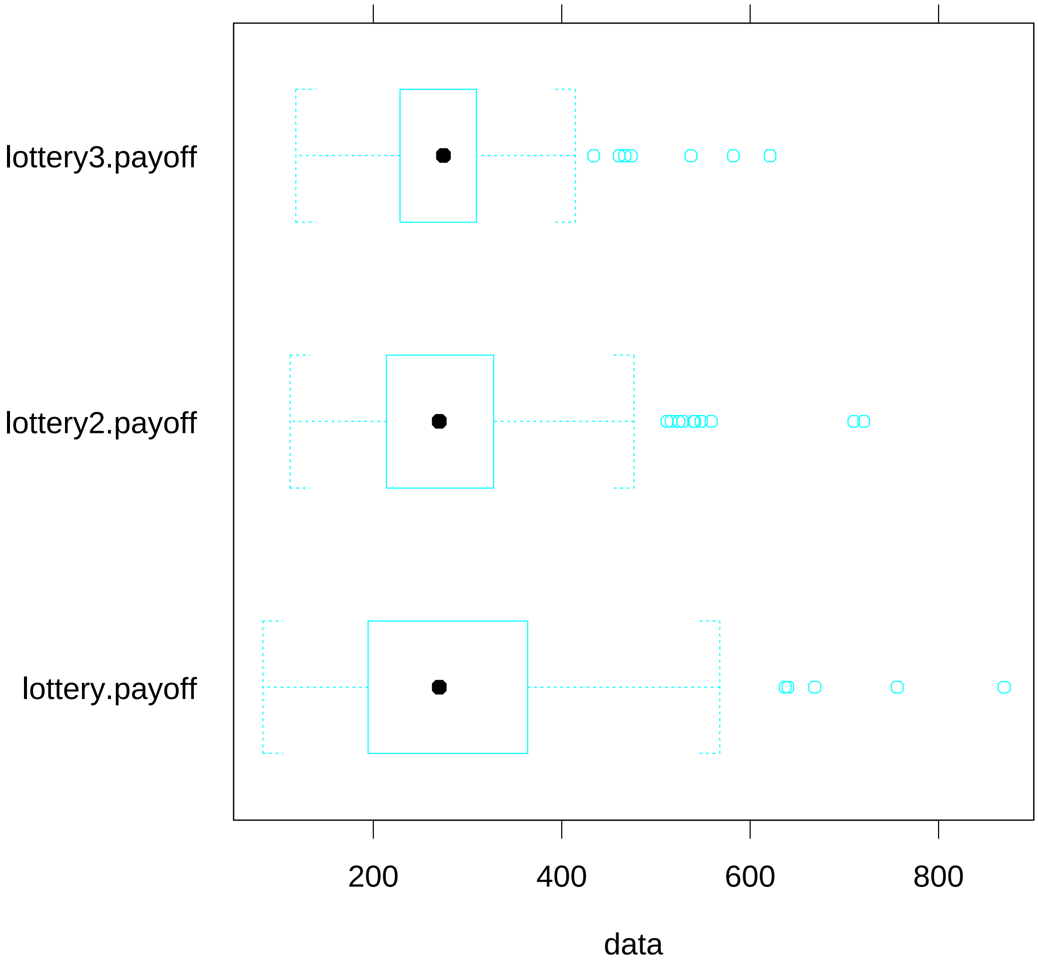

lottery3.payoff
lottery2.payoff
lottery.payoff
200
400
600
800
data

Figure 15.1

We can turn iris into a data frame in preparation for Trellis plotting by using:

```
iris.df <- as.data.frame.array(iris, col.dims = 2)
names(iris.df)[5:6] <- c("flower","variety")
```

The resulting data frame has what used to be its second dimension turned into 4 columns:

```
> iris.df[1:5,]
  Sepal L. Sepal W. Petal L. Petal W. flower variety
 1      5.1      3.5      1.4      0.2      1  Setosa
 2      4.9      3.0      1.4      0.2      2  Setosa
 3      4.7      3.2      1.3      0.2      3  Setosa
 4      4.6      3.1      1.5      0.2      4  Setosa
 5      5.0      3.6      1.4      0.2      5  Setosa
```

Figure 15.2 is a scatterplot matrix of the data:

```
superpose.symbol <- trellis.par.get("superpose.symbol")
for (i in 1:4)
  iris.df[,i] <- jitter(iris.df[,i])

splom(~iris.df[,1:4],
  key = list(
    space = "top", columns = 3,
    text = list(levels(iris.df$variety)),
    points = Rows(superpose.symbol, 1:3)
    ),
  varnames = c("Sepal Length\n(cm)", "Sepal Width\n(cm)",
    "Petal Length\n(cm)", "Petal Width\n(cm)"),
  groups = iris.df$variety,
  panel = panel.superpose)
```

To prevent exact overlap of many of the plotting symbols, the data have been jittered before plotting.

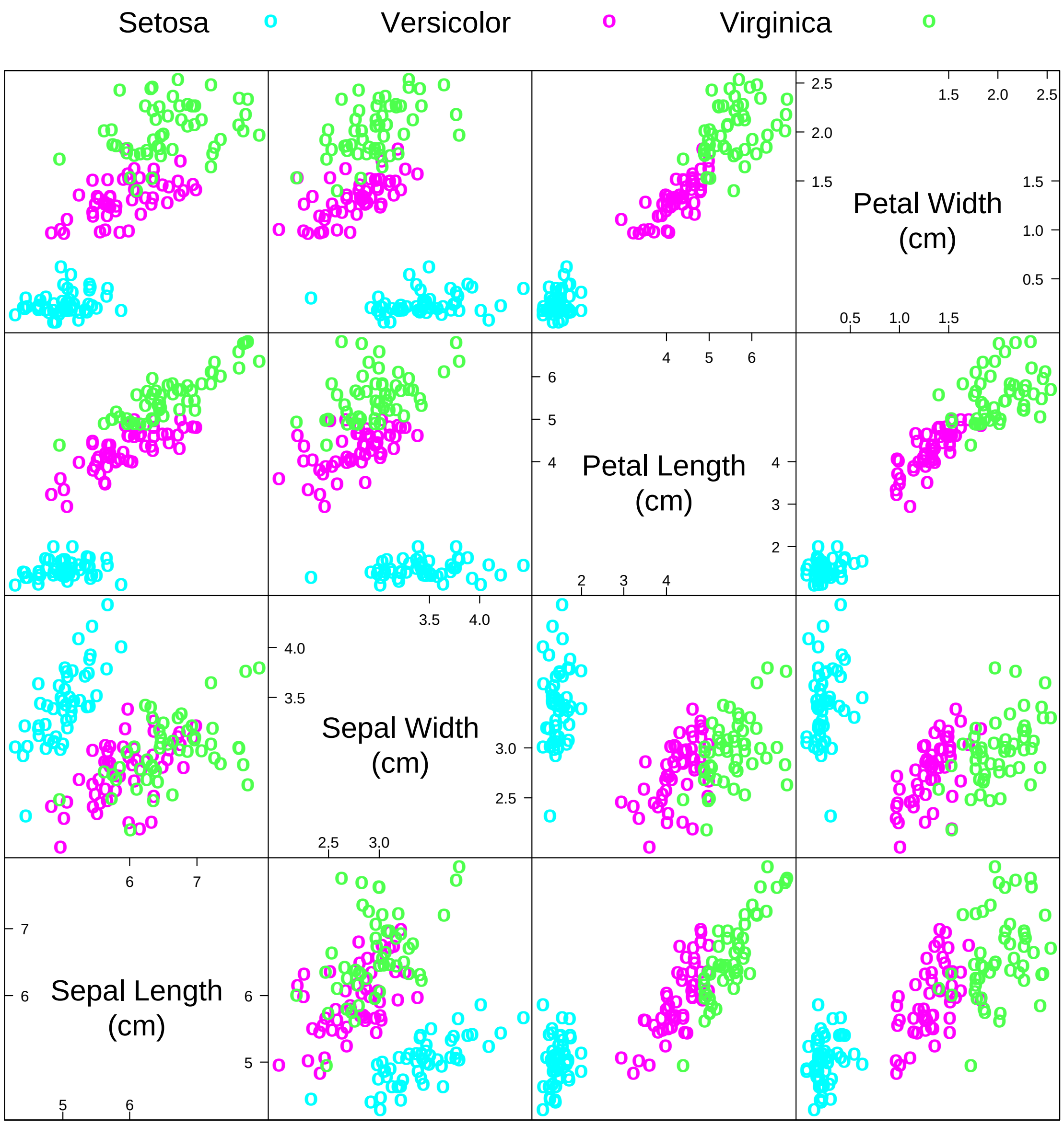

Setosa
Versicolor
Virginica
Petal Width
(cm)
Petal Length
(cm)
Sepal Width
(cm)
Sepal Length
(cm)

Figure 15.2

15.3 as.data.frame.ts()

The function `as.data.frame.ts()` takes one or more time series as arguments and produces a data frame with components named `series`, `which`, `time`, and `cycle`. The `series` component is the data from all of the time series combined into one long vector. The `time` component gives the time associated with each of the points (measured in the same units as the original series, e.g. years), and `cycle` gives the periodic component of the time (e.g. 1=Jan, 2=Feb, ...). Finally, the `which` component is a factor that tells which of the time series the measurement came from. In the following example there is only one series, `hstart`, but in general `as.data.frame.ts` can take many arguments:

```
> as.data.frame.ts(hstart)[1:5,]
   series  which     time cycle
 1   81.9 hstart 1966.000   Jan
 2   79.0 hstart 1966.083   Feb
 3  122.4 hstart 1966.167   Mar
 4  143.0 hstart 1966.250   Apr
 5  133.9 hstart 1966.333   May
```

Figure 15.3 graphs housing starts for each month separately from 1966 to 1974:

```
xyplot(series ~ time|cycle,
  data = as.data.frame.ts(hstart),
  type = "b",
  xlab = "Year",
  ylab = "Housing Starts by Month")
```

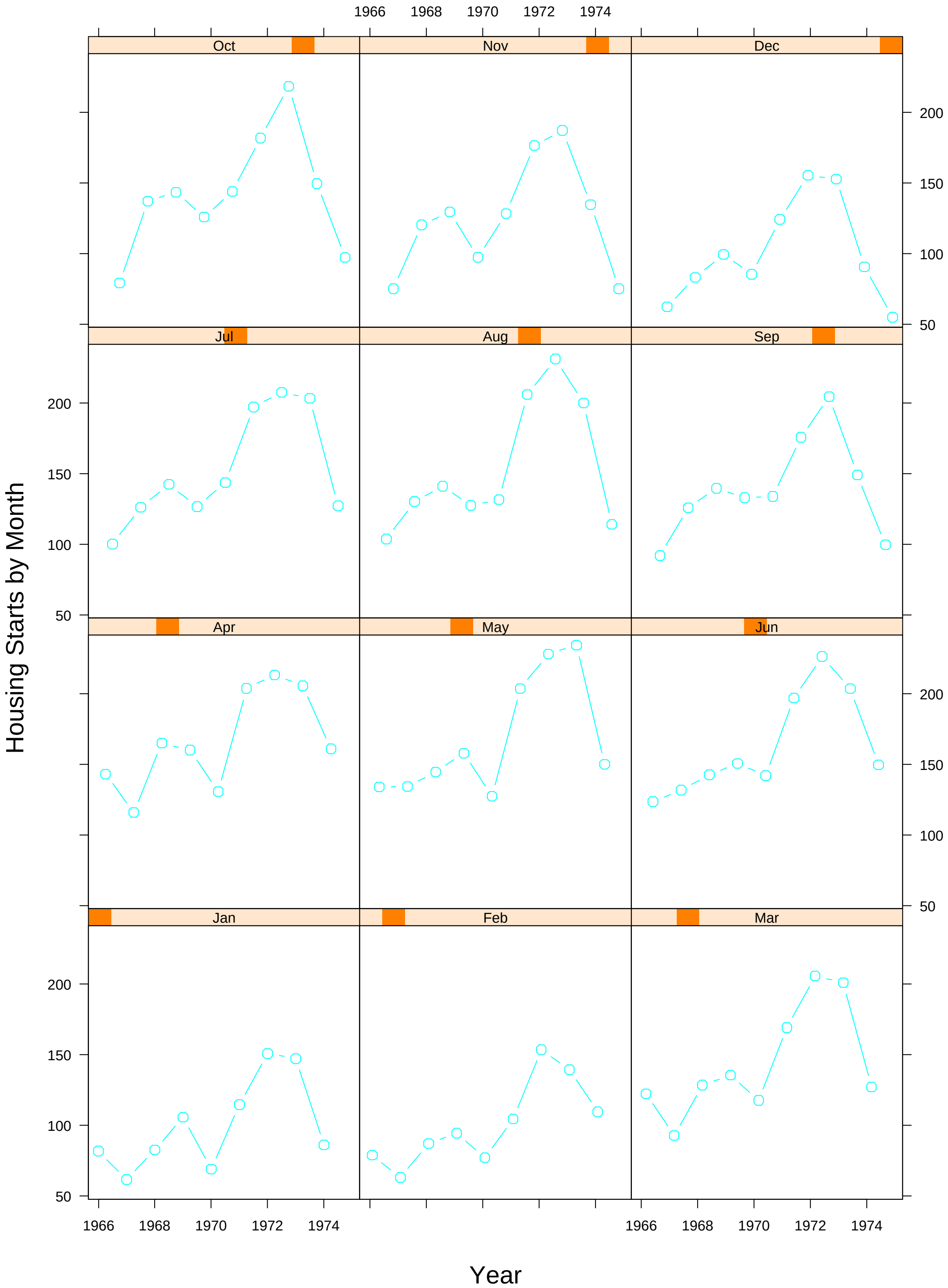

Oct
Nov
Dec
Jul
Aug
Sep
Apr
May
Jun
Jan
Feb
Mar
1966
1968
1970
1972
1974
50
100
150
200
Housing Starts by Month
Year

Figure 15.3

Chapter 16

More on Aspect Ratio and Scales: Prepanel Functions

Banking to 45° is an important display method built into Trellis Graphics through the argument `aspect=`. And the ranges of scales on the panels can be controlled by the arguments `xlim=` and `ylim=`, or by the argument `scales=`. Another argument, `prepanel=`, is a function that supplies information for the banking and range calculations.

16.1 prepanel=

Figure 16.1 is a graph of the ethanol data; NOx is graphed against E given C and loess curves have been superposed.

```
xyplot(NOx ~ E | C, data = ethanol,
  aspect = 0.5,
  layout = c(1,5),
  panel = function(x, y){
    panel.xyplot(x, y)
    panel.loess(x, y, span = 1/2, degree = 2)
    }
  )
```

There are now two things we would like to do with this plot, one involving the aspect ratio and the other involving the ranges of the scales.

First, we have set the aspect ratio to 1/2 using `aspect=`. We could have set `aspect=` to `"xy"` to carry out 45° banking of the line segments that connect the points of the plot, that is, the graphed values of `E` and `NOx`. But normally we do want to carry out banking of the raw data if they are noisy; rather we want to bank an underlying smooth pattern. In this example, we want to bank using the line segments of the loess curves.

Second, in the top panel, the loess curve exceeds the maximum value along the vertical scale and so is chopped off. It is important to understand why this happened. The scales where chosed based on the values of `E` and `NOx`. The loess curves were computed by the panel function after all of the scaling had been carried out. We would like a way for the scaling to take account of the values of the loess curve.

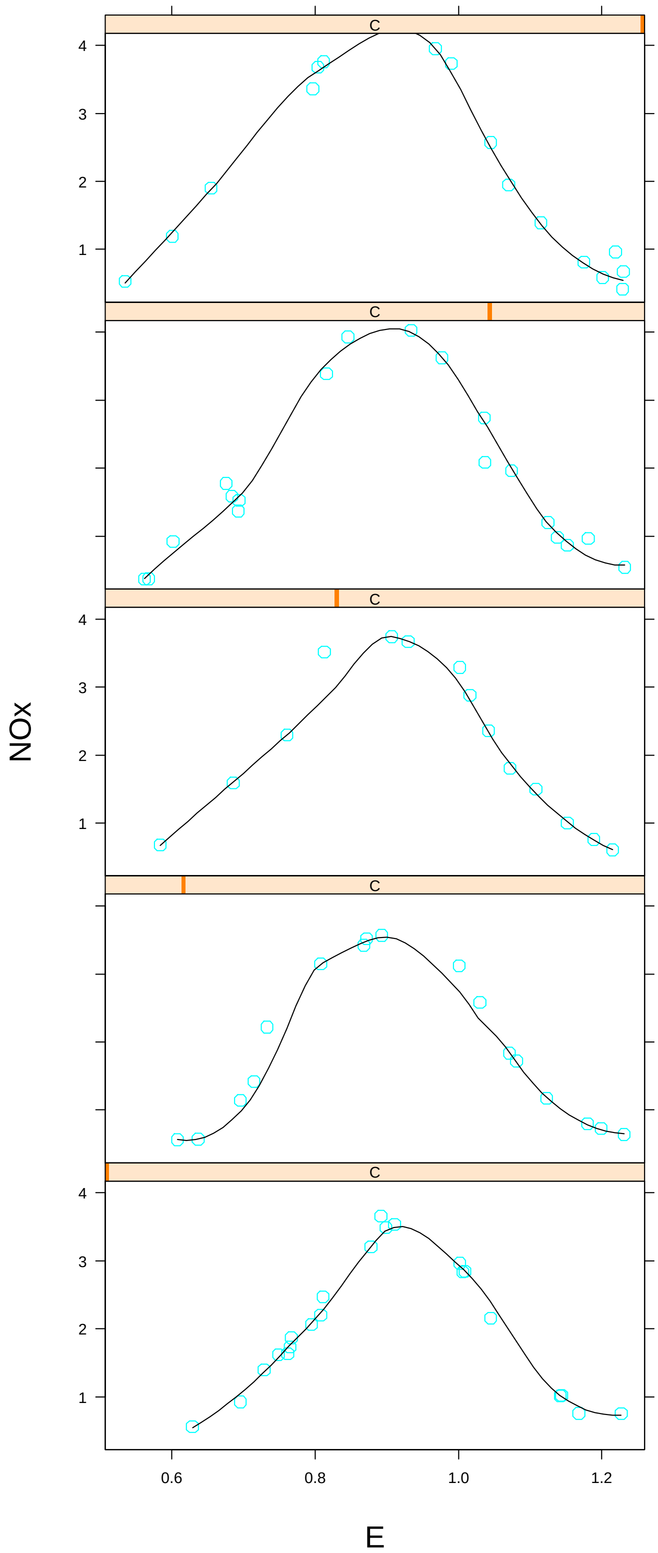


Figure 16.1

The argument `prepanel=` allows us to bank to 45° based on the loess curves and to take the curves into account in computing the ranges of the scales:

```
xyplot(NOx ~ E | C, data = ethanol,
  prepanel = function(x, y)
    prepanel.loess(x, y, span = 1/2, degree = 2),
  panel = function(x, y){
    panel.xyplot(x, y)
    panel.loess(x, y, span = 1/2, degree = 2)},
  layout = c(1,5))
```

The resulting display is shown in Figure 16.2.

`prepanel=` takes a function and does panel-by-panel computations, just like `panel=`, but these computations are carried out before the scales and aspect ratio are determined and so, can be used in their determination. The returned value of a prepanel function is a list with prescribed component names. These names are shown in the prepanel function `prepanel.loess`:

```
> prepanel.loess
function(x, y, ...)
{
  xlim <- range(x)
  ylim <- range(y)
  out <- loess.smooth(x, y, ...)
  x <- out$x
  y <- out$y
  list(xlim = range(x, xlim), ylim = range(y, ylim),
    dx = diff(x), dy = diff(y))
}
```

The component values `xlim` and `ylim` determine ranges for the scales just as they do when they are given as arguments of a general display function. The values of `dx` and `dy` are the horizontal and vertical changes of the line segments that are to be banked to 45°.

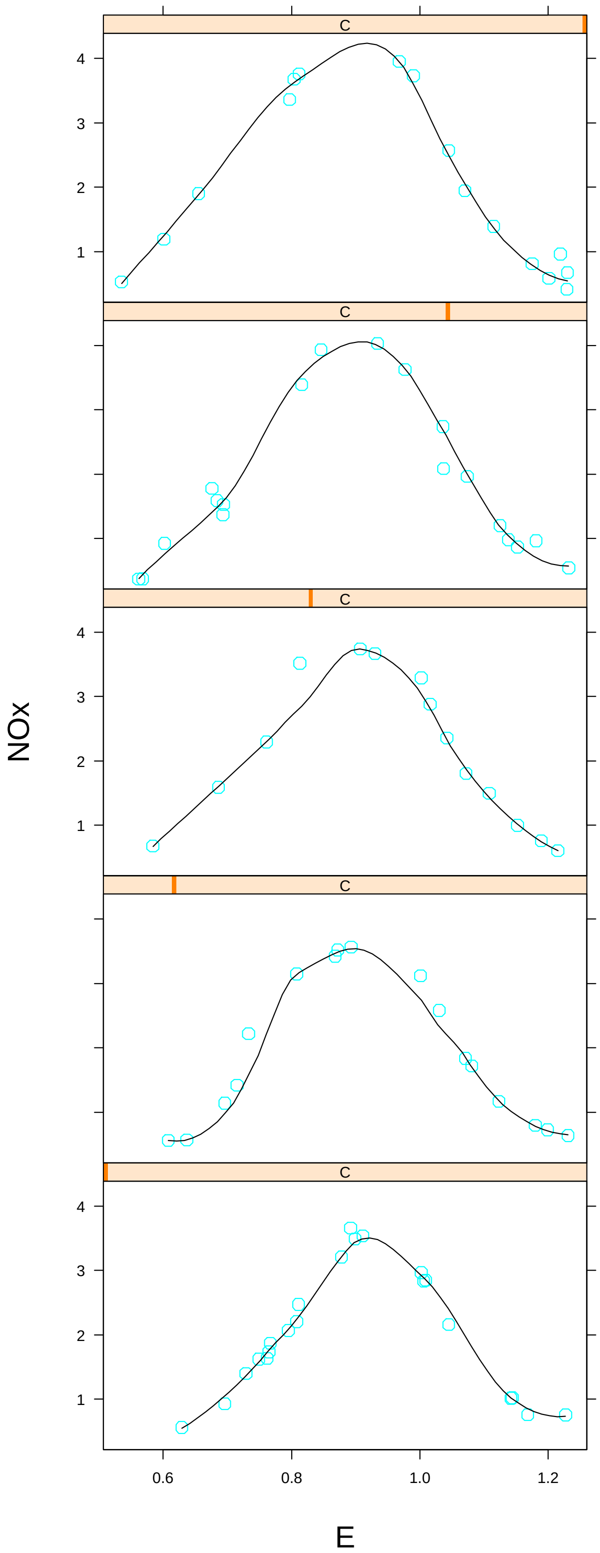


Figure 16.2

For Figure 16.2, `prepanel.loess` computes the smooths for all panels, computes values of `xlim` and `ylim` that insure that the curve will be included in the ranges of the scales, and then passes along the changes of the line segments that will make up the plotted curve. Any of the component names can be missing from the list; if either `dx` or `dy` is missing, the other must be as well. When `dx` and `dy` are present, they give the information needed for banking to 45° as well as the instruction to do so; thus `aspect=` should not be used as an argument when `dx` and `dy` are present.

Chapter 17

More on Multipanel Conditioning

The multipanel conditioning of Trellis Graphics has three more arguments that assist in the control of the layout, visual design, and labeling. `between=` puts space between adjacent columns or adjacent rows. `skip=` allows a panel position to be skipped when packets are sent to the panels for drawing. `page=` can add page numbers, text, or even graphics to each page of a multipage Trellis display.

17.1 between=

Figures 17.1 and 17.2 graph the barley data. In this two-page Trellis display, yield is plotted against site given variety and year.

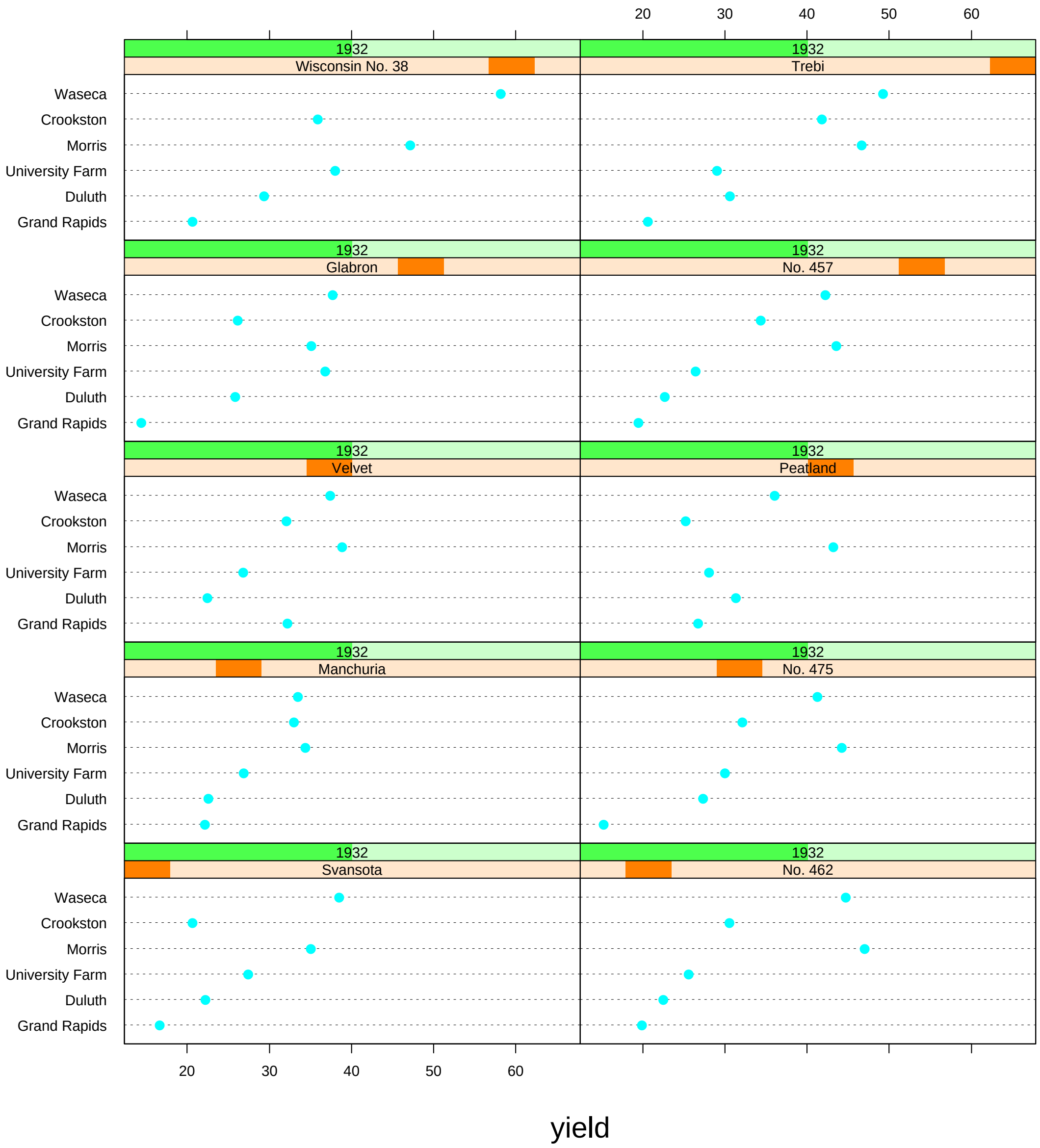

1932
Wisconsin No. 38
1932
Trebi
1932
Glabron
1932
No. 457
1932
Velvet
1932
Peatland
1932
Manchuria
1932
No. 475
1932
Svansota
1932
No. 462
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
20
30
40
50
60
yield

Figure 17.1

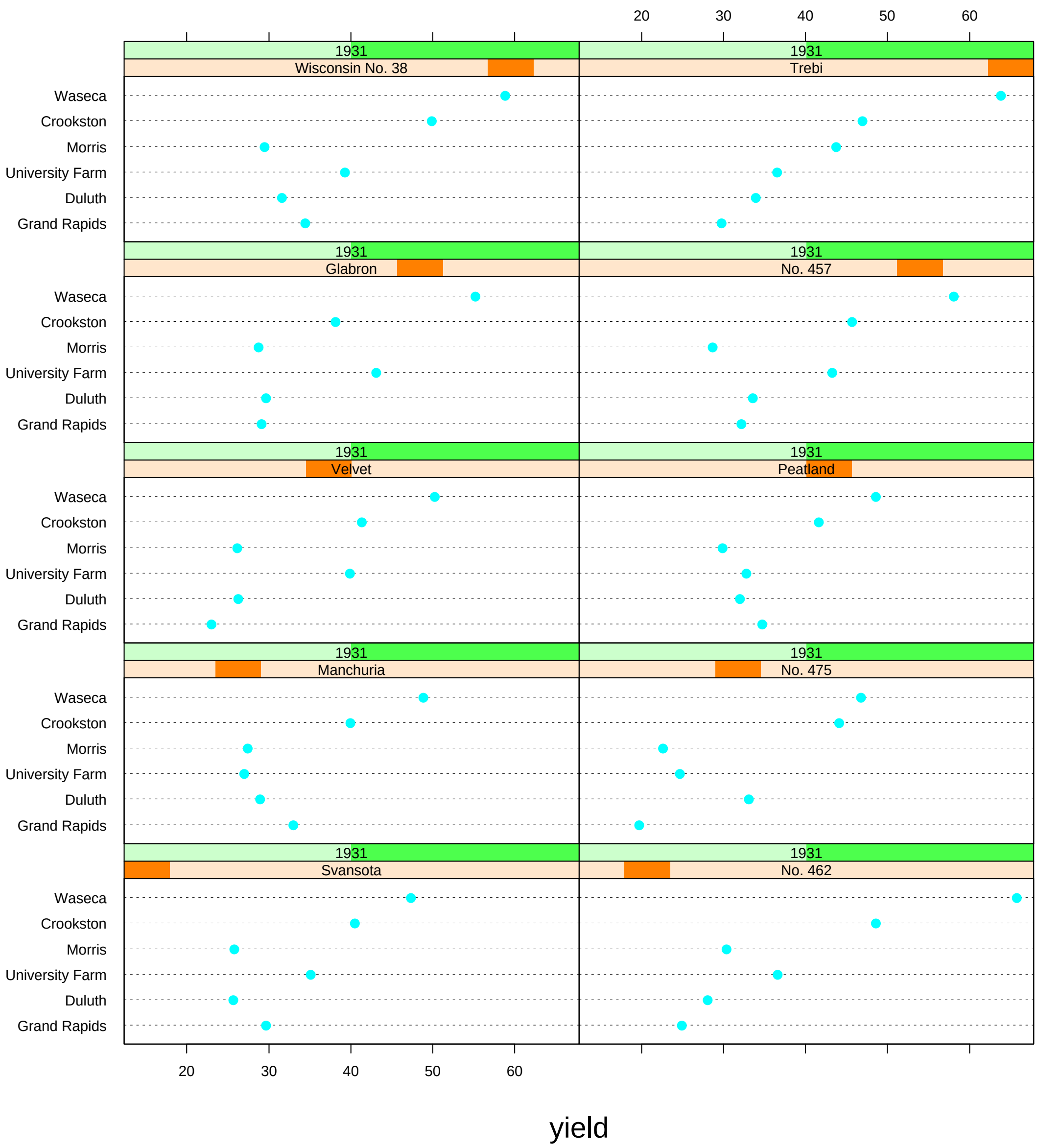

20
30
40
50
60
1931
Wisconsin No. 38
1931
Trebi
1931
Glabron
1931
No. 457
1931
Velvet
1931
Peatland
1931
Manchuria
1931
No. 475
1931
Svansota
1931
No. 462
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
yield

Figure 17.2

Figures 17.1 and 17.2 were produced by:

```
barley.plot <- dotplot(site ~ yield | variety * year,
  data = barley, aspect = "xy", layout = c(2, 5, 2))
barley.plot
```

The layout — 2 columns, 5 rows, and 2 pages — has put the measurements for 1931 on the first page and for 1932 on the second page. The display has been saved in `barley.plot` for future editing.

In Figure 17.3, the panels of Figures 17.1 and 17.2 have been squeezed into one page simply by changing `layout=` from (2,5,2) to (2,10,1):

```
barley.plot <- update(barley.plot, layout = c(2, 10, 1))
barley.plot
```

Rows 1 to 5 (starting from the bottom) have the 1932 data and rows 6 to 10 have the 1931 data. The change in the value of the year variable from rows 5 to 6 is indicated by the text of the strip label, but a stronger indication of a change would occur if there was a break in the display between rows 5 and 6.

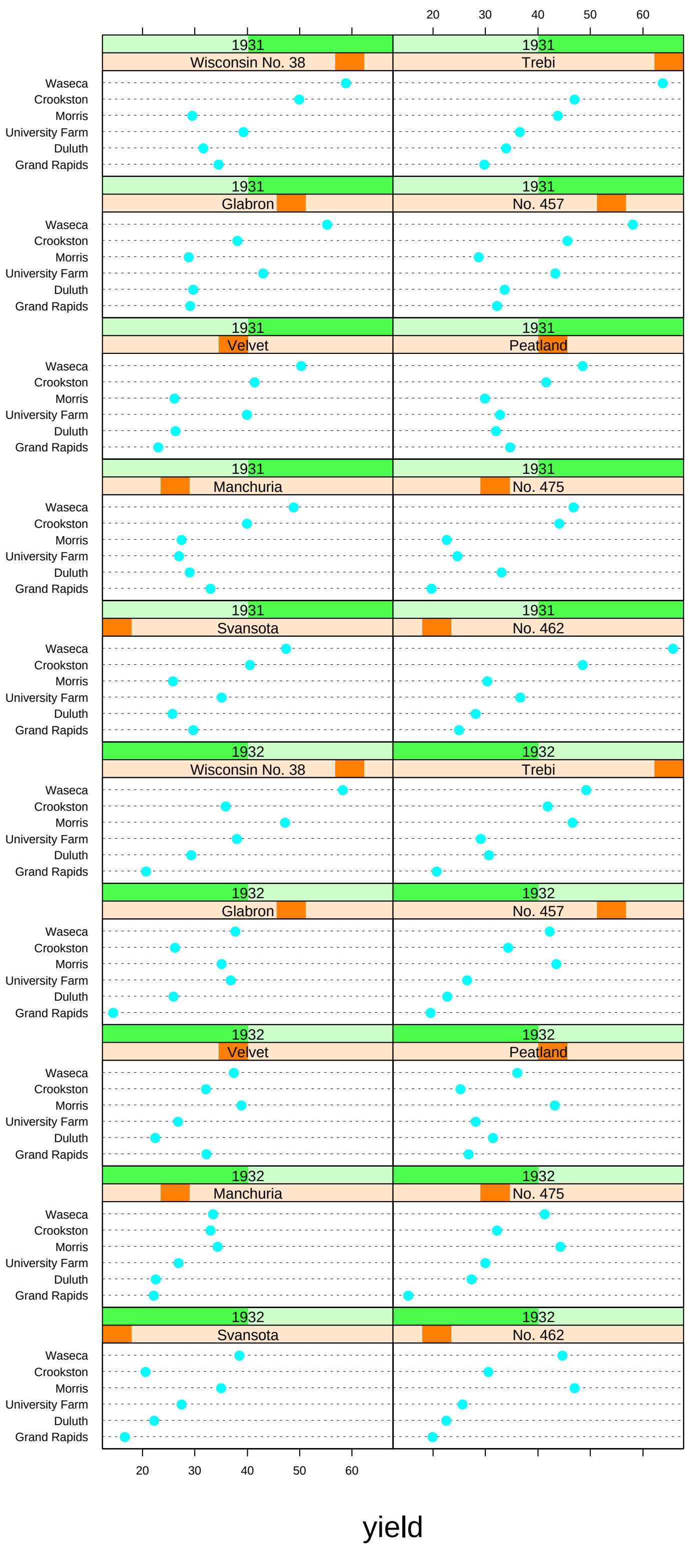

1931
Wisconsin No. 38
1931
Trebi
1931
Glabron
1931
No. 457
1931
Velvet
1931
Peatland
1931
Manchuria
1931
No. 475
1931
Svansota
1931
No. 462
1932
Wisconsin No. 38
1932
Trebi
1932
Glabron
1932
No. 457
1932
Velvet
1932
Peatland
1932
Manchuria
1932
No. 475
1932
Svansota
1932
No. 462
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
20
30
40
50
60
yield

Figure 17.3

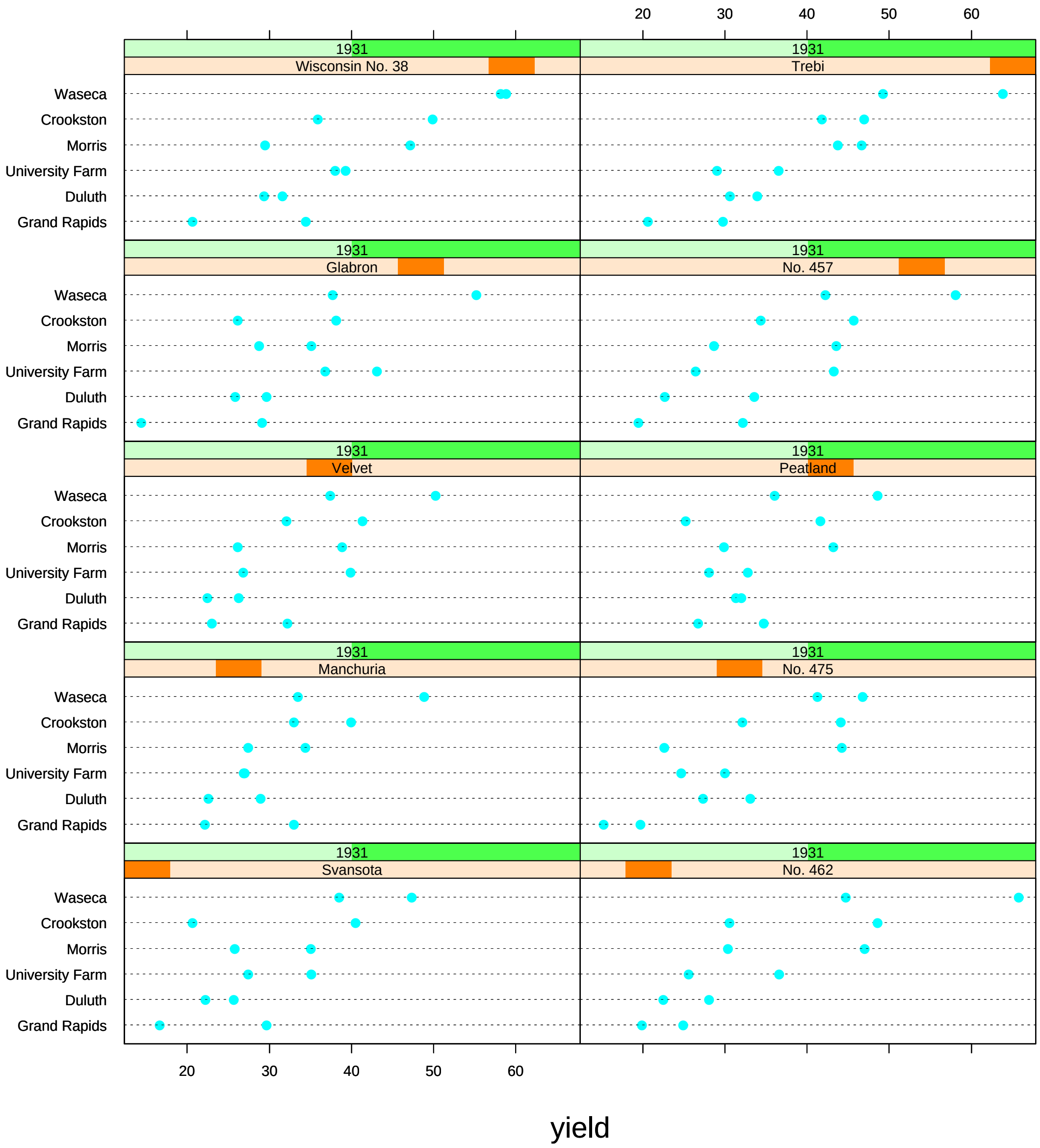
The argument `between=` can be used to insert space between adjacent rows or adjacent columns of a Trellis display. This is illustrated in Figure 17.4, which puts space between rows 5 and 6 of the barley display:

```
barley.plot <- update(barley.plot,
  between = list(y = c(0,0,0,0,1,0,0,0,0)))
barley.plot
```

The argument `between=` is a list with components `x` and `y`; either can be missing. `x` is a vector whose length is equal to the number of columns minus one; the values are the amounts of space, measured in character heights, to be inserted between columns. Similarly, `y` specifies the amounts of space between rows.

17.2 skip=

Figures 17.5 and 17.6 are a display of variables in `market.survey`, a data frame. Each panel has box plots of usage for six age groups. The conditioning variables are, first, seven levels of income and, second, two long distance carriers.



20
30
40
50
60
1931
Wisconsin No. 38
1931
Trebi
1931
Glabron
1931
No. 457
1931
Velvet
1931
Peatland
1931
Manchuria
1931
No. 475
1931
Svansota
1931
No. 462
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
yield

Figure 17.4

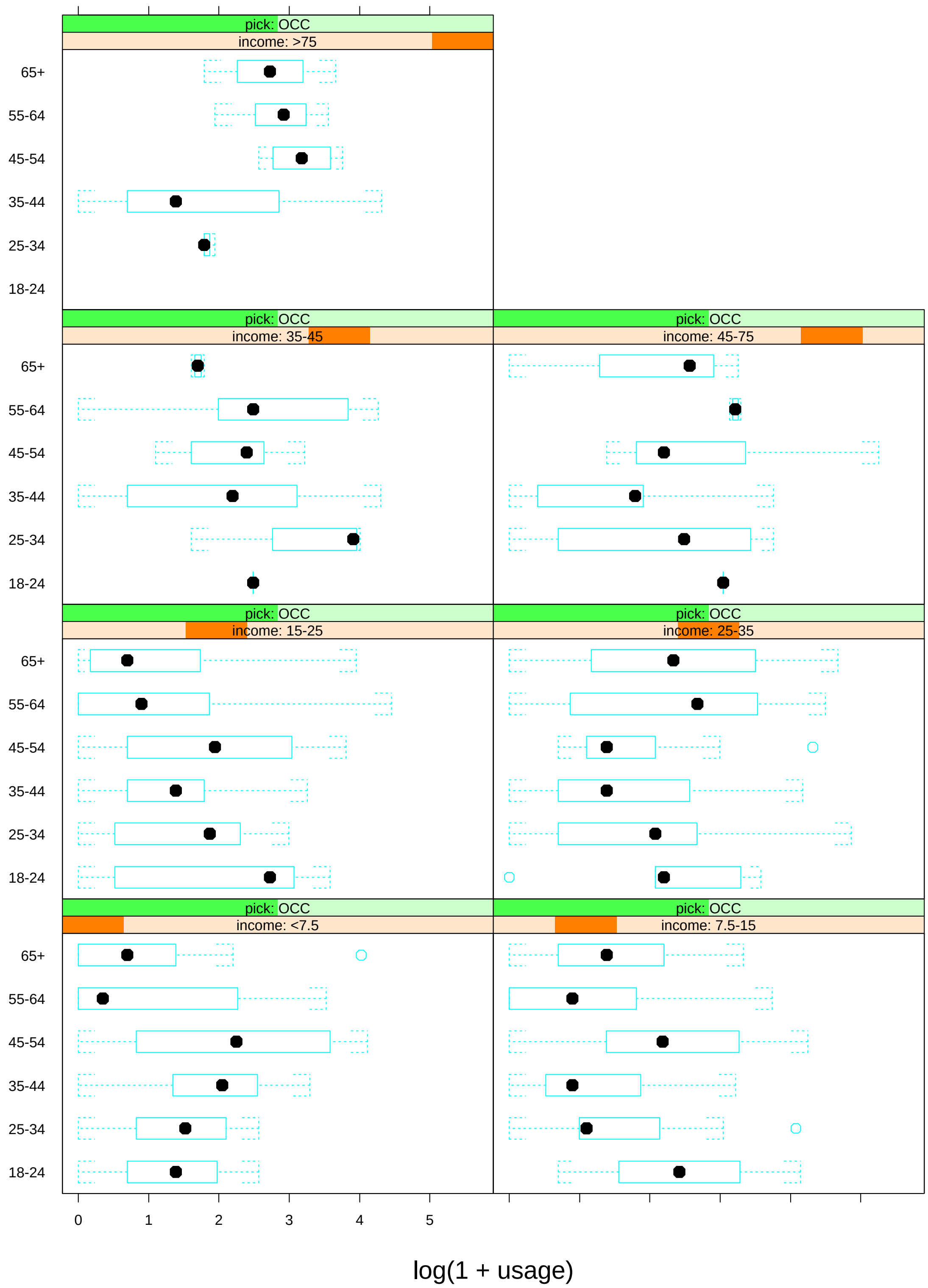

pick: OCC
income: >75
income: 35-45
income: 45-75
income: 15-25
income: 25-35
income: <7.5
income: 7.5-15
65+
55-64
45-54
35-44
25-34
18-24
0
1
2
3
4
5
log(1 + usage)

Figure 17.5

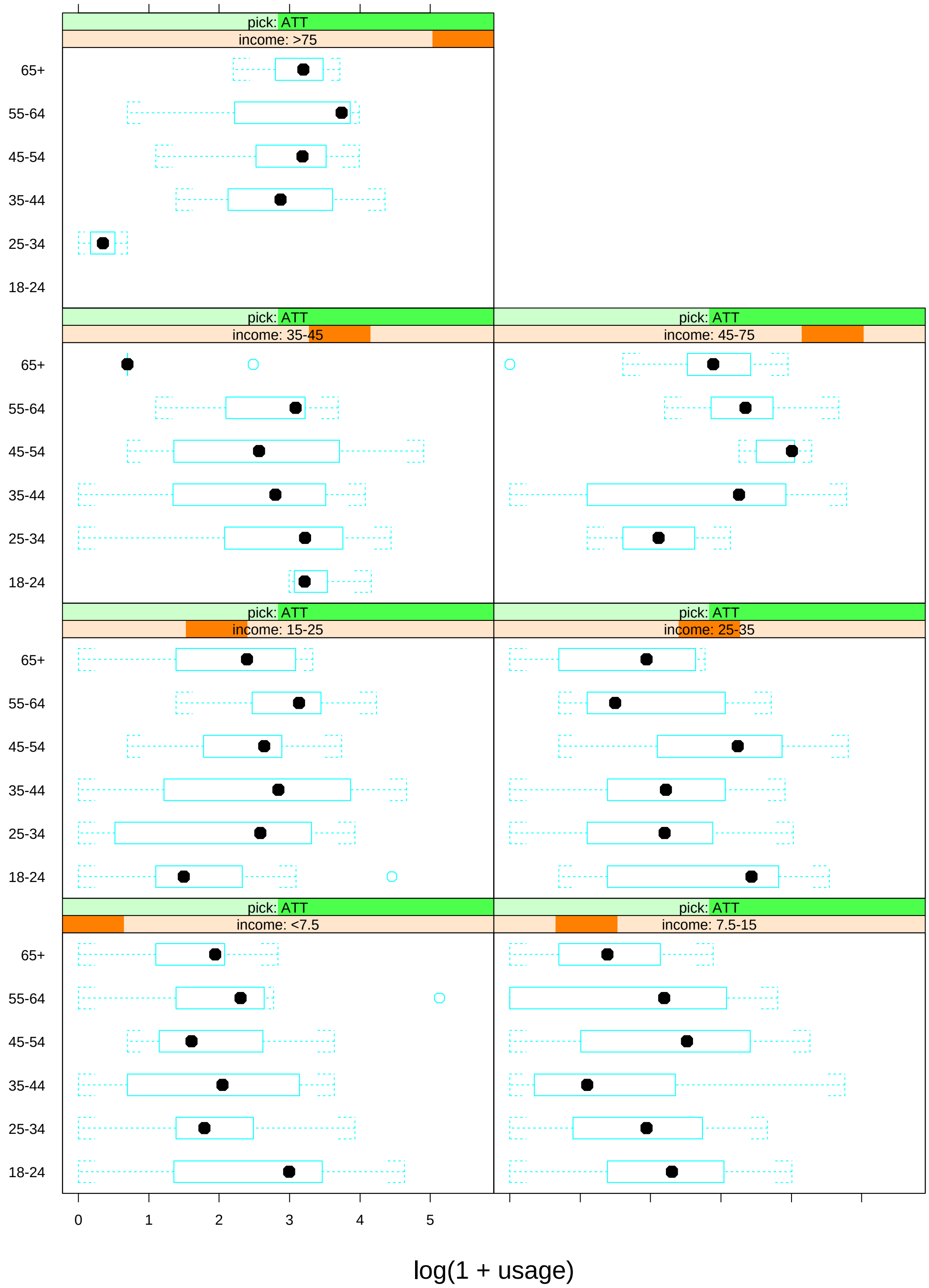

pick: ATT
income: >75
pick: ATT
income: 35-45
pick: ATT
income: 45-75
pick: ATT
income: 15-25
pick: ATT
income: 25-35
pick: ATT
income: <7.5
pick: ATT
income: 7.5-15
65+
55-64
45-54
35-44
25-34
18-24
0
1
2
3
4
5
log(1 + usage)

Figure 17.6

Figures 17.5 and 17.6 were produced by:

```
market.plot <- bwplot(age ~ log(1+usage) | income * pick,
  data = market.survey,
  strip = function(...) strip.default(..., strip.names = T),
  skip = c(F,F,F,F,F,F,F,T),
  layout = c(2,4,2)
  )
market.plot
```

Notice that the layout has eight panels per page but there are seven plots. On both pages, the last panel is skipped. The skipping has been done because the conditioning variable `income` has 7 levels. The argument `skip=`, which takes a logical vector, controls skipping. Each element says whether or not to skip a panel. For Figures 17.5 and 17.6, `skip` is given `c(F,F,F,F,F,F,F,T)`. On the first page, the first seven panels are filled and the eighth is skipped. Since we ran out of elements of `skip=` just as we completed the first page, we went back to the beginning of `skip=` and to determine the skipping for the second page.

17.3 page=

The argument `page=` can add page numbers, text, or graphics to each page of a multipage Trellis display. `page=` should be a function of a single argument `n`, the page number; the function tells what to draw on page `n`. In Figures 17.7 and 17.8, `page=` adds page numbers:

```
update(market.plot,
  page = function(n)
    text(x = .75, y = .95, paste(" page", n), adj = .5))
```

`text()`, an S-PLUS core graphics function, uses a coordinate system that is the same as the panel rectangle coordinate system for the argument `key=`; (0,0) is the lower left corner and (1,1) is the upper left corner.

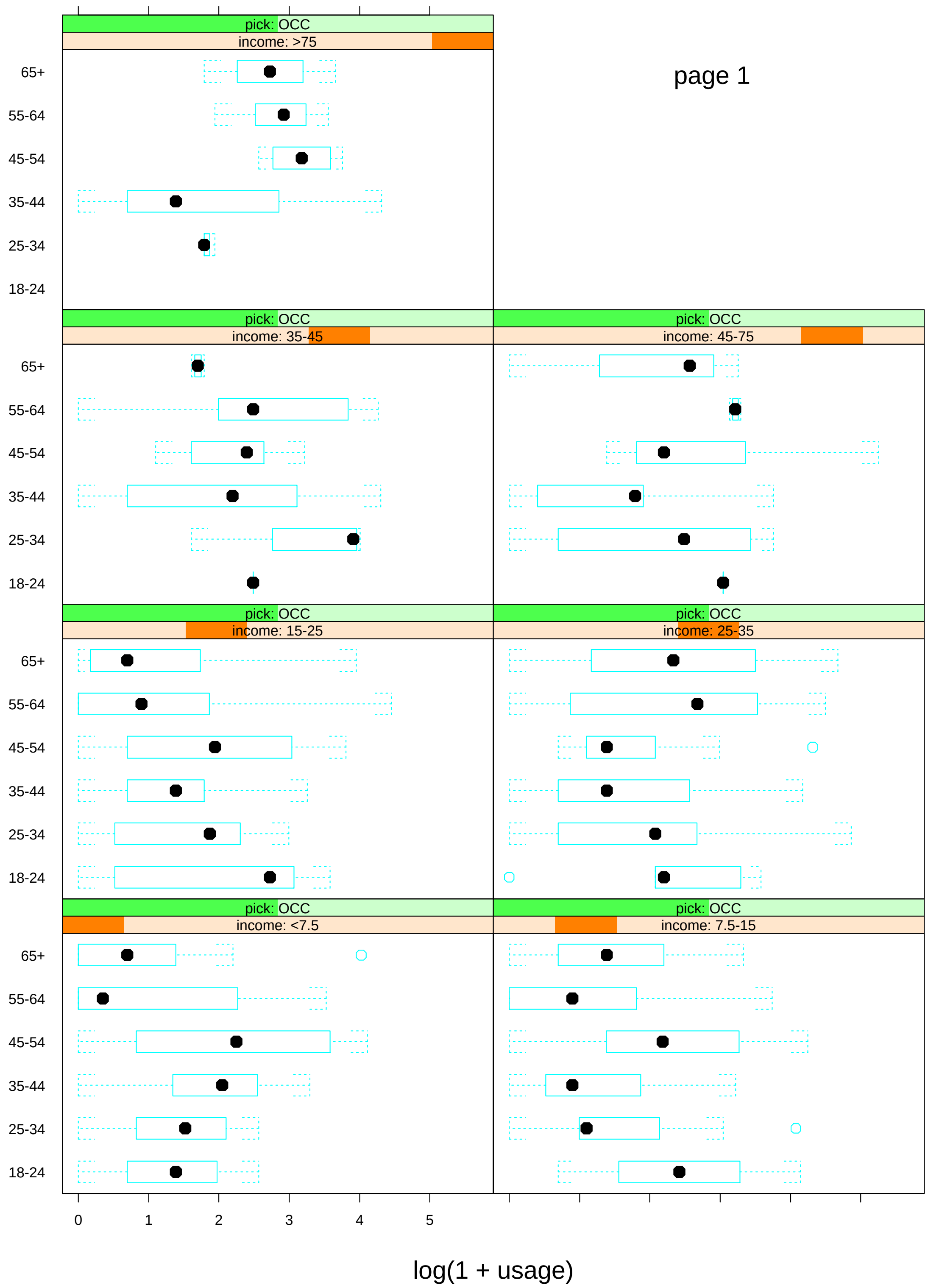

page 1
pick: OCC
income: >75
income: 35-45
income: 45-75
income: 15-25
income: 25-35
income: <7.5
income: 7.5-15
65+
55-64
45-54
35-44
25-34
18-24
0
1
2
3
4
5
log(1 + usage)

Figure 17.7

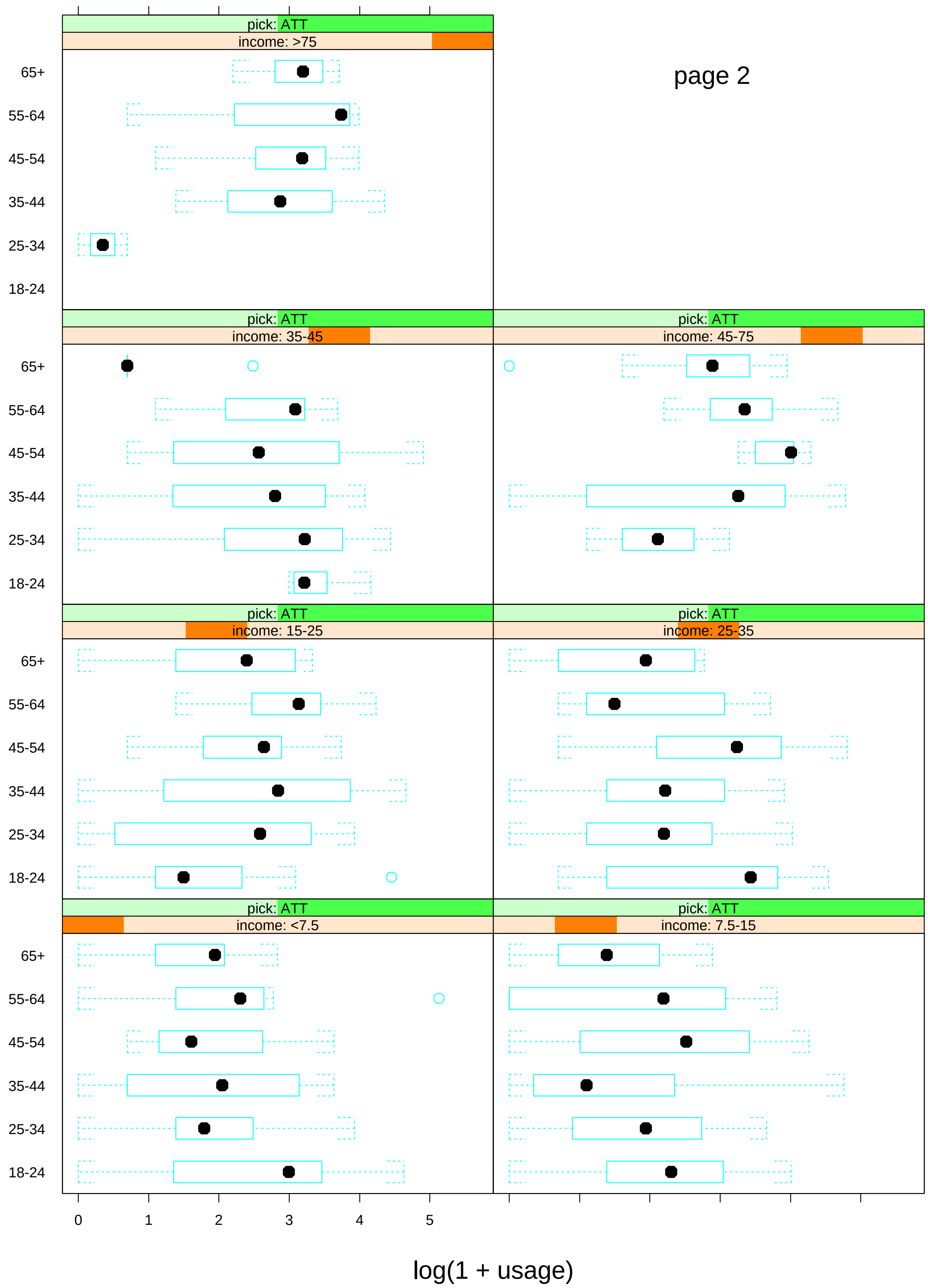

page 2
pick: ATT
income: >75
income: 35-45
income: 45-75
income: 15-25
income: 25-35
income: <7.5
income: 7.5-15
65+
55-64
45-54
35-44
25-34
18-24
0
1
2
3
4
5
log(1 + usage)

Figure 17.8

Chapter 18

More Examples

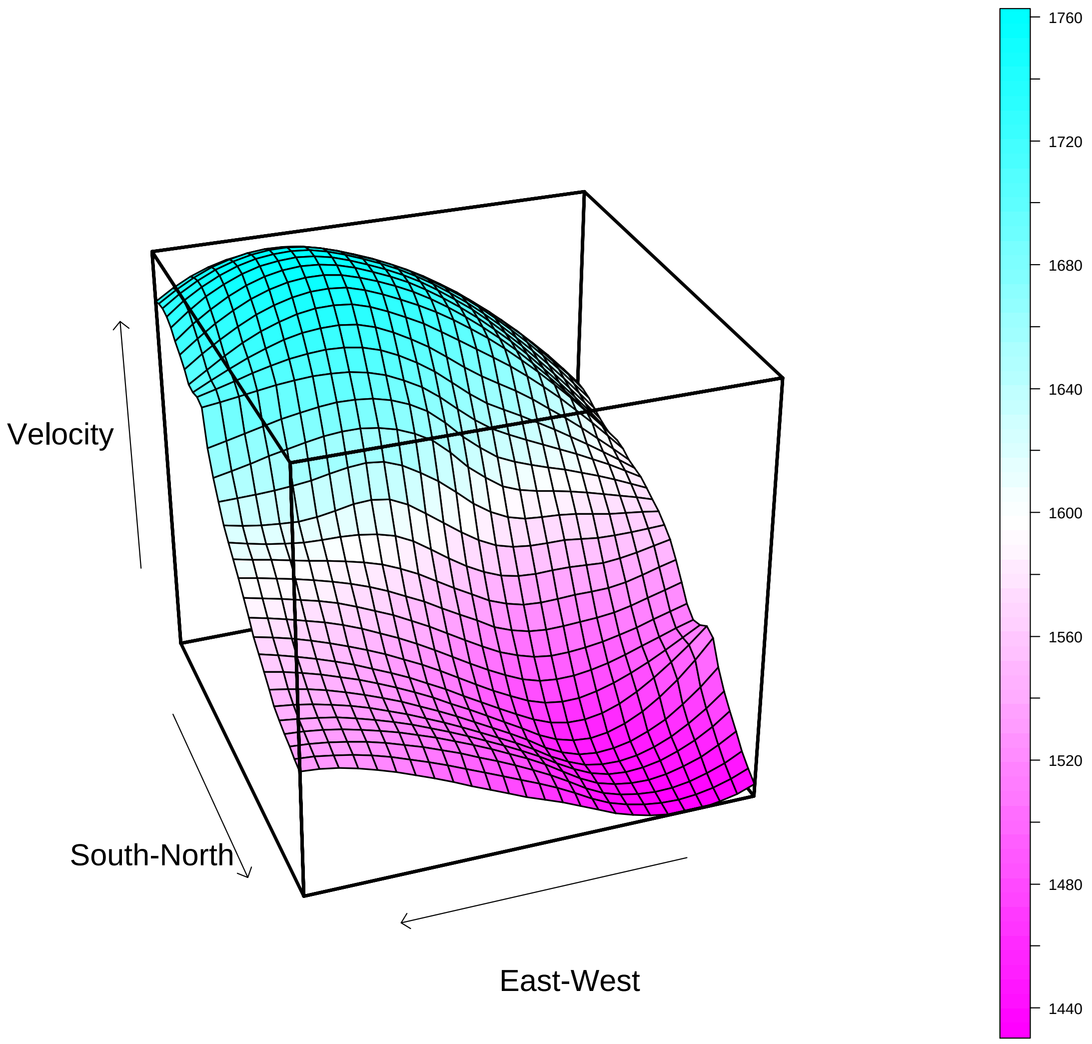
This chapter contains a collection of examples. The displays and the S-PLUS expressions that produce them are given on facing pages. Much can be learned from these examples, which in many cases show advanced usages of Trellis Graphics.

The examples also show how the two displays of chapter 1 are drawn.

The examples use datasets in S-PLUS databases. Any computation that needs to be performed before plotting is given as part of the example. In other words, you can run these examples in S-PLUS.

```
attach(galaxy)
grid <- expand.grid(
  east.west = seq(-25, 25, by = 2),
  north.south = seq( -45, 45, by = 3))
fit <- c(predict(loess(velocity ~ east.west * north.south,
  span = 0.25, degree = 2, normalize = F,
  family = "symmetric"), grid))
detach()

wireframe(fit ~ grid$east.west * grid$north.south,
  screen = list(z = 200, x = -60, y = 0),
  lwd = 1.5,
  par.box = list(lwd = 3, lty = 1, col = 1),
  colorkey = list(skip = c(F, T), tick.number = 17),
  drape = T,
  distance = 0.3,
  xlab = list("East-West", cex = 1),
  ylab = list("South-North", cex = 1),
  zlab = list("Velocity", cex = 1))
```



Velocity
South-North
East-West
1760
1720
1680
1640
1600
1560
1520
1480
1440

Figure 18.1

```
attach(environmental)
ozo.m <- loess(
  (ozone^(1/3)) ~ wind * temperature * radiation,
  parametric = c("radiation", "wind"), span = 1, degree = 2
  )
w.marginal <- seq(min(wind), max(wind),
  length = 50)
t.marginal <- seq(min(temperature), max(temperature),
  length = 50)
r.marginal <- seq(min(radiation), max(radiation),
  length = 4)
wtr.marginal <- list(
  wind = w.marginal,
  temperature = t.marginal,
  radiation = r.marginal)
grid <- expand.grid(wtr.marginal)
grid[, "fit"] <- c(predict(ozo.m, grid))
detach()

levelplot(fit ~ wind * temperature | radiation,
  data = grid,
  cuts = 11,
  pretty = T,
  contour = T,
  labels = F,
  lwd = 6,
  col = 1,
  scale = list(cex = 0.7),
  xlab = "Wind Speed (mph)",
  ylab = "Temperature (F)")
```

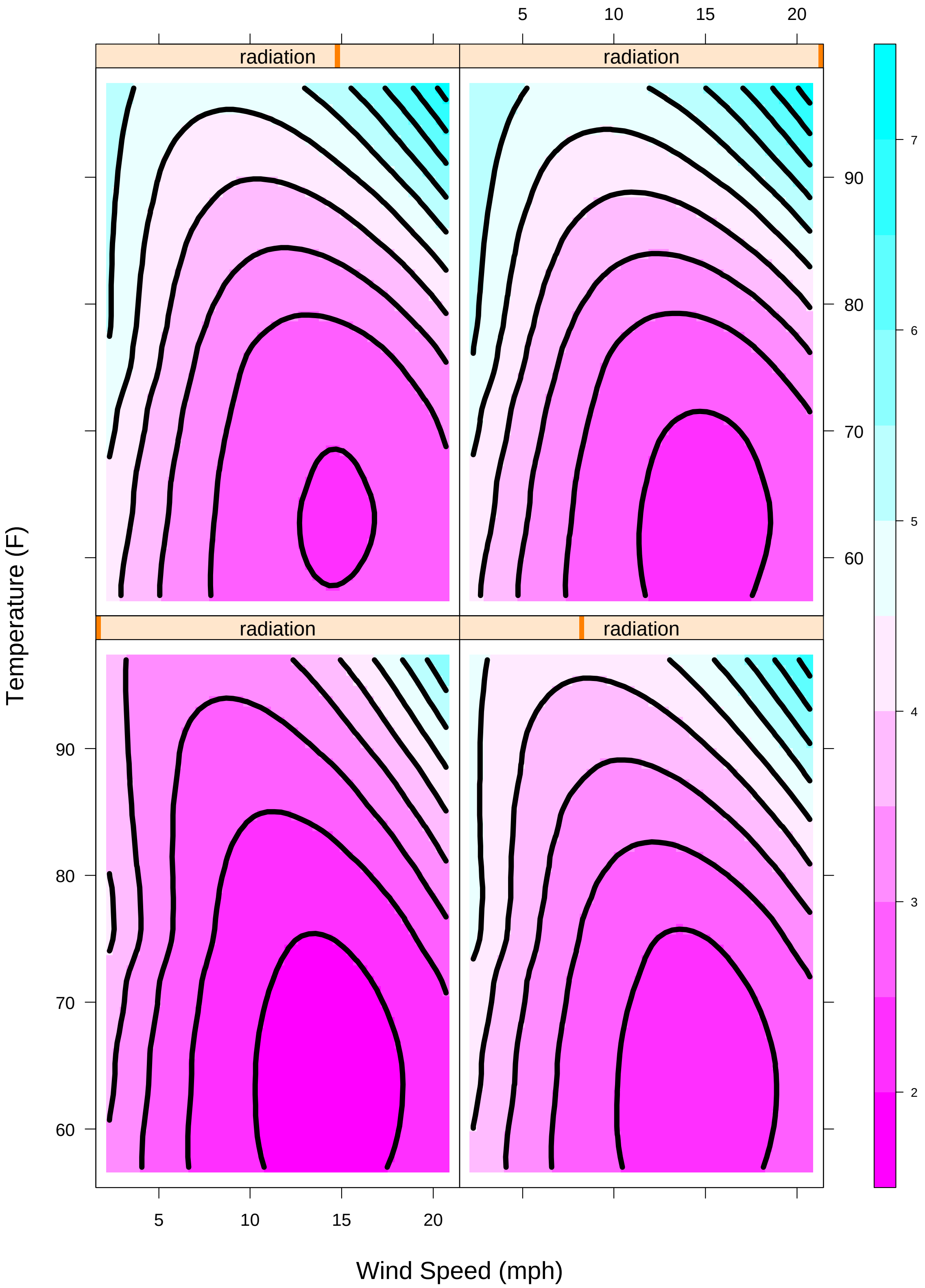

radiation
radiation
radiation
radiation
5
10
15
20
90
80
70
60
7
6
5
4
3
2
Temperature (F)
Wind Speed (mph)

Figure 18.2

```
dotplot(variety ~ yield | site,
  data = barley,
  groups = year,
  panel = function(x, y, ...) {
    dot.line <- trellis.par.get("dot.line")
    abline(h = unique(y), lwd = dot.line$lwd,
      lty = dot.line$lty, col = dot.line$col)
    panel.superpose(x, y, ...)
    },
  scale = list(y = list(cex = .7)),
  layout = c(1, 6),
  par.strip = list(cex = .75),
  aspect = .5,
  xlab = list("Barley Yield (bushels/acre)", cex = 1),
  key = list(
    y = 1.02,
    points = Rows(trellis.par.get("superpose.symbol"), 1:2),
    text = list(levels(barley$year)),
    columns = 2
    )
  )
```

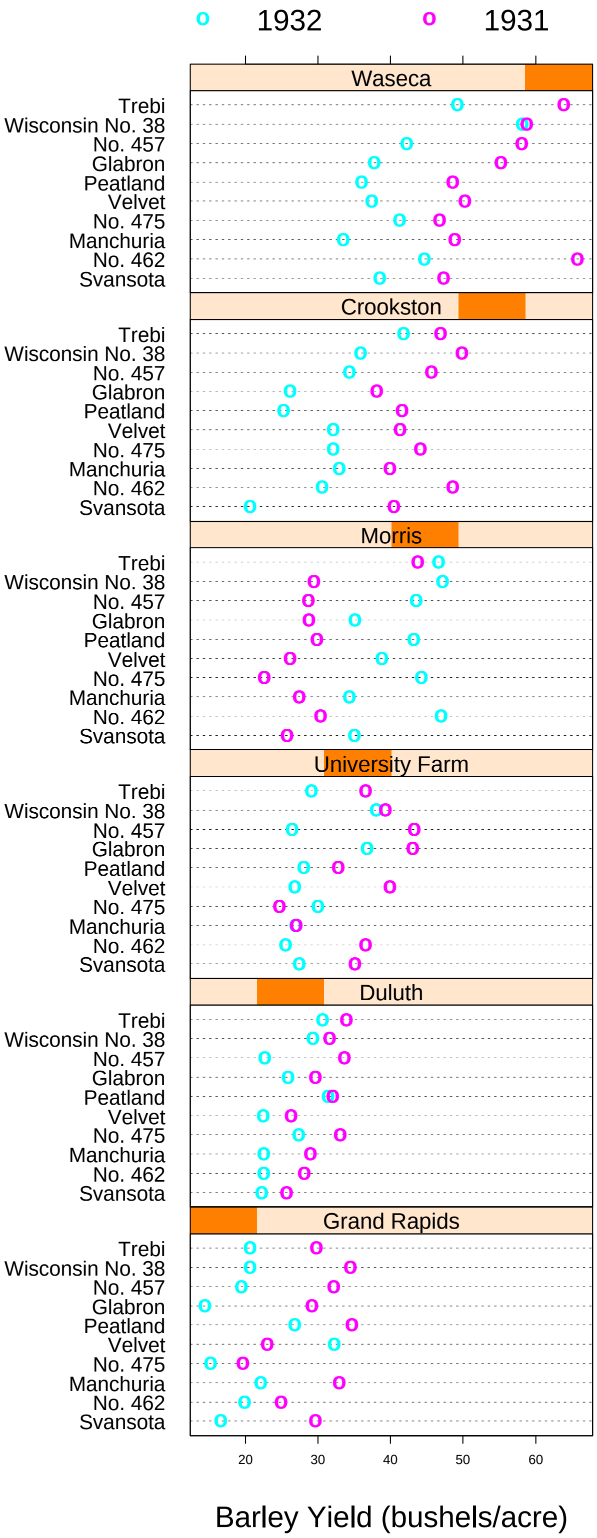

1932
1931
Waseca
Crookston
Morris
University Farm
Duluth
Grand Rapids
Trebi
Wisconsin No. 38
No. 457
Glabron
Peatland
Velvet
No. 475
Manchuria
No. 462
Svansota
20
30
40
50
60
Barley Yield (bushels/acre)

Figure 18.3

```
attach(environmental)
Temp <- equal.count(temperature, 4, 1/2)
Wind <- equal.count(wind, 4, 1/2)

xyplot((ozone^(1/3)) ~ radiation | Temp * Wind,
  prepanel = function(x, y)
    prepanel.loess(x, y, span = 1),
  panel = function(x, y){
    panel.grid(h = 2, v = 2, lwd = .5)
    panel.xyplot(x, y, cex = 0.6)
    panel.loess(x, y, span = 1)
    },
  par.strip = list(cex = .75),
  aspect = 2,
  xlab = list("Solar Radiation (langleys)", cex = 1),
  ylab = list("Cube Root Ozone (cube root ppb)", cex = 1)
  )

detach()
```

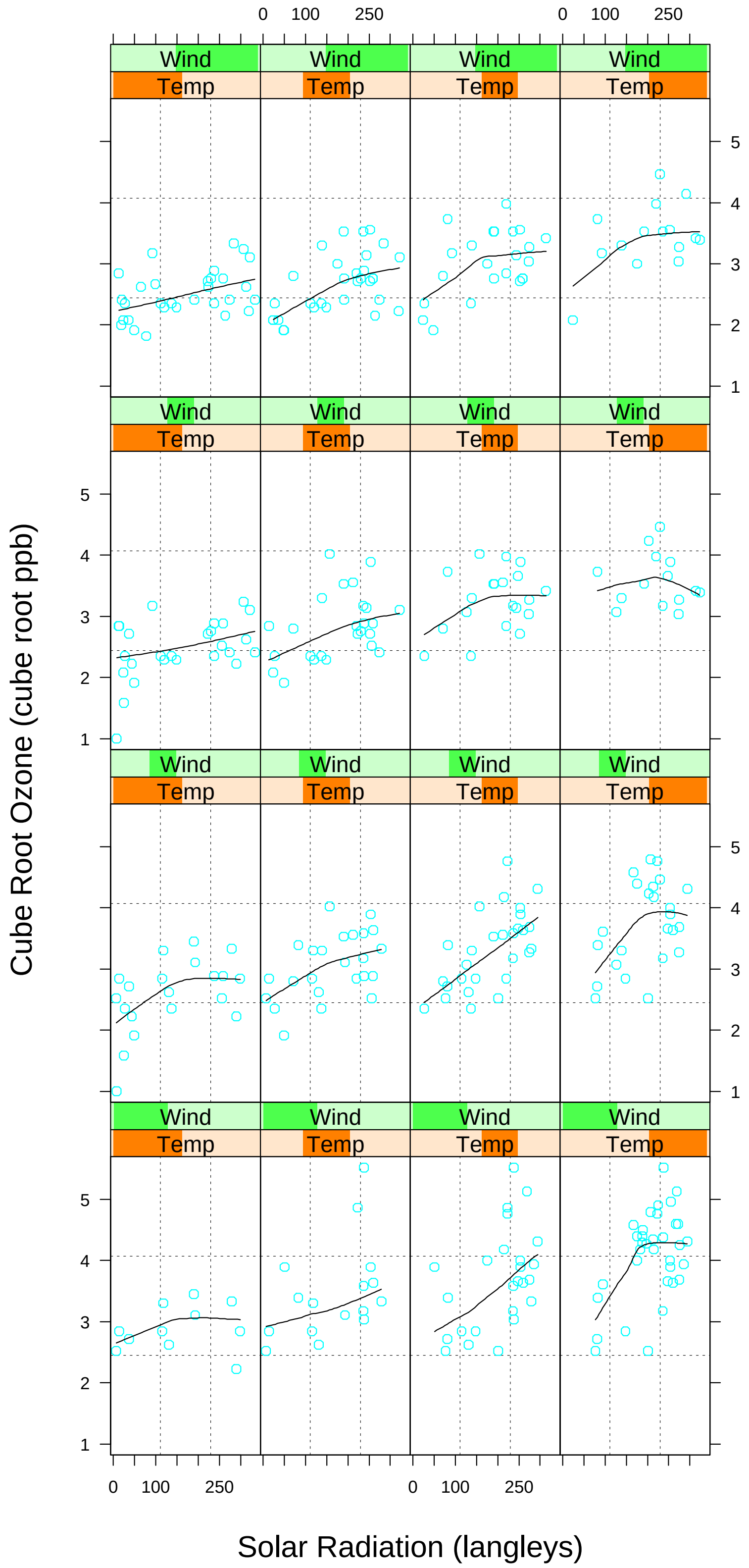

Wind
Temp
Cube Root Ozone (cube root ppb)
Solar Radiation (langleys)
0
100
250
1
2
3
4
5

Figure 18.4

```
iris.df <- as.data.frame.array(iris, col.dims = 2)
names(iris.df)[5:6] <- c("flower", "variety")
for (i in 1:4)
  iris.df[,i] <- jitter(iris.df[,i])

splom( ~ iris.df[,1:4] | iris.df[,"variety"],
  cex = .2,
  varnames = c("SL", "SW", "PL", "PW"),
  page = function(...)
    text(seq(.6, .8, length = 4), seq(.9, .6, length = 4),
      c("Three", "Varieties", "of", "Iris"),
      adj = 0, cex = 1.5
    )
  )
```

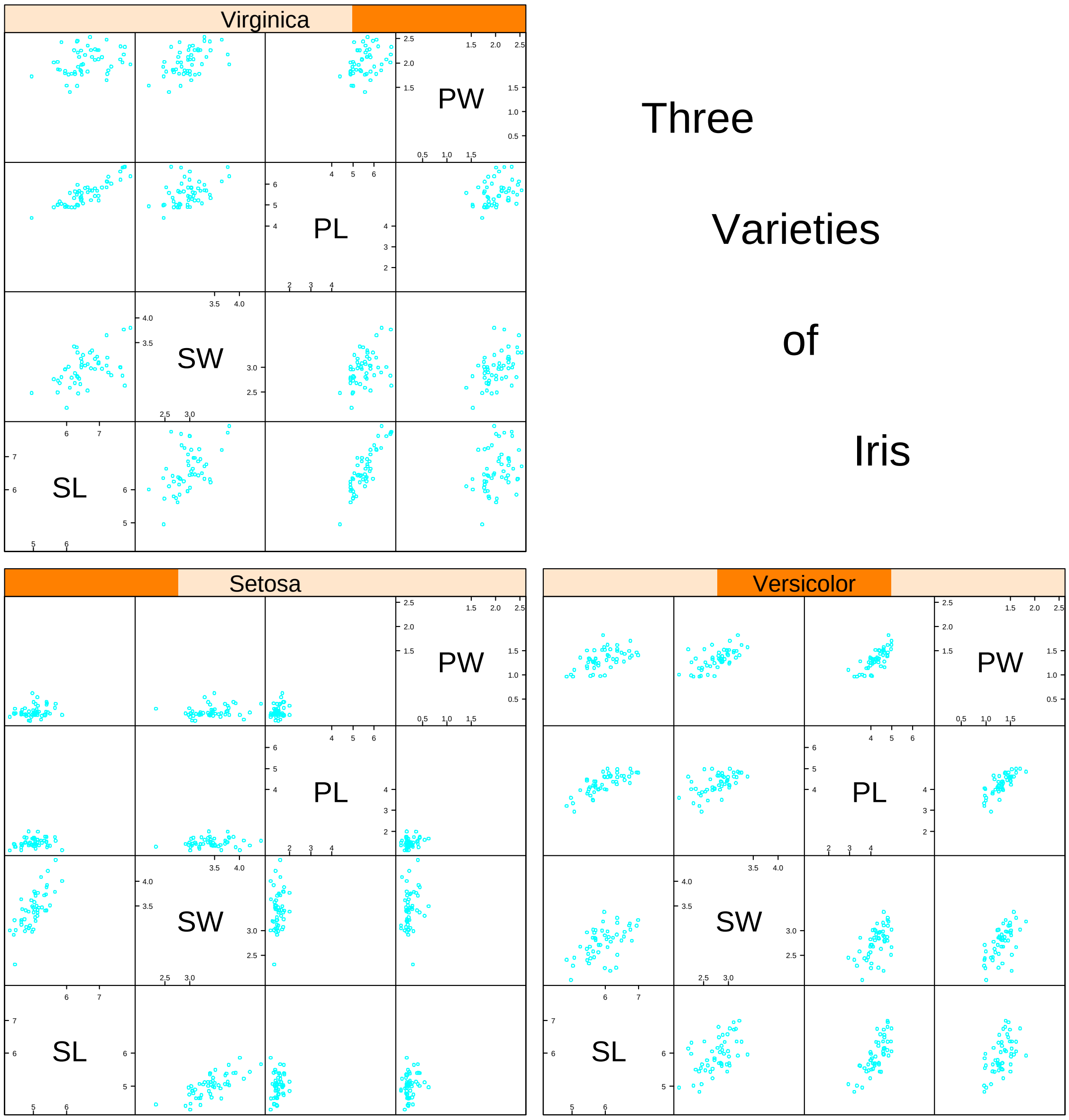
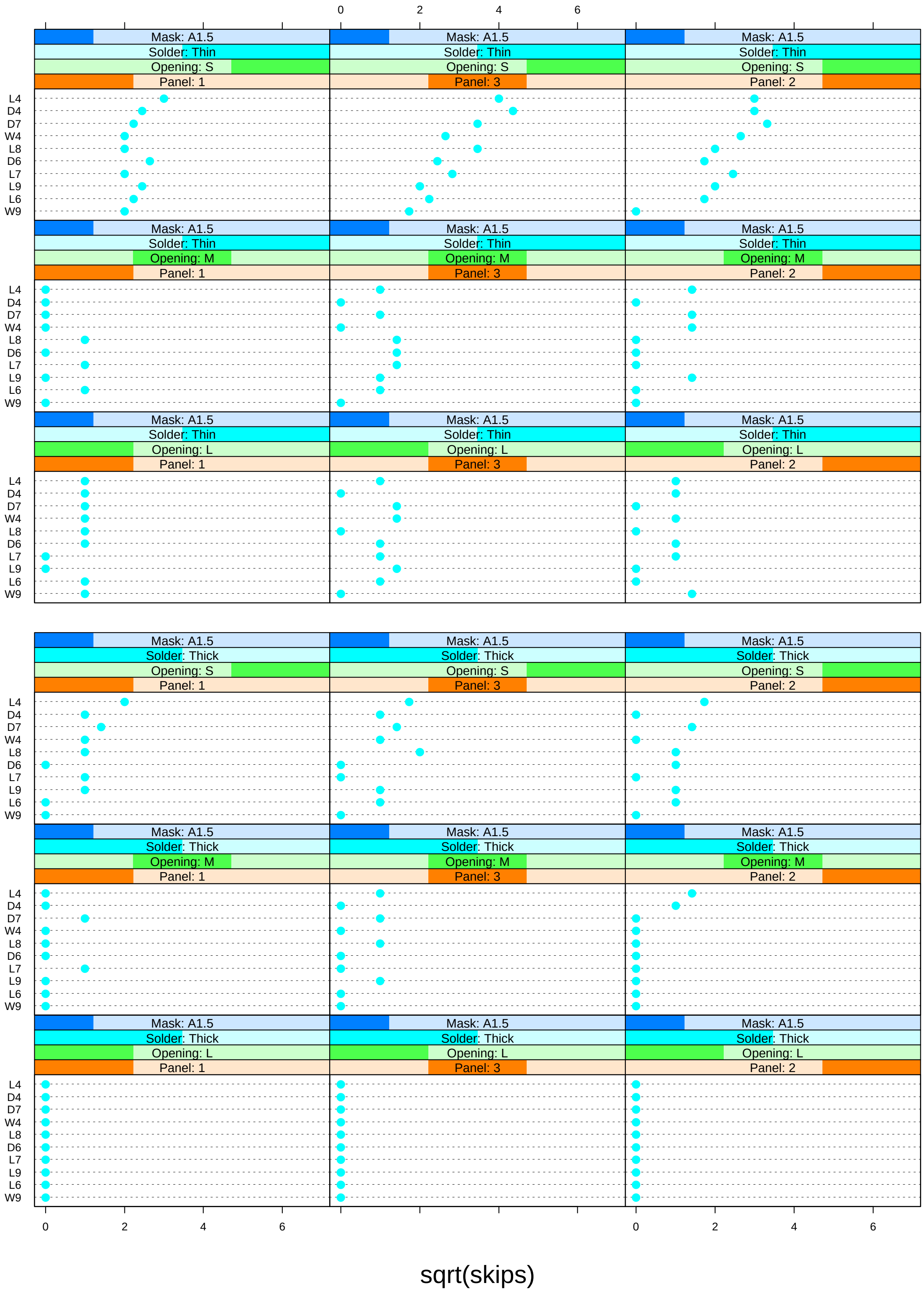

Virginica
Setosa
Versicolor
PW
PL
SW
SL
Three
Varieties
of
Iris

Figure 18.5

```
new.solder <- solder
for (i in 1:5)
  new.solder[,i] <- reorder.factor(new.solder[,i],
    new.solder[,6])

dotplot(PadType ~ sqrt(skips) | Panel*Opening*Solder*Mask,
  data = new.solder,
  strip = function(...) strip.default(..., strip.names = T),
  between = list(y = c(0,0,1,0,0)),
  layout = c(3,6,5))
```



Mask: A1.5
Solder: Thin
Solder: Thick
Opening: S
Opening: M
Opening: L
Panel: 1
Panel: 2
Panel: 3
L4
D4
D7
W4
L8
D6
L7
L9
L6
W9
0
2
4
6
sqrt(skips)

Figure 18.6

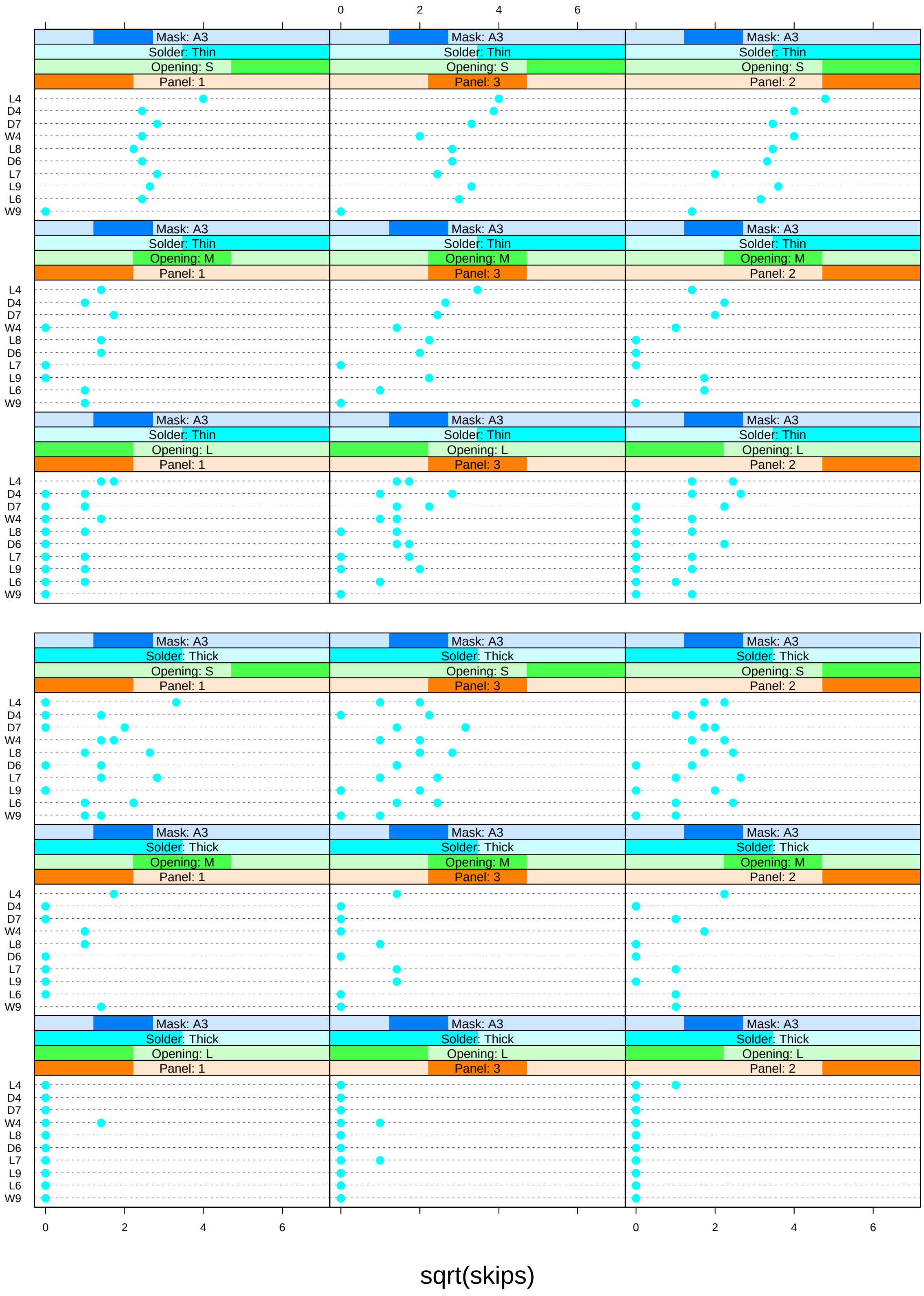

Mask: A3
Solder: Thin
Solder: Thick
Opening: S
Opening: M
Opening: L
Panel: 1
Panel: 2
Panel: 3
L4
D4
D7
W4
L8
D6
L7
L9
L6
W9
0
2
4
6
sqrt(skips)

Figure 18.7

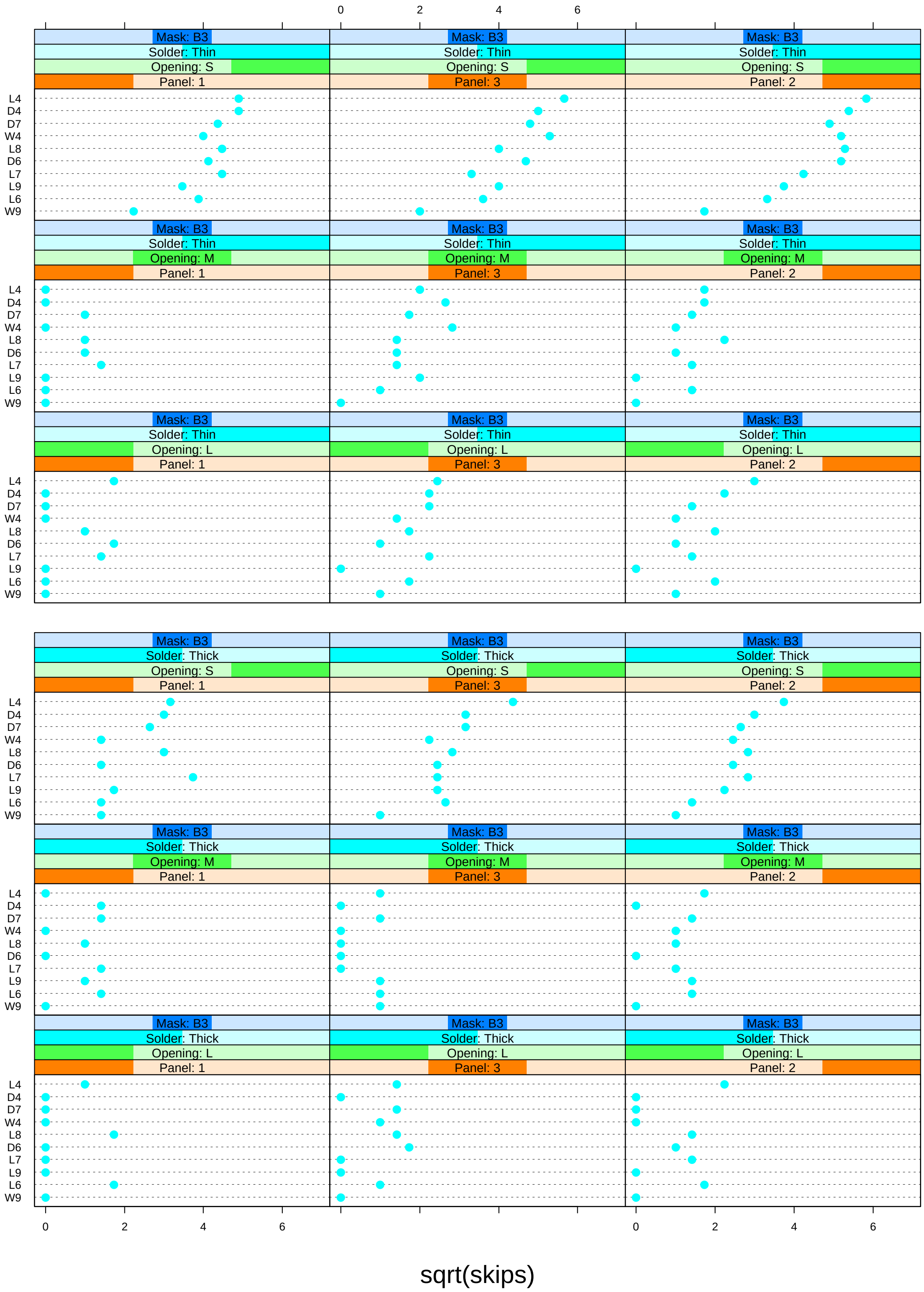

Mask: B3
Solder: Thin
Solder: Thick
Opening: S
Opening: M
Opening: L
Panel: 1
Panel: 2
Panel: 3
L4
D4
D7
W4
L8
D6
L7
L9
L6
W9
0
2
4
6
sqrt(skips)

Figure 18.8

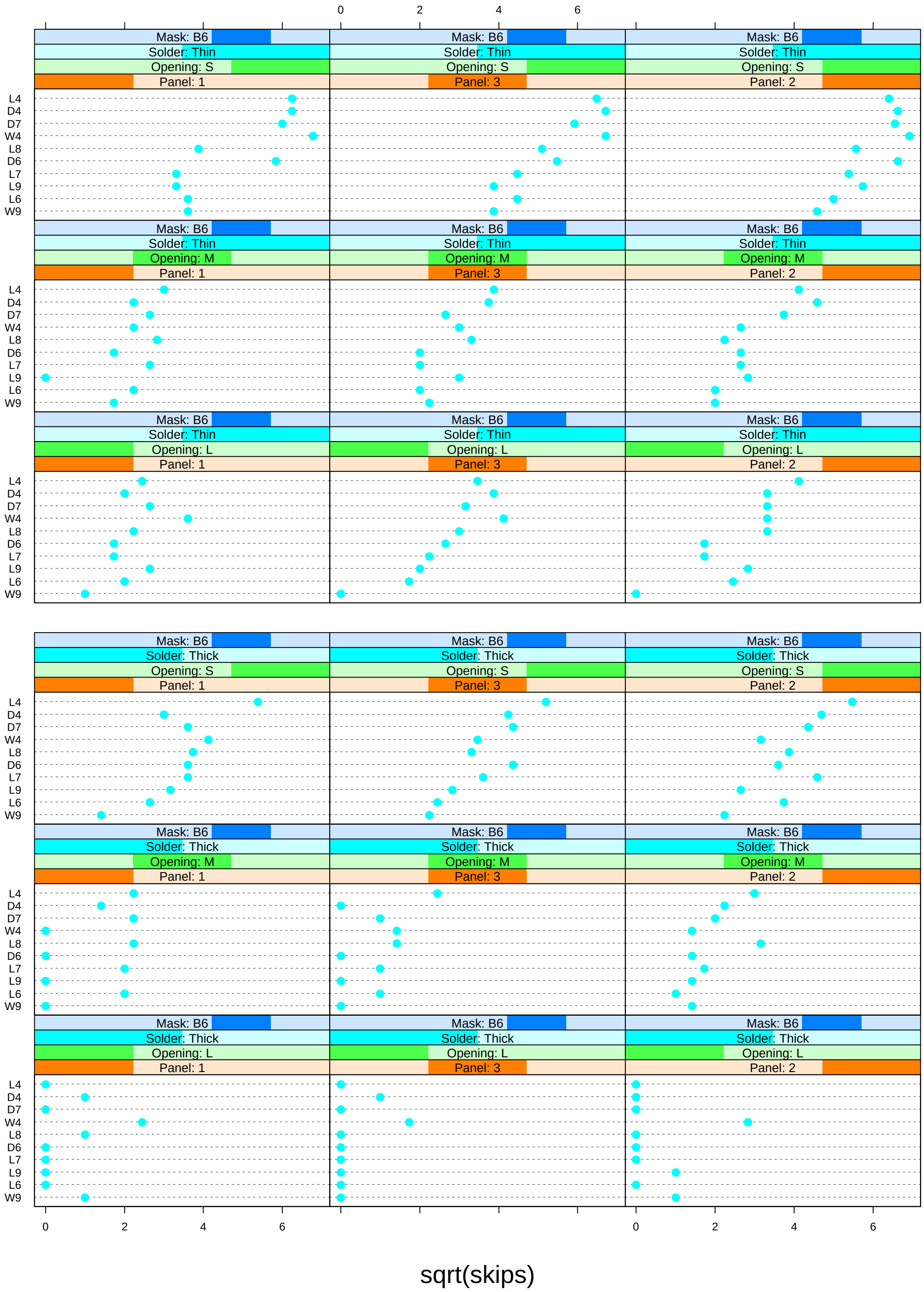

Mask: B6
Solder: Thin
Solder: Thick
Opening: S
Opening: M
Opening: L
Panel: 1
Panel: 2
Panel: 3
L4
D4
D7
W4
L8
D6
L7
L9
L6
W9
0
2
4
6
sqrt(skips)

Figure 18.9

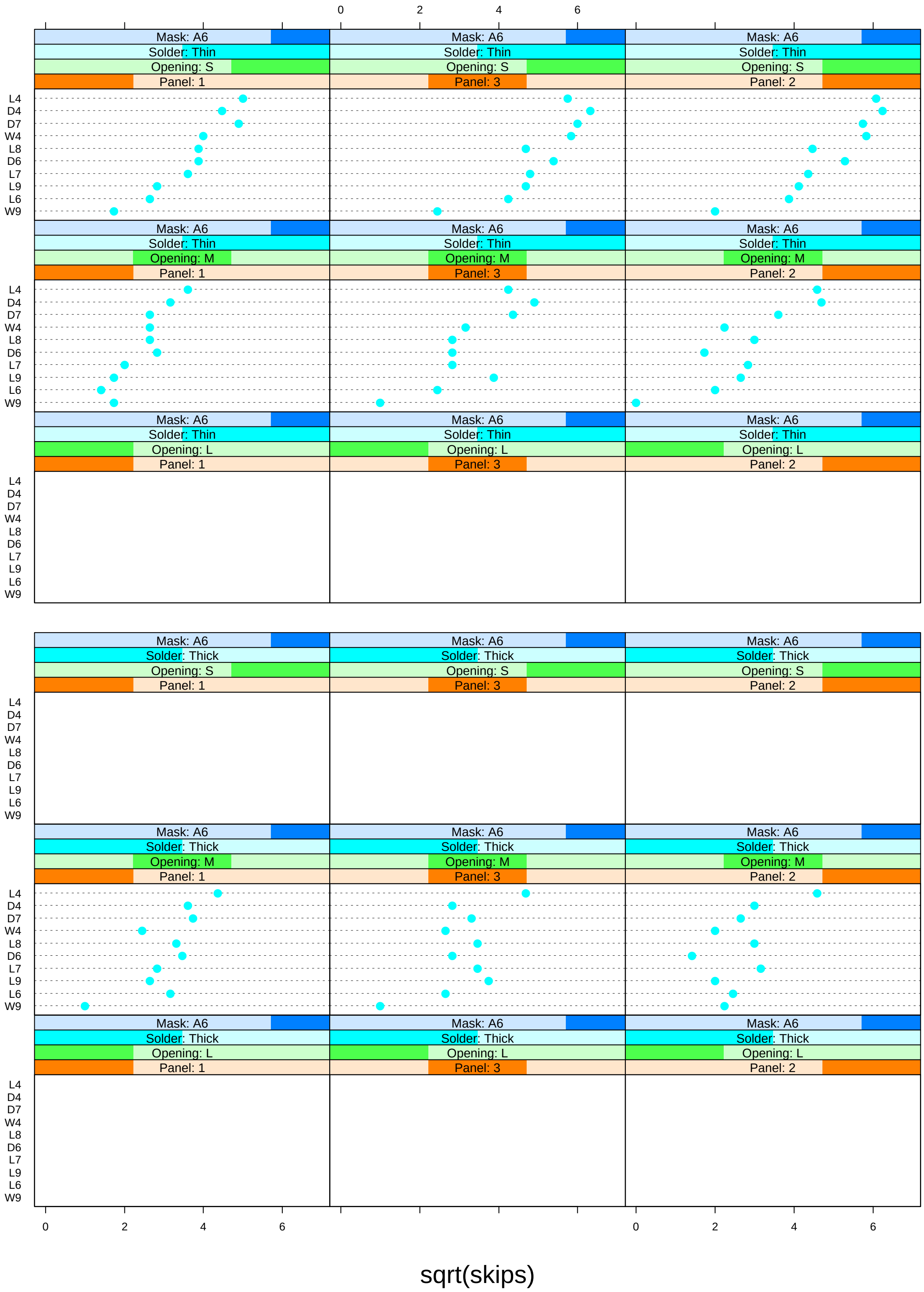

Mask: A6
Solder: Thin
Solder: Thick
Opening: S
Opening: M
Opening: L
Panel: 1
Panel: 2
Panel: 3
L4
D4
D7
W4
L8
D6
L7
L9
L6
W9
0
2
4
6
sqrt(skips)

Figure 18.10

```
attach(barley)
morris31 <- yield[(site=="Morris")&(year=="1931")]
morris32 <- yield[(site=="Morris")&(year=="1932")]
new.yield <- yield
new.yield[(site=="Morris")&(year=="1931")] <- morris32
new.yield[(site=="Morris")&(year=="1932")] <- morris31
wt <- rep(1, length(yield))
for(i in 1:10){
  barley.lm <- lm(new.yield~variety+year*site, weights = wt)
  wt <- wt.bisquare(barley.lm$res/median(abs(barley.lm$res)),
    c = 6)
  }
detach()

rfs(barley.lm,
  scale = list(cex = .8),
  par.strip.text = list(cex = 1),
  aspect = 2,
  ylab = list("Yield (bushels/acre)", cex = 1.25),
  xlab = list("f-value", cex = 1.25))
```

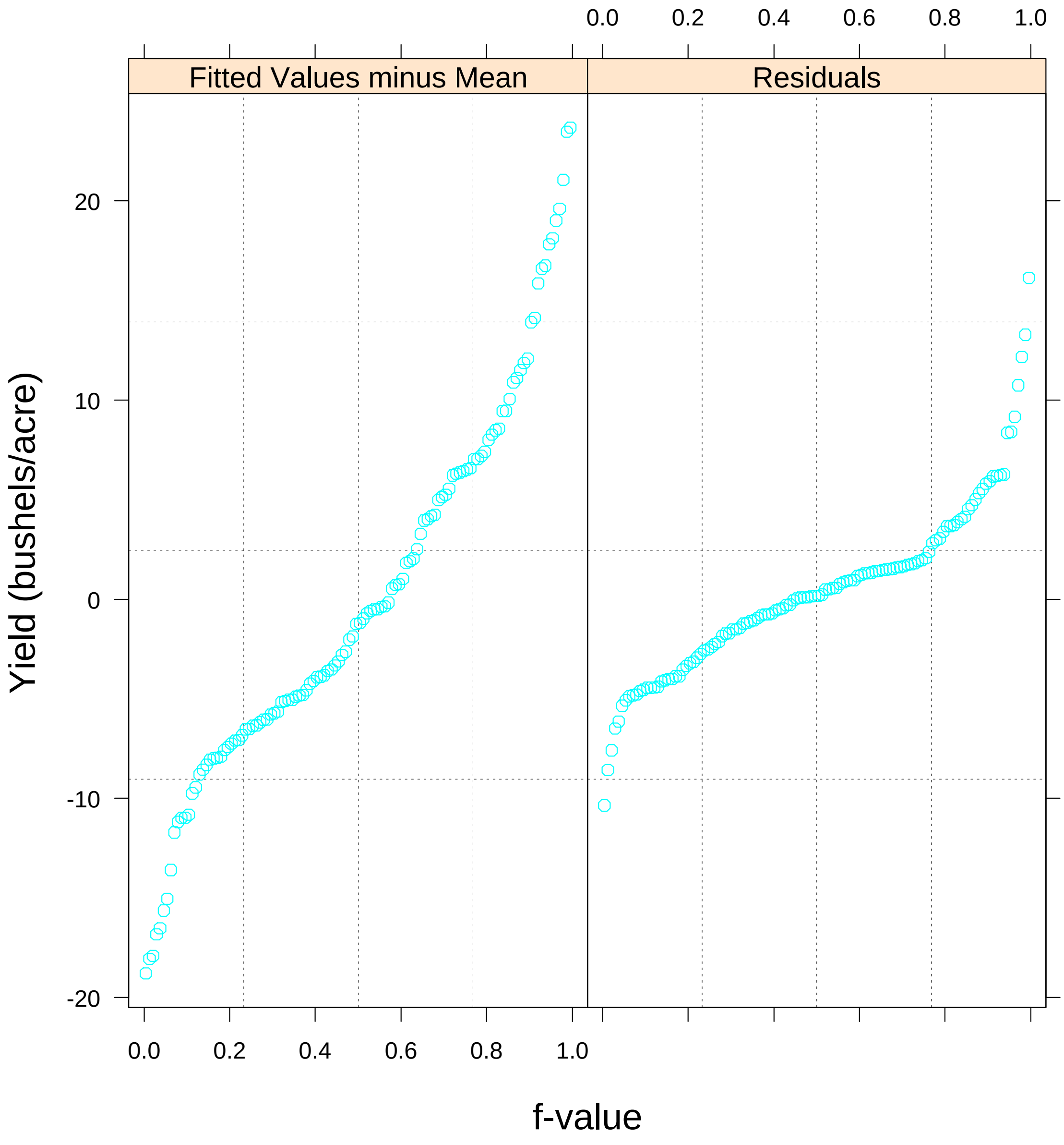
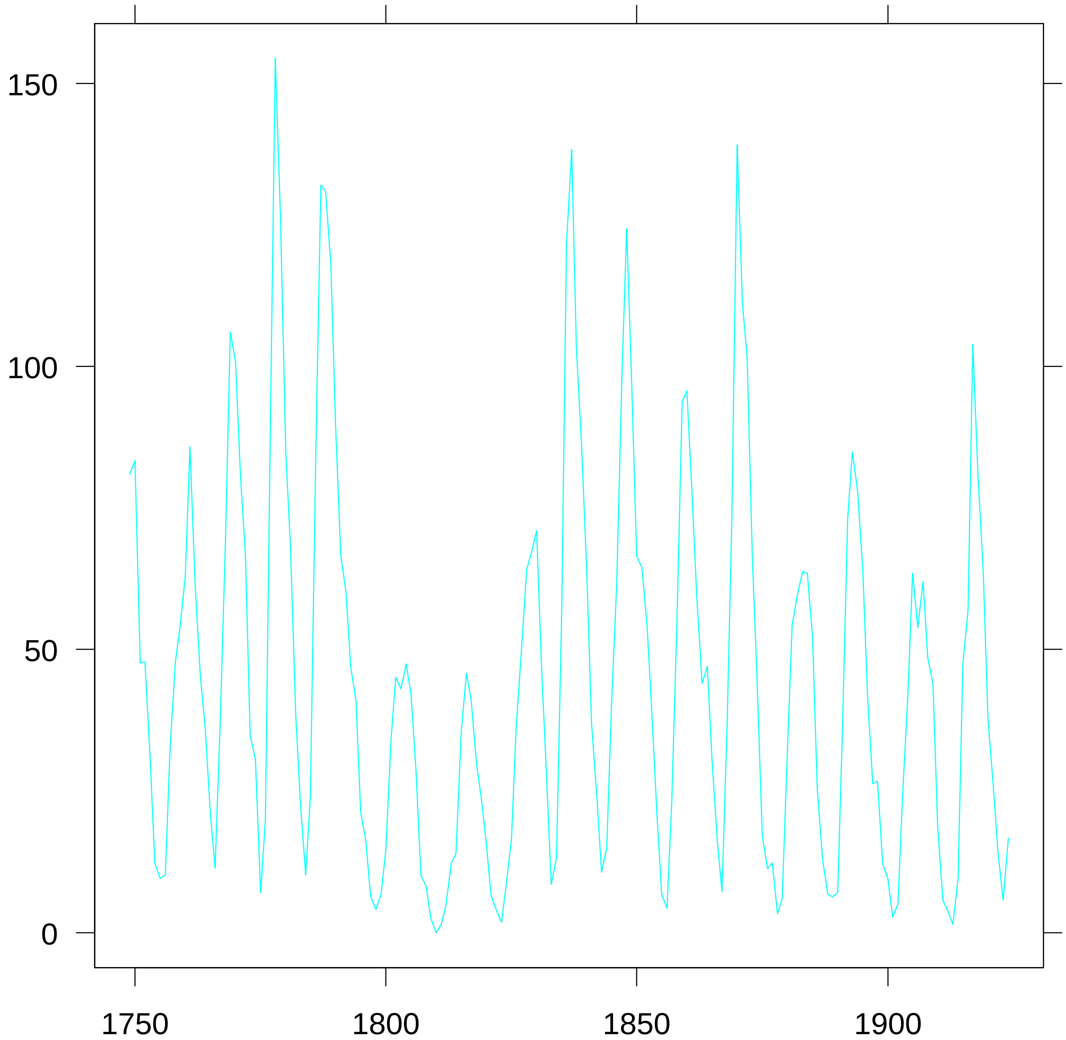

Fitted Values minus Mean
Residuals
Yield (bushels/acre)
f-value
20
10
0
-10
-20
0.0
0.2
0.4
0.6
0.8
1.0

Figure 18.11

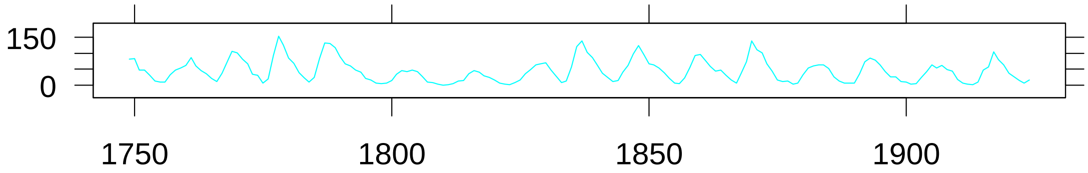
```
wolfer <- window(sunspots, end = c(1924,12))
wolfer <- ts(tapply(wolfer, trunc(time(wolfer)),mean),
  start = 1749)

sun1.0 <- xyplot(wolfer~time(wolfer),
  type = "l",
  aspect = 1.0,
  ylab = "",
  xlab = "Sunspot Number vs. Year")
sunxy <- xyplot(wolfer~time(wolfer),
  type = "l",
  aspect = "xy",
  ylab = "",
  xlab = "Sunspot Number vs. Year",
  ylim = range(wolfer)+c(-1,1)*diff(range(wolfer))*.2)

print(sun1.0, position = c(0,.25,1,1), more = T)
print(sunxy, position = c(0,0,1,.3))
```



Sunspot Number vs. Year



Sunspot Number vs. Year

Figure 18.12

```
attach(galaxy)
grid <- expand.grid(
  east.west = seq(-25, 25, by = 2),
  north.south = seq( -45, 45, by = 3))
fit <- c(predict(loess(velocity ~ east.west * north.south,
  span = 0.25, degree = 2, normalize = F,
  family = "symmetric"), grid))
detach()

angle <- c(22.5, 67.5, 112.5, 337.5, 157.5, 292.5, 247.5, 202.5)
Angle <- shingle(rep(angle, rep(length(fit), 8)), angle)

wireframe( rep(fit, 8) ~ rep(grid$east.west, 8) *
  rep(grid$ north.south, 8) | Angle,
  groups = Angle,
  panel = function(x, y, subscripts, z, groups, ...) {
    w <- groups[subscripts][1]
    panel.wireframe(x, y, subscripts, z,
      screen = list(z = w, x = -60, y = 0), ...)
    },
  strip = FALSE,
  skip = c(F, F, F, F, T, F, F, F, F),
  layout = c(3,3),
  distance = .3,
  xlab = "E-W",
  ylab = "S-N",
  zlab = "V"
  )
```

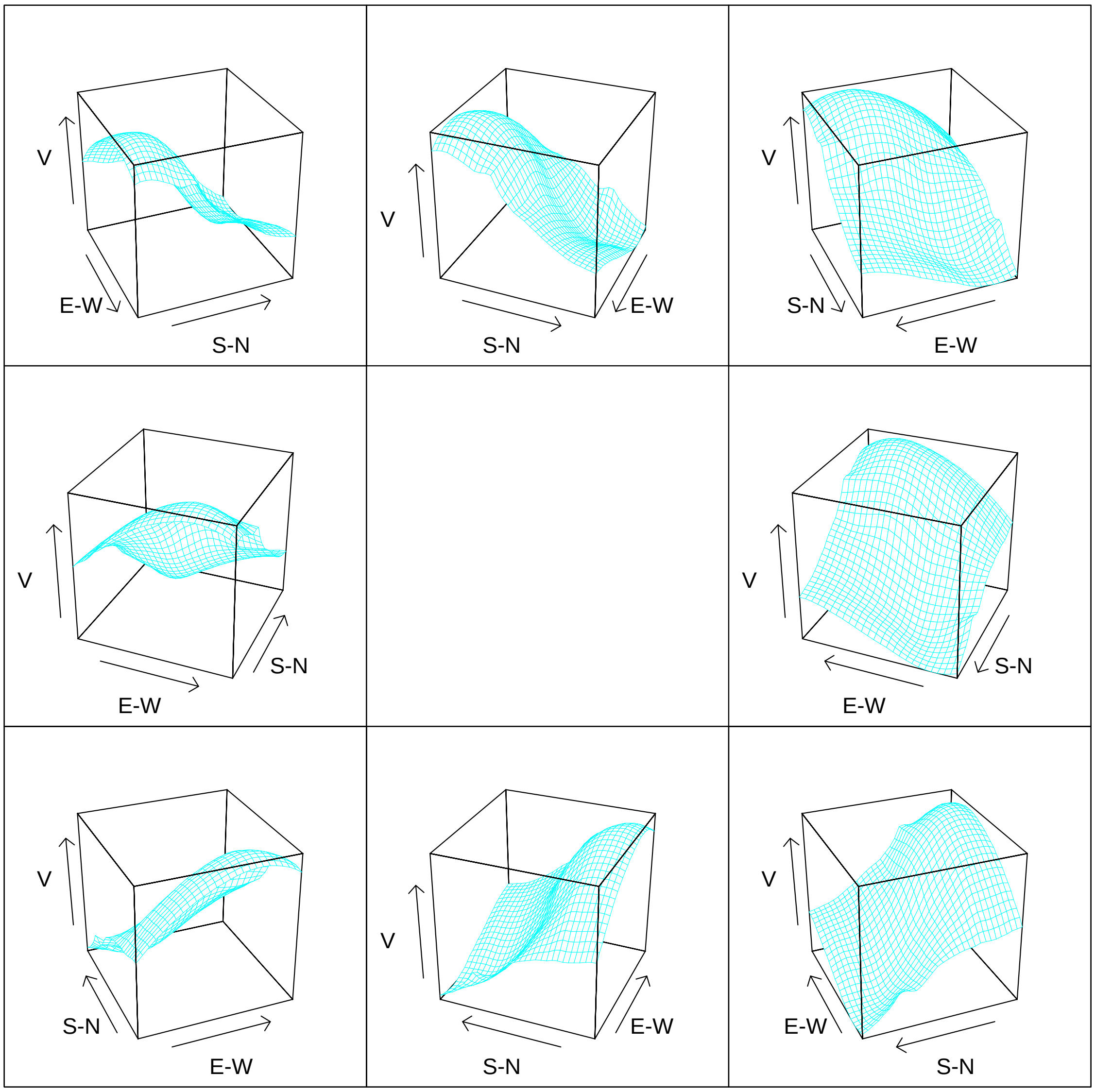


Figure 18.13