2

A Technical Overview of lattice

This chapter gives a broad overview of lattice, briefly describing the most important features shared by all high-level functions. Some of the topics covered are somewhat technical, but they are important motifs in the "big picture" view of lattice, and it would hinder rather than help to introduce them later at arbitrary points in the book. For readers that are new to lattice, it is recommended that they give this chapter a cursory overview and move on to the subsequent chapters. Each of the remaining chapters in Part I can be read, for the most part, directly after Chapter 1, although some advanced examples do require some groundwork laid out in this chapter. This nonlinear flow is inconvenient for those new to lattice, but it is somewhat inevitable; one should not expect to learn all the complexities of Trellis graphics in a first reading.

2.1 Basic usage

Strictly speaking, all high-level functions in lattice are generic functions, and suitable methods can be written for particular classes. In layman's terms, this means that the code that gets executed when a user calls such a function (e.g., `dotplot()`) will depend on the arguments supplied to the function. In practice, most such methods are simple wrappers to the *"formula"* method (i.e., the function that gets executed when the first argument is a *"formula"* object), because it allows for the most flexible specification of the structure of the display. Other methods can be valuable, as we see in later chapters. In this chapter, we restrict our attention to the *"formula"* methods.

2.1.1 The Trellis formula

The use of formulae is the standard when it comes to specifying statistical models in the S language, and they are the primary means of defining the structure of a lattice display as well. A typical Trellis formula looks like

```
y ~ x | a * b
```

The tilde (`~`) is what makes it a *"formula"* object, and is essential in any Trellis formula. Equally important is the vertical bar (`|`), which denotes conditioning. Variables (or more precisely, terms) to the right of the conditioning symbol are called *conditioning variables*, and those to the left are considered *primary variables*. A Trellis formula must contain at least one primary variable, but conditioning variables are optional. The conditioning symbol `|` must be omitted if there are no conditioning variables. There is no limit on the number of conditioning variables that can be specified, although the majority of actual use is covered by up to two. Conditioning variables may be separated by `*` or `+`; unlike many modeling functions, these are treated identically in lattice. The conditioning part of the formula has the same interpretation for all lattice functions, whereas that for the first part may vary by function. Thus,

```
~ x
```

and

```
log(z) ~ x * y | a + b + c
```

are both valid Trellis formulae (although not in all high-level functions). As the last example suggests, the formula can involve terms that are expressions involving one or more variables. After evaluation, all terms in the formula have to have the same length.

2.1.2 The `data` argument

Apart from specifying the structure of the display, use of a formula also allows one to separately specify, as the `data` argument, an object containing variables referenced in the formula. This is similar to other formula-based interfaces in R,[1] and reduces the temptation to use `attach()` (which is fraught with pitfalls) by obviating the need to repeatedly refer to the data source by name.[2] The `data` argument occupies the second position in the list of arguments in all high-level lattice functions, and is often not named in a call.

A less obvious implication of having a separate `data` argument is that methods can extend the types of objects that can act as a data source. The standard *"formula"* methods allow `data` to be data frames, lists, or environments (see `?eval`). In Chapter 14, we show how other types of objects may be used.

2.1.3 Conditioning

The case where the Trellis formula does not have any conditioning variables is fairly straightforward. To give analogies with base graphics functions, `histogram(~ x)` is similar to `hist(x)`, `xyplot(y ~ x)` is similar to `plot(x, y)`

[1] With the same caveats, briefly described in Section 10.1.

[2] An alternative is to use `with()`, which is sometimes more convenient.

or `plot(y ~ x)`, and so on. The rest of this chapter primarily deals with the situation where we do have one or more conditioning variables. In the first case, we can simply pretend to have one conditioning variable with a single level.

Conditioning variables are most often categorical variables, or *factors* in R parlance. They can also be *shingles*, which provide means to use continuous variables for conditioning.

Factors have a set of levels, representing its possible values. Each unique combination of the levels of the conditioning variables determines a *packet*, consisting of the subset of the primary variables that correspond to that combination.[3] It is possible for a packet to be empty if the corresponding combination of levels is not represented in the data. Each packet potentially provides the data for a single panel in the Trellis display, which consists of such panels laid out in an array of columns, rows, and pages. Choosing a proper layout is critical in obtaining an informative display; lattice tries to make the default choice as useful as possible, and provides ways to customize it.

Although packets are defined entirely by the formula, it is possible to omit or repeat certain levels of the conditioning variables when displaying a *"trellis"* object, in which case the corresponding packets may be omitted or repeated as well. Examples of these can be found in Figures 2.2 and 11.4.

2.1.4 Shingles

Multivariable relationships often involve many continuous variates, and the ability to condition on them is useful. Shingles afford a very general means to do so. The simplest possible approach to using a numeric variable for conditioning is to treat each of its unique values as a distinct level. This is, in fact, the default behavior in lattice. However, this is often unhelpful when the number of unique values is large. Another standard way to convert a continuous variate into an ordinal categorical variable is to discretize it, that is, to partition its range into two or more non-overlapping intervals, and replace each value by only an indicator of the interval to which it belonged. Such discretization can be performed by the R function `cut()`.

Shingles encompass both these ideas and extend them by allowing the intervals defining the discretization to overlap.[4] The intervals can be single points, or have no overlap, thus reducing to the two approaches described above. Each such interval is now considered a "level" of the shingle. Clearly, the level of a particular observation is no longer necessarily unique, as it can fall into more than one interval. This is not a hindrance to using the shingle

[3] Strictly speaking, a vector of subscripts indicating which rows in the original data contribute to the packet is also often part of the packet, although this is an irrelevant detail in most situations.

[4] Shingles are named after the overlapping pieces of wood or other building material often used to cover the roof or sides of a house.

as a conditioning variable; observations that belong to more than one level of the shingle are simply assigned to more than one packet.

This still leaves the issue of how best to choose the intervals that define a shingle, given a continuous variate. Cleveland (1993) suggests the "equal count" algorithm, which given a desired number of levels and amount of overlap, chooses the intervals so that each interval has roughly the same number of observations. This algorithm is used by the `equal.count()` function in the lattice package to create shingles from numeric variables. Shingles are discussed further in Chapter 10.

2.2 Dimension and physical layout

Multipanel conditioning can be viewed as an extended form of cross-tabulation, naturally conferring the concept of dimensions to *"trellis"* objects. Specifically, each conditioning variable defines a dimension, with extents given by the number of levels it has.

Consider the following graph, which uses data from a split-plot experiment (Yates, 1935) where yield of oats was measured for three varieties of oats and four nitrogen concentrations within each of six blocks.

```
> data(Oats, package = "MEMSS")
> tp1.oats <-
      xyplot(yield ~ nitro | Variety + Block, data = Oats, type = "o")
```

Although we do not formally encounter the `xyplot()` function until later, this is fairly typical usage, resulting in a scatter plot of yield against nitrogen concentration in each panel. The display produced by plotting `tp1.oats` is given in Figure 2.1. There are two conditioning variables (dimensions), with three and six levels. This is reflected in

```
> dim(tp1.oats)
[1] 3 6
> dimnames(tp1.oats)
$Variety
[1] "Golden Rain" "Marvellous"  "Victory"

$Block
[1] "I"   "II"  "III" "IV"  "V"   "VI"
```

These properties are shared by the cross-tabulation defining the conditioning.

```
> xtabs(~Variety + Block, data = Oats)
             Block
Variety       I II III IV V VI
  Golden Rain 4  4   4  4 4  4
  Marvellous  4  4   4  4 4  4
  Victory     4  4   4  4 4  4
```

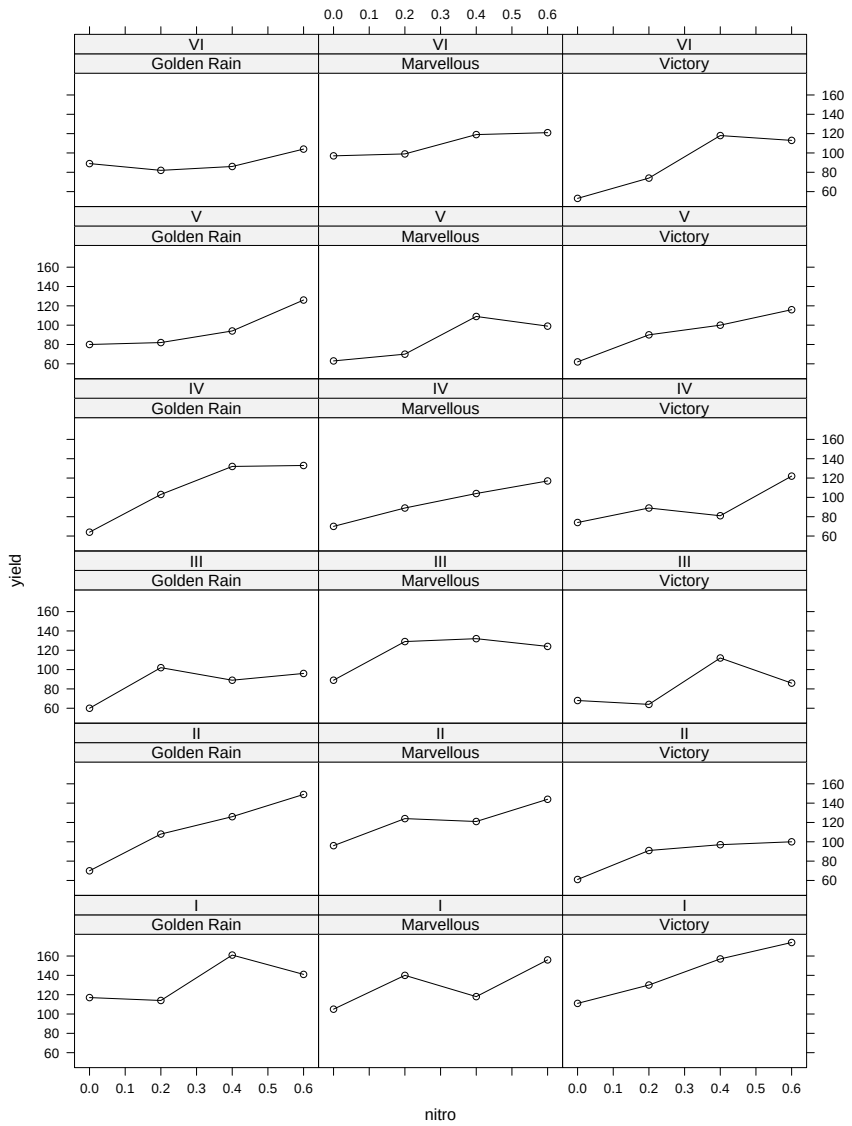

Figure 2.1. A Trellis display of the `Oats` data. The yield of oats is plotted against nitrogen concentration for three varieties of oats and six blocks. This is an example of a split-plot design. See `help(Oats, package = "MEMSS")` for more details about the experiment.

This cross-tabulation, with suitable modifications for shingles to account for the fact that packets may overlap, is in fact printed when a *"trellis"* object is summarized:

```
> summary(tp1.oats)
Call:
xyplot(yield ~ nitro | Variety + Block, data = Oats, type = "o")

Number of observations:
             Block
Variety       I II III IV V VI
  Golden Rain 4  4   4  4 4  4
  Marvellous  4  4   4  4 4  4
  Victory     4  4   4  4 4  4
```

It is possible to extract subsets of such objects in a natural way, treating them as arrays with the appropriate dimensions; for example,

```
> summary(tp1.oats[, 1])
Call:
xyplot(yield ~ nitro | Variety + Block, data = Oats, type = "o",
    index.cond = new.levs)

Number of observations:
             Block
Variety       I
  Golden Rain 4
  Marvellous  4
  Victory     4
```

The corresponding plot is shown in Figure 2.2. This view of a *"trellis"* object implies a linear ordering of the packets in it, similar to the ordering of elements in general arrays in R. Specifically, the order begins with the packet corresponding to the first index (level) of each dimension (conditioning variable) and proceeds by varying the index of the first dimension fastest, then the second, and so on. This order is referred to as the *packet order*.

Another array-like structure comes into play when a *"trellis"* object is actually displayed, namely, the physical layout of the panels. Whereas the number of dimensions of the abstract object is arbitrary, a display device is conventionally bound to two dimensions. Trellis displays, in particular, choose to divide the display area into a rectangular array of panels. An additional dimension is afforded by spreading out a display over multiple pages, which can be important in displays with a large number of combinations. All high-level lattice functions share a common paradigm that dictates how this layout is chosen, and provides common arguments to customize it to suit particular situations. Once the layout is determined, it defines the *panel order*, that is, the sequential order of panels in the three-way layout of columns, rows, and pages. The eventual display is created by matching packet order with panel

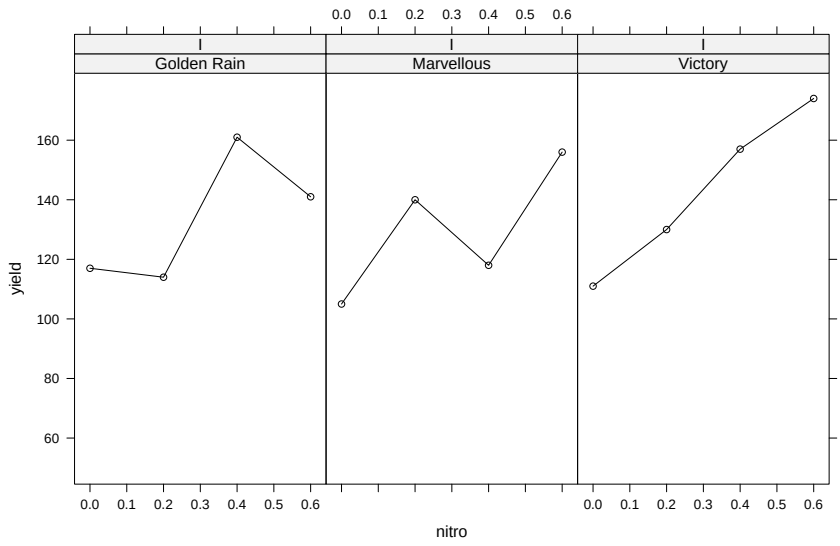

Figure 2.2. A Trellis display of a subset of a *"trellis"* object. The display represents the bottom row of Figure 2.1.

order.[5] The rest of this section discusses details of how the layout is controlled, and the choice of aspect ratio, which is closely related.

2.2.1 Aspect ratio

The aspect ratio of a panel is the ratio of its physical height and width. The choice of aspect ratio often plays a crucial role in determining the effectiveness of a display. There is no general prescription for choosing the aspect ratio, and one often needs to arrive at one by trial and error. In certain situations, a good aspect ratio can be automatically determined by the 45° banking rule, which is derived from the following idea. Consider a display, such as the `Oats` example above, where the changes in successive values (represented by line segments) contain information we wish to perceive. For a non-zero change, the corresponding line grows steeper as the aspect ratio increases, and shallower as it decreases. Cleveland et al. (1988) note that this information is best grasped when the orientation of such line segments is close to 45°, and recommend an algorithm that can be used to select an aspect ratio automatically based on this criterion. When the `aspect = "xy"` argument is specified in a high-level call, this 45° banking rule is used to compute the aspect ratio (see Chapter 8 for details). The `aspect` argument can also be an explicit numeric ratio, or the string `"iso"`, which indicates that the number of units per cm (i.e., the

[5] For the record, this can be changed; see `?packet.panel.default` for details.

relation between physical distance on the display and distance in the data scale) should be the same for both axes. This is appropriate in situations where the two scales have the same units, for example, in plots of spatial data, or plots of ROC curves where both axes represent probability.

2.2.2 Layout

A good choice of layout needs to take the aspect ratio into account. To make this point, let us look at Figure 2.3, which is produced by updating[6] Figure 2.1 to use an aspect ratio chosen by the 45° banking rule. As we can see, the default display does not make effective use of the available space. This is related to the rules that determine the default layout.

A Trellis display consists of several panels arranged in a rectangular array, possibly spanning multiple pages. The `layout` argument determines this arrangement. For an exact specification, `layout` should be a numeric vector giving the number of columns, rows, and pages in a multipanel display. Unless one wants to restrict the number of pages, the third element need not be specified; it is automatically chosen to accommodate all panels. The coordinate system used by default is like the Cartesian coordinate system: panels are drawn starting from the lower-left corner, proceeding first right and then up. This behavior can be changed by setting `as.table = TRUE` in a high-level lattice call,[7] in which case panels are drawn from the upper-left corner, going right and then down.

If there are two or more conditioning variables, `layout` defaults to the lengths of the first two dimensions, that is, the number of columns defaults to the number of levels of the first conditioning variable and the number of rows to the number of levels of the second conditioning variable (consequently, the number of pages is implicitly the product of the number of levels of the remaining conditioning variables, if any). This is clearly a sensible default, even though it is responsible for the somewhat awkward display in Figure 2.3.

The obvious way to "fix" Figure 2.3 is to switch the order of the conditioning variables. This can be done by regenerating the *"trellis"* object, or by simply transposing the existing one using

```
> t(tp1.oats)
```

However, we use another approach that makes use of a special form of the `layout` argument. The first element of `layout` can be 0, in which case its second element is interpreted as (a lower bound on) the total number of panels per page, leaving the software free to choose the exact layout. This is done by considering the aspect ratio and the device dimensions, and then choosing the layout so that the space occupied by each panel is maximized. The result of using this on our plot of the `Oats` data is given in Figure 2.4, where much better use is made of the available space.

[6] The `update()` function is formally discussed in Chapter 11.

[7] Chapter 7 describes how to change the default for `as.table` globally.

```
> update(tp1.oats,
         aspect="xy")
```

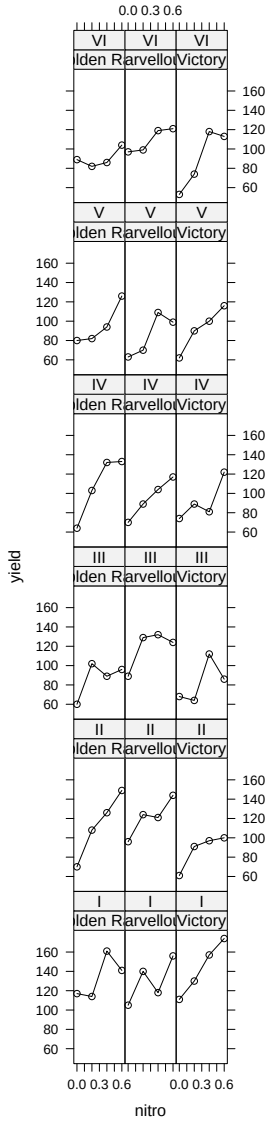

Figure 2.3. The display in Figure 2.1 updated to use the 45° banking rule to choose an aspect ratio. Although it is now easier to assess the changes in yield, the default layout results in considerable wastage of the available display area. This can be rectified using the `layout` argument, as we show in our next attempt.

```
> update(tp1.oats, aspect = "xy",
         layout = c(0, 18))
```

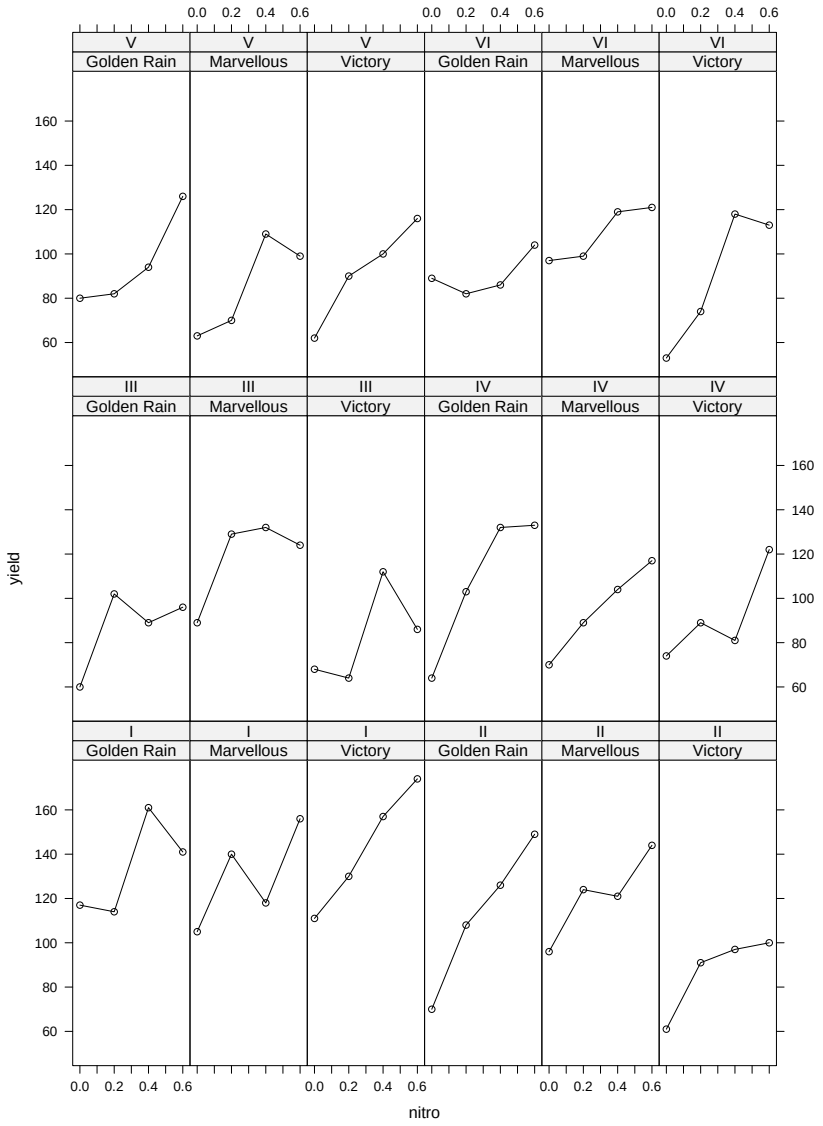

Figure 2.4. The display in Figure 2.3 updated to use an unconstrained layout. The aspect ratio calculated by the banking rule is taken into account when computing the layout, resulting in larger panels than before. However, there are now multiple blocks in each row of the layout, with no visual cue drawing attention to this fact.

```
> update(tp1.oats, aspect = "xy", layout = c(0, 18),
         between = list(x = c(0, 0, 0.5), y = 0.5))
```

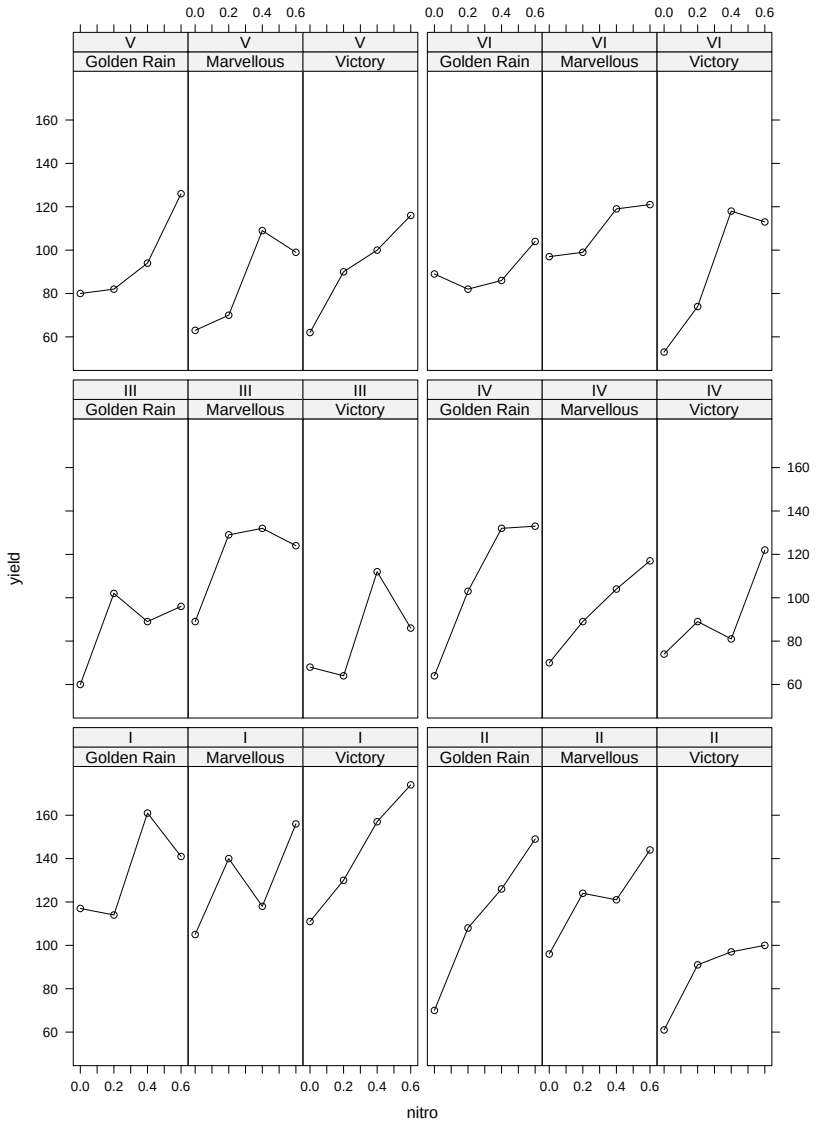

Figure 2.5. Figure 2.4 updated to put spacing between appropriate columns and rows, providing a visual cue separating panels into groups of blocks. This is possible because the layout happens to have exactly two blocks in every row; that is, none of the blocks spans multiple rows.

If there is only one conditioning variable with `n` levels, the default value of `layout` is `c(0,n)`, thus taking advantage of this automatic layout computation. When `aspect = "fill"` (the default in most cases), this computation is carried out with an initial aspect ratio of 1, but in the eventual display the panels are expanded to fill up all the available space.

2.2.3 Fine-tuning the layout: `between` and `skip`

The `between` argument can be a list, with components `x` and `y` (both usually 0 by default) which are numeric vectors specifying the amount of blank space between the panels (in units of character heights). `x` and `y` are repeated to account for all panels in a page, and any extra components are ignored. This is often useful in providing a visual cue separating panels into blocks, as in Figure 2.5.

Another argument useful in fine-tuning the layout is `skip`, which is specified as a logical vector (default `FALSE`), replicated to be as long as the number of panels. For elements that are `TRUE`, the corresponding panel position is skipped; that is, nothing is plotted in that position. The panel that was supposed to be drawn there is now drawn in the next available panel position, and the positions of all the subsequent panels are bumped up accordingly. This is often useful for arranging plots in an informative manner.

2.3 Grouped displays

Trellis graphics is intended to foster easy and effective visualization of multivariate relationships in a dataset. As we saw in Chapter 1, a powerful construct that forces direct comparison is superposition, where data associated with different levels of a grouping variable are rendered together within a panel, but with different graphical characteristics. For example, different curves could be drawn in different color or line type, or points could be drawn with different symbols. Superposition is usually more effective than multipanel conditioning when the number of levels of the grouping variable is small. For many lattice functions, specifying a `groups` argument that refers to a categorical variable is enough to produce a "natural" grouped display.

We have seen grouped displays in Chapter 1. Perhaps the most well-known example in the context of Trellis graphics is Figure 1.1 from Cleveland (1993), which is recreated in Figure 2.6 using the following code.

```
> dotplot(variety ~ yield | site, barley,
          layout = c(1, 6), aspect = c(0.7),
          groups = year, auto.key = list(space = "right"))
```

The plot is a visualization of data from a barley experiment run in Minnesota in the 1930s (Fisher, 1971), and discussed extensively by Cleveland (1993). The plot effectively combines grouping and conditioning to highlight an anomaly in the data not easily noticed otherwise.

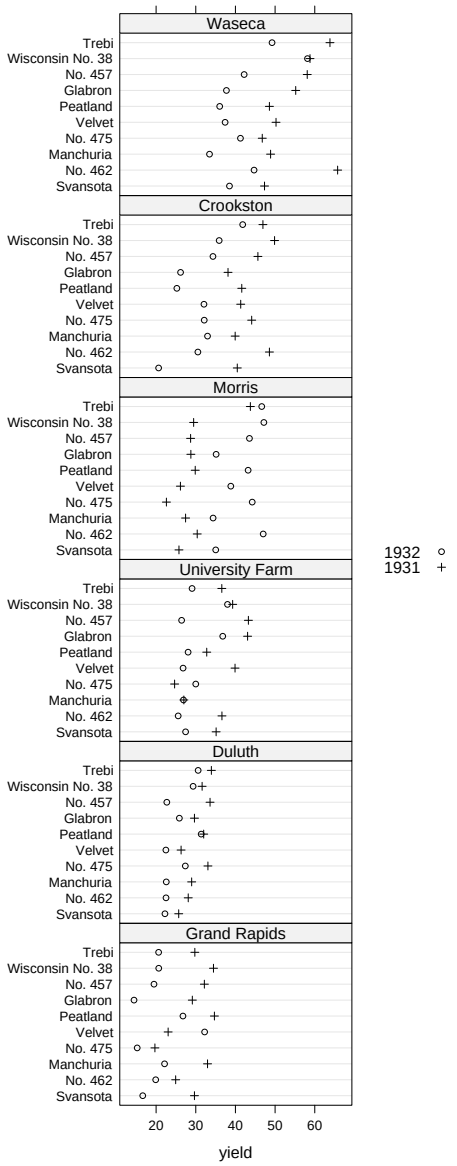

Figure 2.6. A multiway dot plot of data from a barley experiment run in Minnesota in the 1930s. Yield is plotted for several varieties of barley, conditioned on six sites. Different symbols are used to differentiate the year. The grouping and conditioning combine to highlight an anomaly in the data from Morris. Another subtle choice that enhances the effectiveness of the display is the ordering of the panels (sites) and the y variable (variety).

2.4 Annotation: Captions, labels, and legends

In Figure 2.6, as in Chapter 1, we have annotated the display by adding a legend, or key, that explains the correspondence of the different symbols to the respective levels of the grouping variable. Such legends are natural in grouped displays, but are not drawn by default. Usually, the simplest (although not the most general) way to add a suitable legend to a grouped display is to set `draw.key = TRUE` in the call. Often the key thus produced needs minor tinkering to get a more desirable result; this can be achieved by specifying `auto.key` as a list with suitable components. Generally speaking, legends can be placed in any of the four sides of a display, in which case enough space is automatically allocated for them. Alternatively, they can be placed anywhere inside the display, in which case no extra space is left, and the user has to make sure that they do not interfere with the actual display.

Other common means of annotating a display are to add meaningful captions and labels. Just as with traditional high-level graphics functions, most lattice functions allow the addition of four basic captions: a main title at the top (specified by the argument `main`), a subtitle at the bottom (`sub`), an x-axis label just below the x-axis (`xlab`), and a y-axis label to the left of the y-axis (`ylab`). `xlab` and `ylab` usually have some sensible defaults, whereas the other two are omitted. These labels are usually text strings, but can also be *"expression"* objects,[8] or for more generality, arbitrary grid objects (grobs). Another type of annotation directly supported by lattice functions is through the `page` argument. If specified, it has to be a function, and is called after each page is drawn. It can be used, for example, to mark the page numbers in a multipage display.

A full discussion of these annotation facilities is given in Chapter 9. Here, in Figure 2.7, we present one simple example with various labels and a legend. However, to fully appreciate even this simple example, we need to learn a little about how legends are specified.

2.4.1 More on legends

The construction of legends is a bit more involved than text labels, because they potentially have more structure. A template rich enough for most legends is one with (zero, one, or more) columns of text, points, lines, and rectangles, with suitably different symbols, colors, and so on. Such legends can be constructed using the `draw.key()` function, which can be indirectly used to add a legend to a plot simply by specifying a suitable list as the `key` argument in a high-level lattice function. To construct this list, we need to know what goes into the legend. The one in Figure 2.7 has a column of text with the levels of `Variety`, and a column of points with the corresponding symbols.

[8] Expressions, as typically produced by the `expression()` function, can be used to produce LaTeX-like mathematical annotation, as described in the help page `?plotmath`.

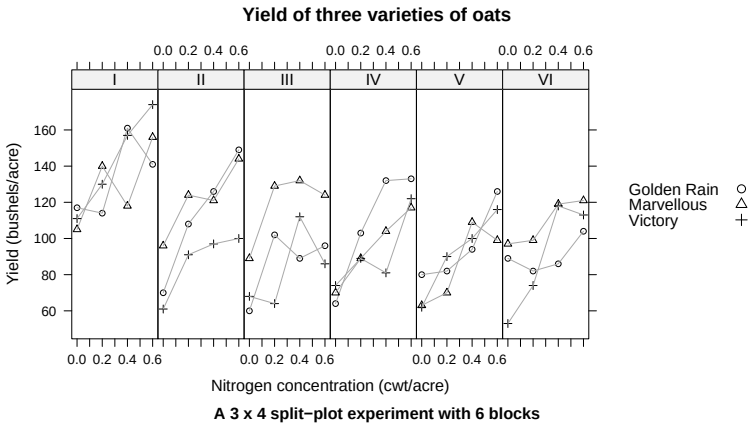

Figure 2.7. An alternative display of the `Oats` data. `Variety` is now used as a grouping variable, and a legend describes the association between its levels and the corresponding plotting characters. Various other labels are also included.

Here we run into a problem. The symbols and colors used by default in a lattice display are not determined until the plot is actually drawn, so that the current graphical settings can be taken into account (see Chapter 7 for details). For example, most plots on the pages of this book are black and white, but a reader trying to reproduce them will most likely do so interactively on a computer terminal, and will see them in color. In other words, when making the call to `xyplot()`, we do not know what the graphical parameters in the plot, and hence the legend, are going to be. A clumsy solution, used to produce Figure 2.7, is to bypass the problem by explicitly specifying the colors and symbols in the call itself.

```
> key.variety <-
      list(space = "right", text = list(levels(Oats$Variety)),
           points = list(pch = 1:3, col = "black"))
> xyplot(yield ~ nitro | Block, Oats, aspect = "xy", type = "o",
         groups = Variety, key = key.variety, lty = 1, pch = 1:3,
         col.line = "darkgrey", col.symbol = "black",
         xlab = "Nitrogen concentration (cwt/acre)",
         ylab = "Yield (bushels/acre)",
         main = "Yield of three varieties of oats",
         sub = "A 3 x 4 split-plot experiment with 6 blocks")
```

In most cases, a better solution is to use the `auto.key` argument, which we have already encountered on a couple of occasions. Chapter 9 examines this problem in more detail and explains the precise role of `auto.key`.

2.5 Graphing the data

At the end of the day, the usefulness of a statistical graphic is determined by how it renders the information it is supposed to convey. Multipanel conditioning, if used, imposes some preliminary structure on a Trellis display by systematically dividing up the data in a meaningful way. After determining these data subsets (packet) and their layout, they next need to be graphed. This involves a graphical encoding of the data, typically with a rendering of the relevant axes (tick marks and labels) to provide a frame of reference. For multipanel displays, an additional element describing each panel, specifically the associated levels of the conditioning variables, is necessary. This is done using strips, which can be customized or completely omitted by specifying a suitable `strip` (and in some cases `strip.left`) argument to any high-level lattice function (see Section 10.7 for details).

A fundamental assumption made in the Trellis design is that the nature of the graphical encoding will be repetitive; that is, the same *procedure* will be used to visualize each packet. This permits a decoupling of the procedures that draw the data and the axes, which can then be controlled separately. Recall that each panel in the display has an associated packet, a subset of the entire data. The exact form of a packet will depend on the high-level function used. Given the prescription for the graphic, a packet determines the data rectangle, a two-dimensional region enclosing the graphic. For example, in a bivariate scatter plot this is usually a rectangle defined by the range of the data; for a histogram, the horizontal extent of the data rectangle is the minimal interval containing all the bins, and the vertical scale ranges from 0 at the bottom to the height of the highest bin (which would depend on the type of histogram drawn) at the top. Another possibly relevant piece of information determined by the packet is a suitable aspect ratio for this data rectangle. In all lattice displays, these pieces of information are computed by the so-called prepanel function, which is discussed in detail in Chapter 8. Note that this view is not entirely satisfactory, as for some displays (e.g., scatter-plot matrices using `splom()` and three-dimensional scatter plots using `cloud()`) the usual axes have no meaning and the data display procedure itself has to deal with scales.

2.5.1 Scales and axes

For a single-panel display, one can proceed to draw the axes and the graphic once the data rectangle and aspect ratio are determined. However, for multipanel displays, there needs to be an intermediate step of combining the information from different packets. A common aspect ratio is chosen by some form of averaging if necessary. There are three alternative rules available to determine the scales. The default choice is to use the same data rectangle for each panel, namely, the smallest rectangle that encloses all individual data rectangles. This allows easy visual comparison between panels without constantly having to refer to the axes. This choice also allows the panels to share a

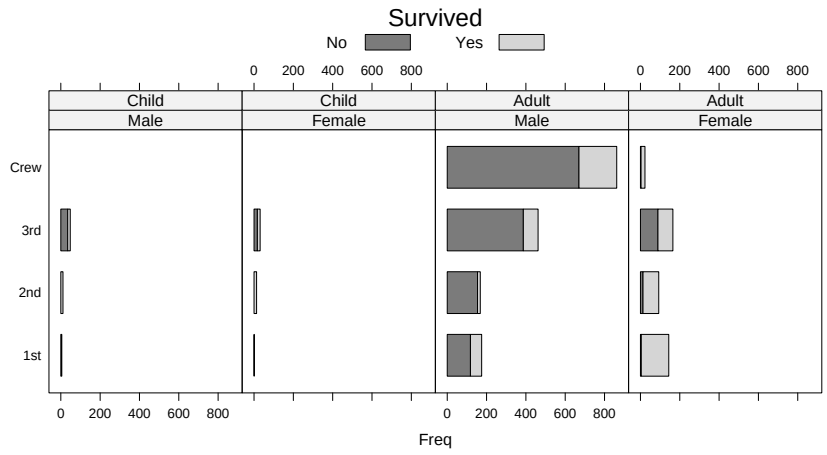

Figure 2.8. A bar chart summarizing the fate of passengers of the Titanic, classified by sex, age, and class. The plot is dominated by the third panel (adult males) as heights of the bars encode absolute counts, and all panels have the same limits.

common set of tick marks and axis labels along the boundary, saving valuable space. Sometimes this is not satisfactory because the ranges of the data in different packets are too different. If the data do not have a natural baseline and the relevant comparison is essentially done in terms of differences, it often suffices to have different scales as long as the number of units per cm is the same. The third choice, mainly useful for qualitative comparisons, is to allow completely independent scales, in which case the data rectangle for each panel is determined just by the corresponding packet. All these choices can be made selectively for either axis. The choice of which rule to use is controlled by the `scales` argument, which can also be used to control other aspects of axis annotation, such as the number of tick marks, position and labels of ticks, and so on. More directly, the arguments `xlim` and `ylim` allow explicit specification of the data rectangle, overriding the default calculations. This is an important and extensive topic, and is given due consideration in Chapter 8. We give one simple example here.

The `Titanic` dataset provides (as a four-dimensional array) a cross-tabulation of the fates of 2201 passengers of the famous ship, categorized by economic status (class), sex, and age. To use the data in a lattice plot, it is convenient to coerce it into a data frame. Our first attempt might look like the following, which produces Figure 2.8.

```
> barchart(Class ~ Freq | Sex + Age, data = as.data.frame(Titanic),
           groups = Survived, stack = TRUE, layout = c(4, 1),
           auto.key = list(title = "Survived", columns = 2))
```

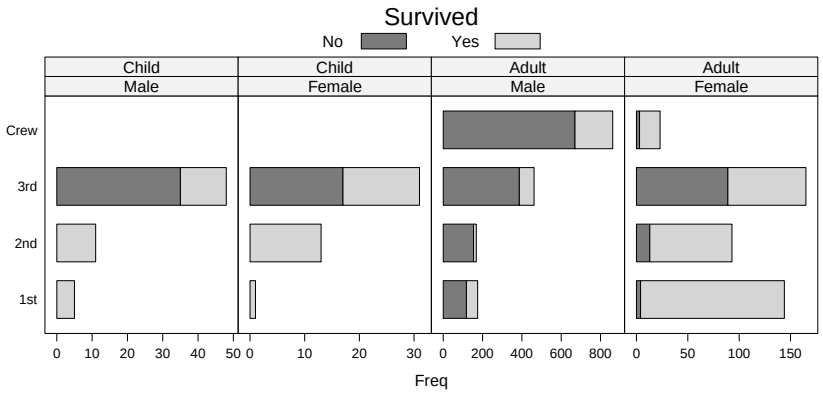

Figure 2.9. Survival among different subgroups of passengers on the Titanic, with a different horizontal scale in each panel. This emphasizes the proportion of survivors within each subgroup, rather than the absolute numbers. The proportion of survivors is smallest among third-class passengers, although the absolute number of survivors is not too low compared to the other classes.

All this plot really tells us is that there were many more males than females aboard (particularly among the crew, which is the largest group), and that there were even fewer children; which, although true, is unremarkable. The point we really want to make is that the "save women and children first" policy did not work as well for third-class passengers. This is more easily seen if we emphasize the proportions of survivors by allowing independent horizontal scales for different panels. Figure 2.9 is created using

```
> barchart(Class ~ Freq | Sex + Age, data = as.data.frame(Titanic),
           groups = Survived, stack = TRUE, layout = c(4, 1),
           auto.key = list(title = "Survived", columns = 2),
           scales = list(x = "free"))
```

2.5.2 The panel function

Once the rest of the structure (layout, data rectangles, annotation) is in place, packets are plotted in the appropriate panel. The actual plotting is done by a separate function, known as the *panel function* and specified as the `panel` argument, that is executed once for every panel with the associated data packet as its arguments. Each high-level lattice function has its own default panel function. By convention, the name of this function is given by "`panel.`" followed by the name of the high-level function. For example, the default panel function for `barchart()` is called `panel.barchart`, that for `histogram()` is `panel.histogram`, and so on. The remaining chapters in Part I describe the various high-level functions and their default panel functions in greater detail.

A lot can be achieved by the default panel functions, but one is not restricted to them by any means. In fact, it is the ability to define custom panel functions that allows the user to create novel Trellis displays easily, a process described in depth in Chapter 13. Even when predefined panel functions are adequate, an understanding of this process can greatly enhance the ability to use them effectively. For this reason, we spend some time here exploring this aspect. Readers new to R and lattice may want to skip the next part on first reading if they find it confusing.

2.5.3 The panel function demystified

Panel functions are, first and foremost, functions. This may sound obvious, but the concept of functions as arguments to other functions is often difficult to grasp for those not used to functional languages. To fix ideas, let us consider the call that produced Figure 2.9. As we plan to experiment just with the panel function, there is no point in repeating the full call every time. So, we save the object in a variable and use the `update()` method to manipulate it further.

```
> bc.titanic <-
      barchart(Class ~ Freq | Sex + Age, as.data.frame(Titanic),
               groups = Survived, stack = TRUE, layout = c(4, 1),
               auto.key = list(title = "Survived", columns = 2),
               scales = list(x = "free"))
```

Figure 2.9 can be reproduced by printing this object.

```
> bc.titanic
```

Because the default panel function for `barchart()` is `panel.barchart()`, this is equivalent to

```
> update(bc.titanic, panel = panel.barchart)
```

which has the same effect as specifying `panel = panel.barchart` in the original call. Note that the result of the call to `update()`, which is itself an object of class *"trellis"*, has not been assigned to a variable and will thus be printed as usual. The variable `bc.titanic` remains unchanged. To make more explicit the notion that `panel` is a function, we can rewrite this as

```
> update(bc.titanic,
         panel = function(...) {
             panel.barchart(...)
         })
```

Although this does nothing new, it illustrates an important feature of the S language whose significance is easy for the beginner to miss; namely the `...` argument. Complicated functions usually achieve their task by calling simpler functions. The `...` argument in a function is a convenient way for it to capture arguments that are actually meant for another function called by it, without needing to know explicitly what those arguments might be. This trick

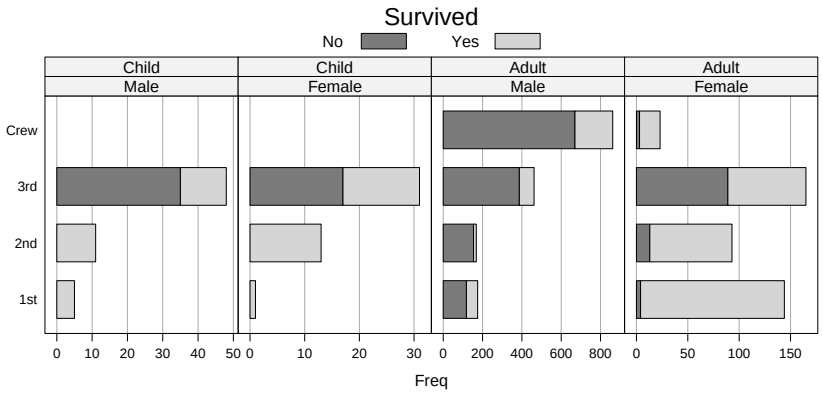

Figure 2.10. A modified version of Figure 2.9, with vertical reference lines added in the background. This is achieved using a custom panel function.

is very useful in lattice calls, because often one wants not to replace the panel function, but to add to it. A typical example is the addition of reference lines. The function `panel.grid()`, which is one of many utility functions in lattice, can be used to draw such reference lines as follows to produce Figure 2.10.

```
> update(bc.titanic,
         panel = function(...) {
             panel.grid(h = 0, v = -1)
             panel.barchart(...)
         })
```

Thanks to the `...` argument, we used `panel.barchart()` without even knowing what arguments it accepts. It should also be noted that without the call to `panel.barchart()` in our custom panel function, only the reference lines would have been drawn.

Most default panel functions are designed to be quite flexible by themselves, and simple variations can frequently be achieved by changing one or more of their arguments. Suppose that we want to remove the black borders of the bars in Figure 2.10, which do not really serve any purpose as the bars are already shaded. Most panel functions have arguments to control the graphical parameters they use; in `panel.barchart()`, the border color is determined by the `border` argument (as described in the documentation). Thus, to make the borders transparent, we can use

```
> update(bc.titanic,
         panel = function(..., border) {
             panel.barchart(..., border = "transparent")
         })
```

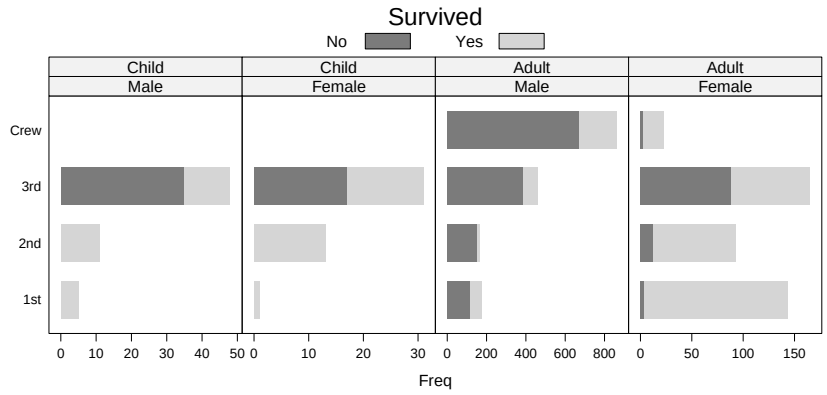

Figure 2.11. Another version of Figure 2.9 with the borders of the bars made transparent. This can be achieved using a custom panel function, but a simpler alternative is to specify a `border` argument to `barchart()` which is passed on to `panel.barchart()`.

which produces Figure 2.11. Once again, we make use of `panel.barchart()` without needing to know what its arguments are, except for the one we wanted to change.

This brings us to a simple, but extremely useful feature of high-level lattice functions. All of them have a `...` argument, and will thus accept without complaint any number of extra named arguments. After processing the arguments it recognizes itself, a high-level function will collect all remaining arguments and pass them on to the panel function whenever it is called. The implication of this is that arguments that are intended for panel functions can be given directly to the high-level function. This panel function can of course be the default one, in which case the user does not even have to specify the panel function explicitly. Thus, an alternative way to produce Figure 2.11 is

```
> update(bc.titanic, border = "transparent")
```

We have already used this feature several times so far, and do so extensively in the next few chapters as well.

2.6 Return value

As briefly discussed in Chapter 1, high-level lattice functions do not draw anything themselves, instead returning an object of class *"trellis"*. In this chapter, we have made use of this fact several times without drawing attention to it, especially when calling the convenient `update()` method to make incremental changes to such objects. We learn more about *"trellis"* objects in Chapter 11.