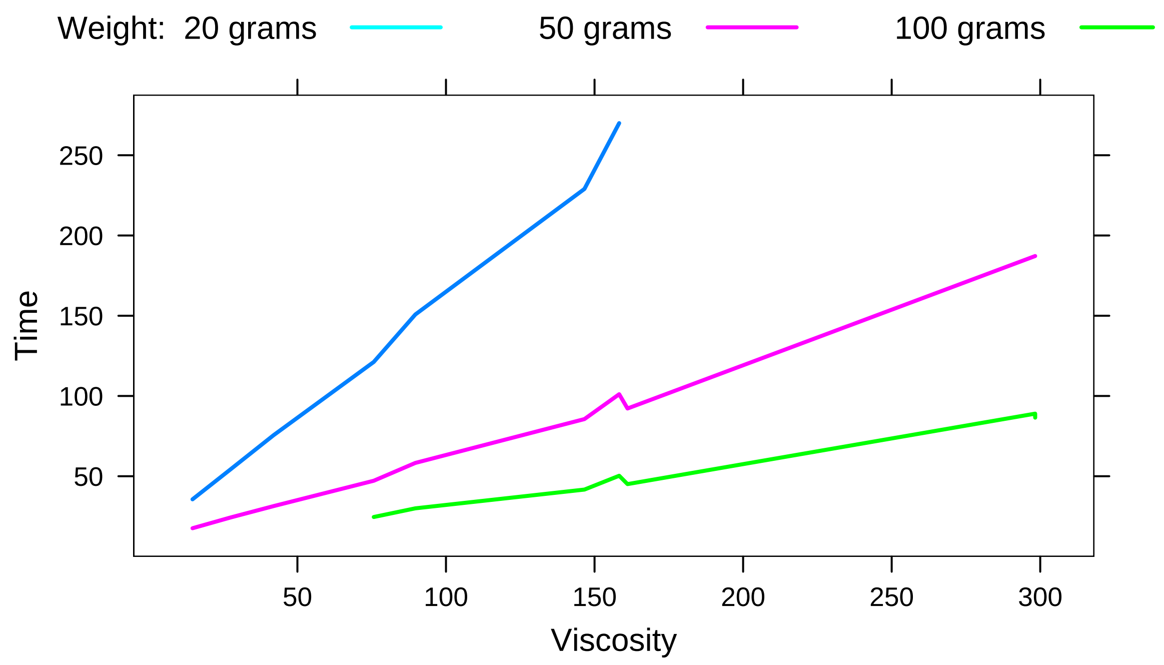
Trellis Graphics Continued

Statistics 135

Autumn 2005

`groups` **Option**

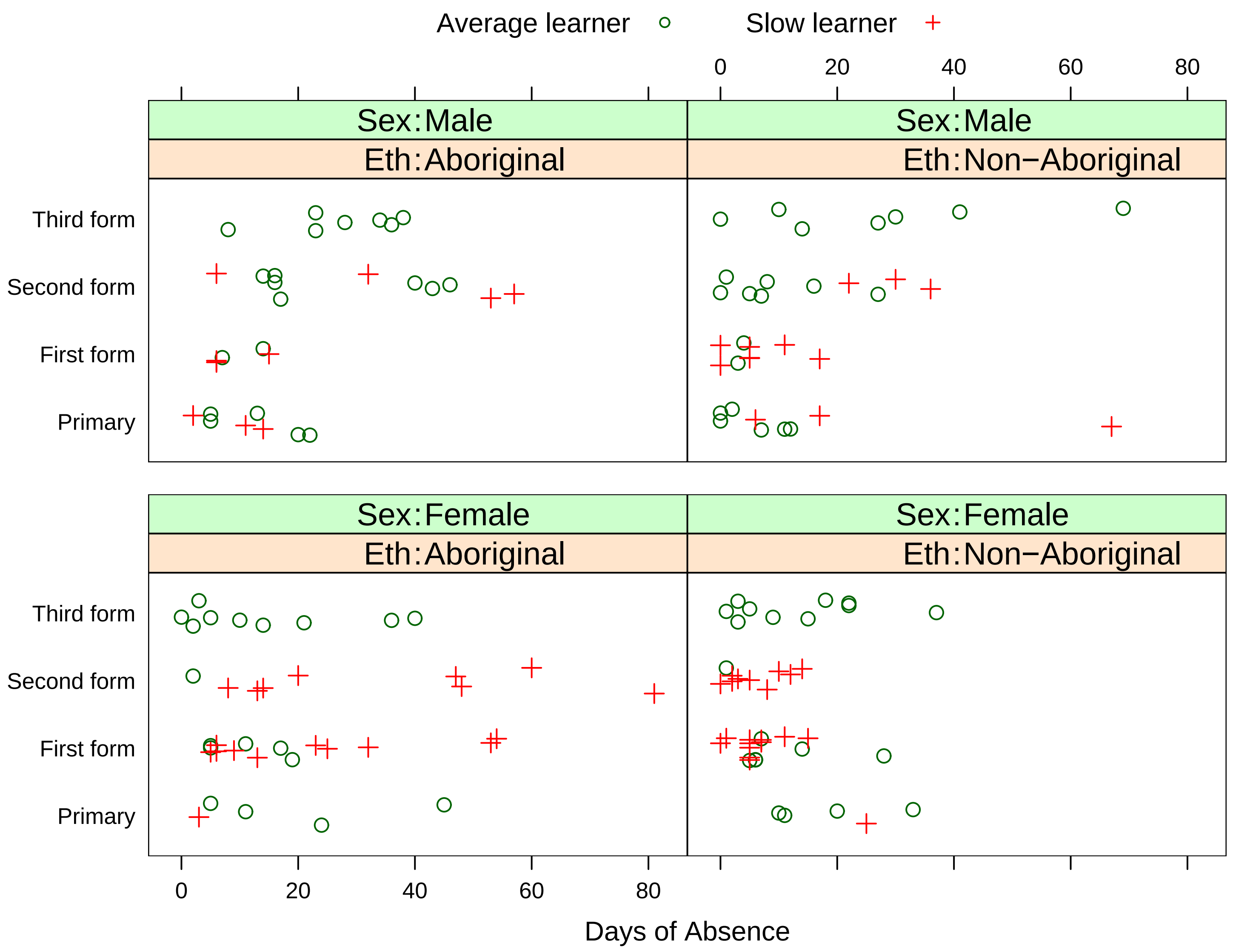
```
sps <- trellis.par.get("superpose.symbol")
xyplot(Time ~ Viscosity, data=stormer, groups=Wt,
  panel = panel.superpose, type="l",
  key = list(columns=3, lines=Rows(sps,1:3),
      text=list(paste(c("Weight: ", "",""), unique(stormer$Wt), "grams")))
  )
```



As mentioned last time, the option `groups` allows for an additional way of adding a conditioning variable to a trellis plot.

Instead of adding addition panels to the figure, for each panel, based on the conditioning variables `v1 * v2 * ... * vn`, the plotting is done for each level of the group variable.

For example, another way of looking at the `quine` dataset is via a stripplot, where instead of treating `Lrn` as a conditioning variable, lets use it as a `groups` variable instead.


Average learner ○
Slow learner +
Sex:Male
Eth:Aboriginal
Sex:Male
Eth:Non−Aboriginal
Sex:Female
Eth:Aboriginal
Sex:Female
Eth:Non−Aboriginal
Third form
Second form
First form
Primary
0
20
40
60
80
Days of Absence

```
trellis.device("postscript", file="../quinestrip.eps", width=8, height=6,
  horiz=F, col=T, theme="col.whitebg")
sps <- trellis.par.get("superpose.symbol")
sps$pch <- 16:22
trellis.par.set("superpose.symbol",sps)

stripplot(Age ~ Days | Eth*Sex, data=Quine,
  groups=Lrn, jitter=T, cex=1,
  panel = function(x,y, subscripts, jitter.data=F, ...) {
    y <- as.numeric(y)  # only needed in R
    if(jitter.data) y <- jitter(y)
    panel.superpose(x,y,subscripts,...)
  },
  xlab = "Days of Absence",
  between = list(y=1), par.strip.text = list(cex=1.2),
  key = list(columns=2, text=list(levels(Quine$Lrn)),
    points=Rows(trellis.par.get("superpose.symbol"), 1:2), cex=1),
  strip = function(...)
    strip.default(..., strip.names=c(T,T),style=1)
)
dev.off()
```

In this plot, the points are jittered slightly, which moves the plotting location by a small amount, so overlapping points can be seen.

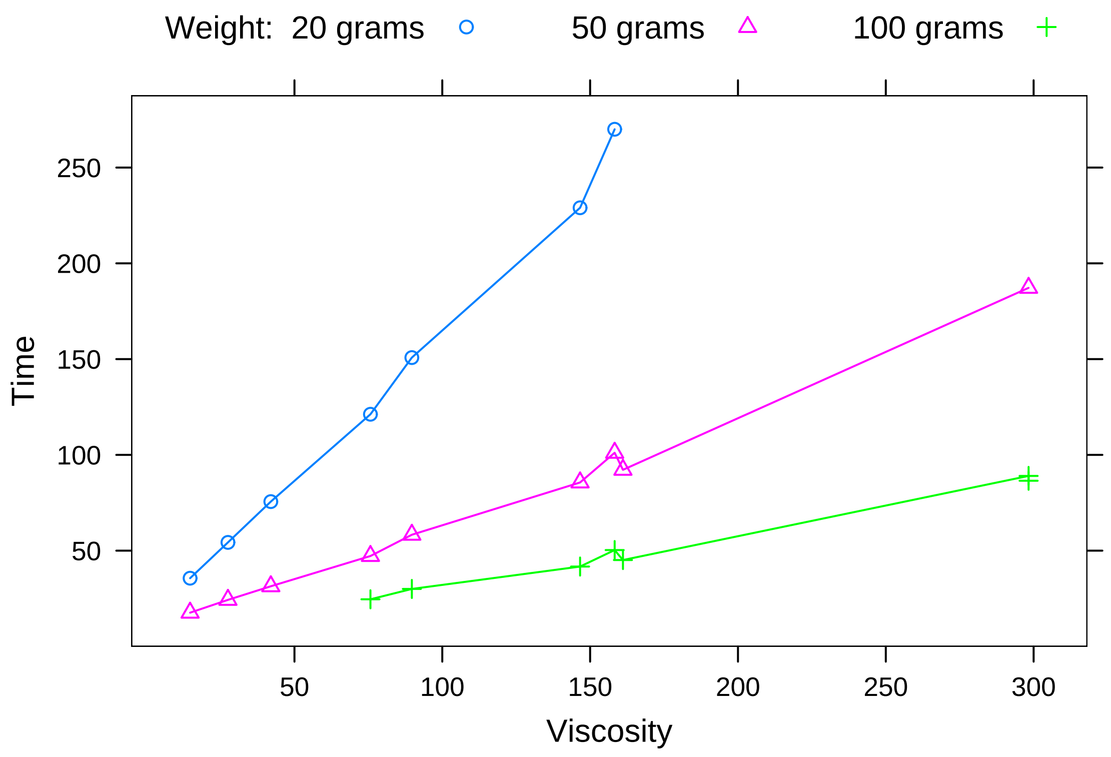
In the last class, the example showing the first use of `groups` didn't come out as I expected. The problem was that I reset the plotting symbols before opening the `trellis.device` i.e.,

```
sps <- trellis.par.get("superpose.symbol")
sps$pch <- 1:7
trellis.par.set("superpose.symbol",sps)
trellis.device("postscript", file="../stormer.eps",
  width=9, height=5, horiz=F, col=T)
```

It should have been the other way, i.e.,

```
trellis.device("postscript", file="../stormer.eps",
  width=9, height=5, horiz=F, col=T)
sps <- trellis.par.get("superpose.symbol")
sps$pch <- 1:7
trellis.par.set("superpose.symbol",sps)
```

The desired plot looks like this



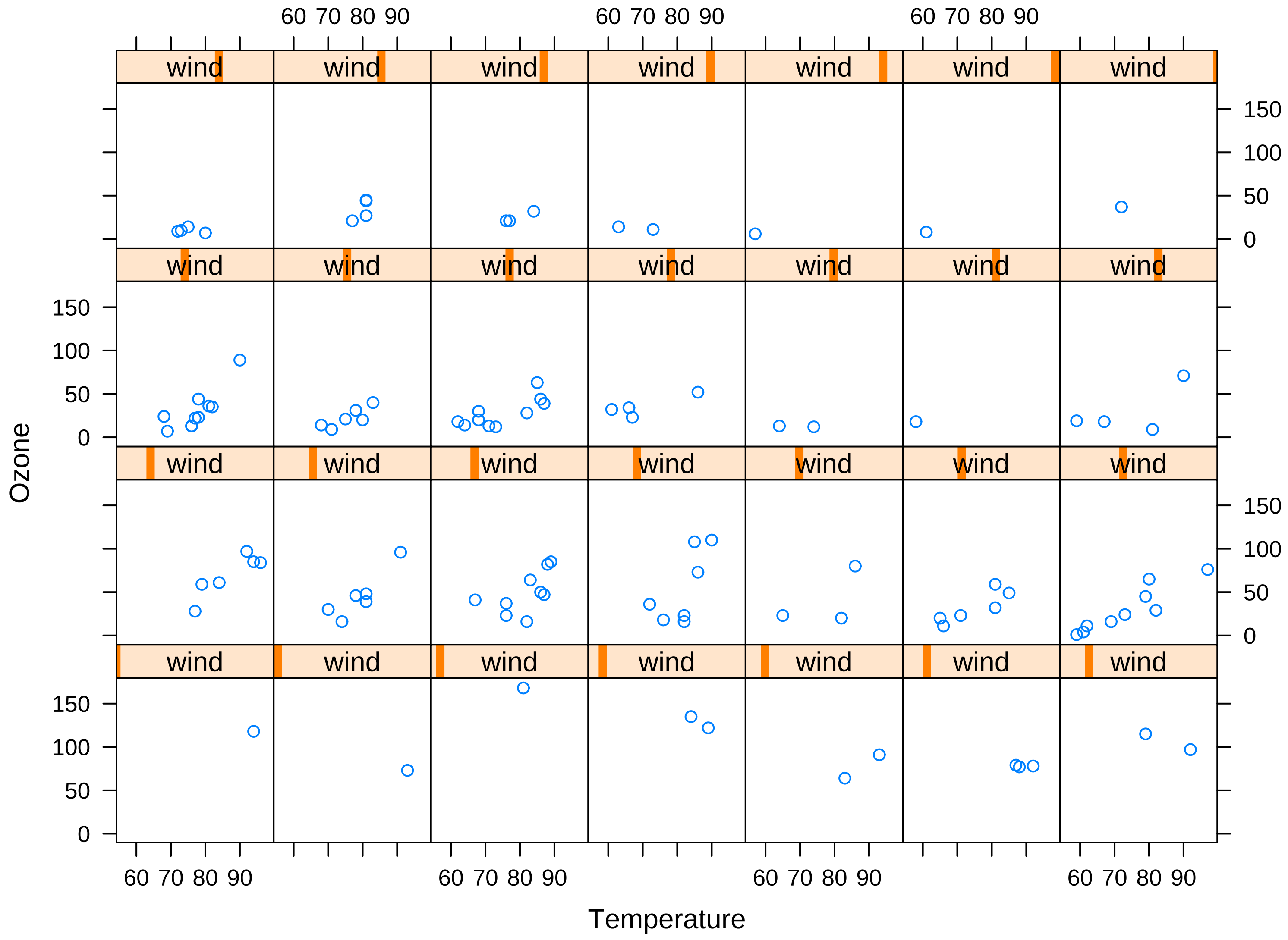
Conditioning on numeric variables

Lets take a look at the dataset from the `lattice` package, `environmental`, which has 111 measurements on 4 variables, ozone, temperature, wind speed, and solar radiation.

```
xyplot(ozone ~ temperature | wind, data=environmental,
  ylab="Ozone",xlab="Temperature", layout=c(7,4))

> length(unique(environmental$wind))
[1] 28
```

In this data set, there are 28 different wind speeds among the 111 observations.


wind
Ozone
Temperature
60 70 80 90
0
50
100
150

For most panels, there are very few observations that can be plotted. What would be more useful is to combine observations with similar wind speeds, so there are fewer panels, but with more observations per panel. This can be done by creating a shingle.

There are two approaches for creating a shingle. The simplest is `equal.count(x, number, overlap)`, which takes the data in vector `x`, and creates `number` intervals, such that each interval has roughly the same number of observations. The option `overlap` allows for overlap of intervals.

For example, to create a shingle with 6 intervals with an overlap of 20%, we can use

```
> Wind <- equal.count(environmental$wind, number=6, overlap=0.2)
> Wind

Data:
  [1]  7.4  8.0 12.6 11.5  8.6 13.8 20.1  9.7  9.2 10.9 13.2
 . . .
[106] 10.3 16.6  6.9 14.3  8.0 11.5

Intervals:
    min   max count
1  2.05  7.15    24
2  6.65  8.25    22
3  7.75  9.95    25
4  9.45 11.75    35
5 10.65 14.05    27
6 12.35 20.95    23

Overlap between adjacent intervals:
[1]  6  7  9 16  7
```

It is also possible to create a shingle with user specified interval with the function `shingle(x, intervals)` where `intervals` is a k by 2 matrix with the endpoints of the intervals.

For example

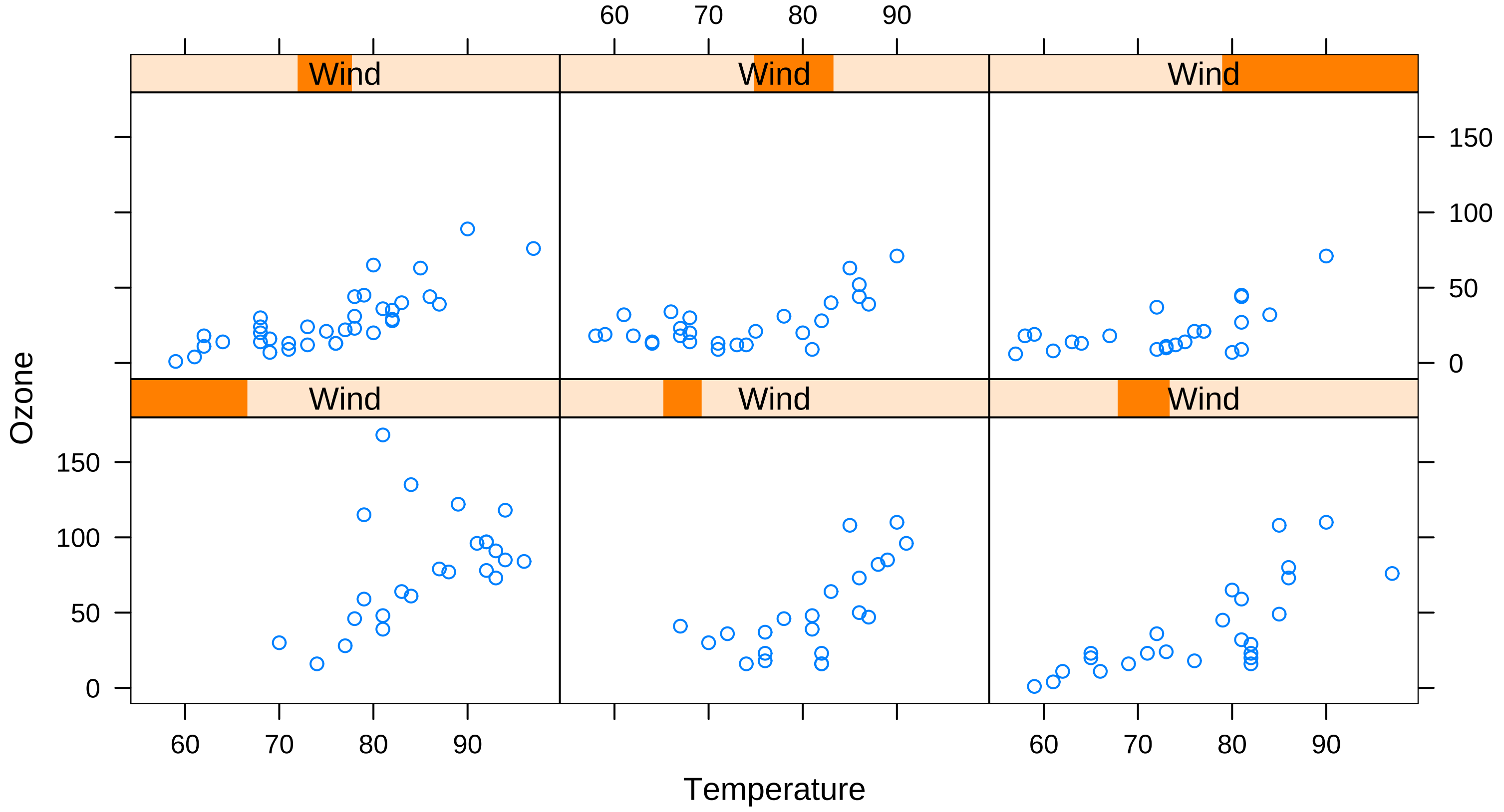
```
> wind.int <- matrix(c(2,6, 5,10, 8,15, 15,21), ncol=2, byrow=T)
> Wind2 <- shingle(environmental$wind, intervals=wind.int)
> Wind2   # Data part of output deleted

Intervals:
  min max count
1   2   6    12
2   5  10    51
3   8  15    70
4  15  21     8

Overlap between adjacent intervals:
[1]  5 25  0
```

So instead of conditioning on a real-valued variable, we can condition on the shingle instead

```
xyplot(ozone ~ temperature | Wind, data=environmental,
  ylab="Ozone",xlab="Temperature")
```



Controlling Presentation of Trellis Graphs

The trellis plot functions actually create complicated objects, a feature that we haven't used so far. One important fact is that they can be saved as any **S** object can. For example, graphical summaries of the two shingles discussed earlier can be saved by

```
wind.plt <- plot(Wind, aspect=0.4)
wind2.plt <- plot(Wind2, aspect=0.4)
```

Instead of being displayed in the current trellis device, they are saved as objects.

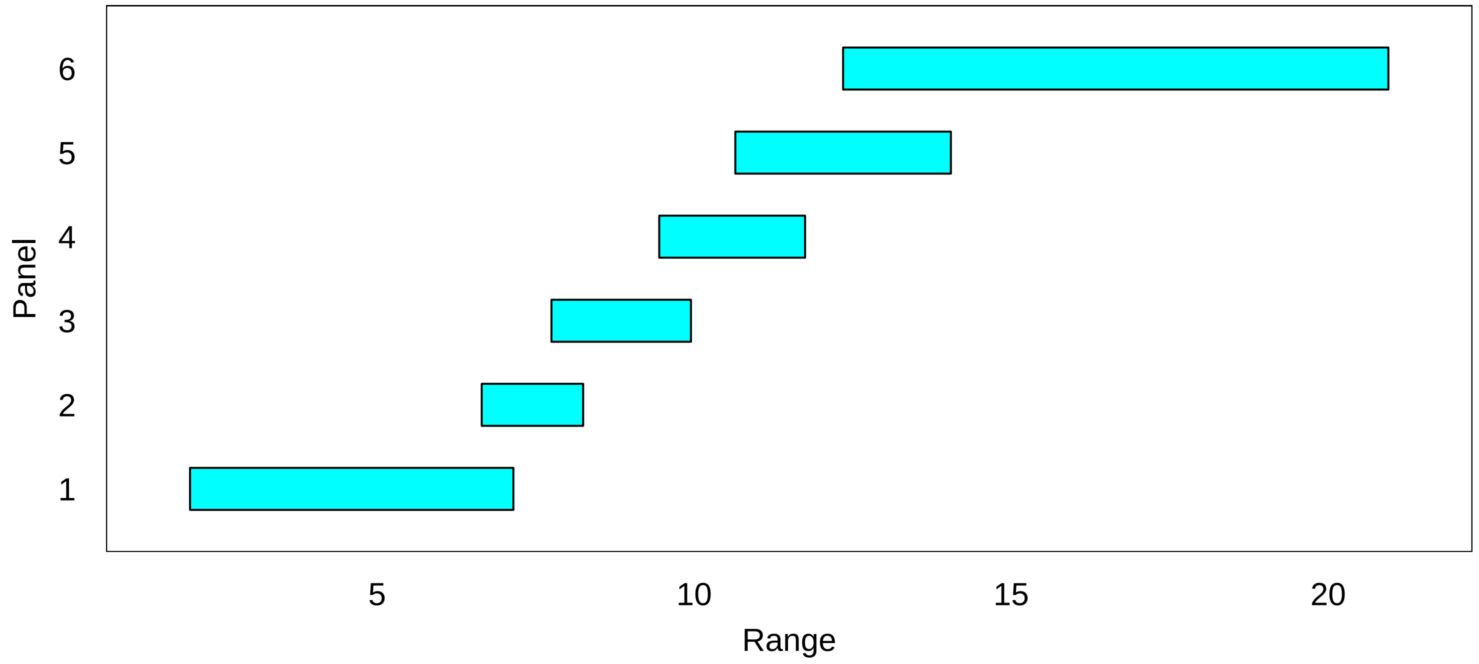
This allows for some nice features.

First it allows a graph to be created once, but written to a number of trellis devices. For example

```
trellis.device(windows, theme=col.whitebg)
wind.plt
```

```
trellis.device("postscript", file="../windshingle.eps",
  width=8, height=4, horiz=F, col=T)
wind.plt   # equivalently print(wind.plt)
dev.off()
```

first displays the figure on the screen and then writes it to the the following postscript file



Second it allows for trellis graphs to be updated with `update` function. (I told a fib earlier saying trellis figures can't be updated.) For example

```
wind.plt.up <- update(wind.plt,
      main="equal.count() created shingle for wind")
wind2.plt.up <- update(wind2.plt,
      main="shingle() created shingle for wind")
```

which adds titles to each shingle plot. This is mainly useful for changing, titles or labels, font or symbol sizes, etc.

The third, and maybe most useful, is that it allows for multiple plots to be combined into a single figure. This can be done with the `print` function (actually the `print.trellis` method). One form of the function is

```
     print(graph, split=c(col, row, ncol, nrow), more)
```

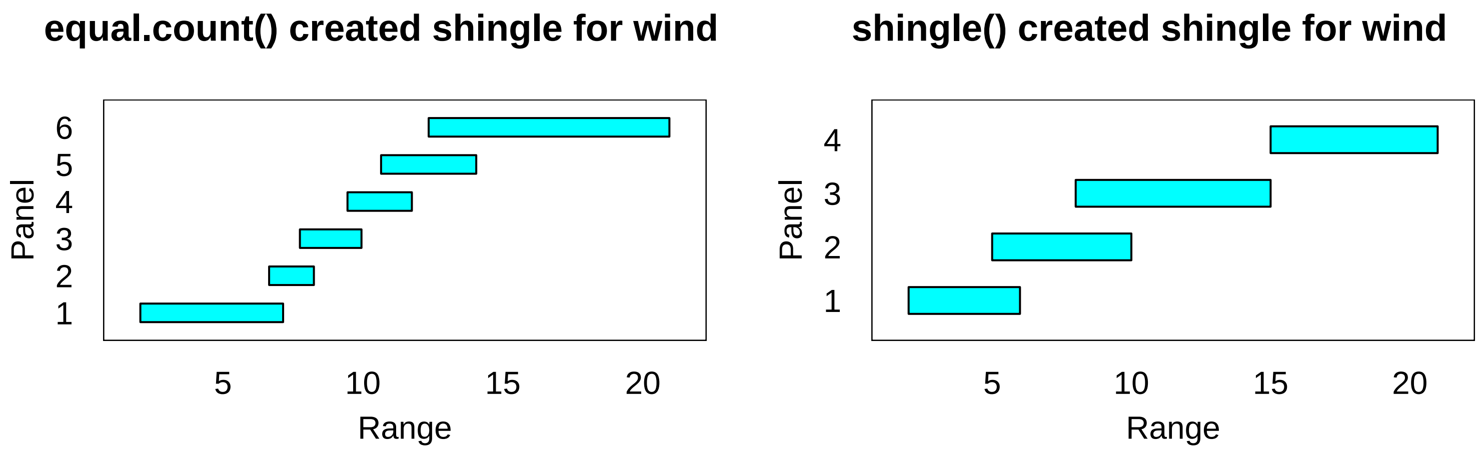
where `graph` is a stored trellis plot, `split` indicates where to place the plot, and `more` is a logical indicating whether more plots are to be added.

Note Krause and Olsen get the `split` description wrong, at least for **R**.

For example, the two shingle plots can be combined by

```
trellis.device("postscript", file="../windshingle2.eps",
  width=8, height=6, horiz=F, col=T)
print(wind.plt.up, split=c(1,1,2,1), more=T)
print(wind2.plt.up, split=c(2,1,2,1), more=F)
dev.off()
```

yielding



There is a second approach which allows for different size plots in a figure or for plots not to have to occur in a standard rectangular grid. This can be done by using the `position` option instead of `split` in the print function. This form of the function is

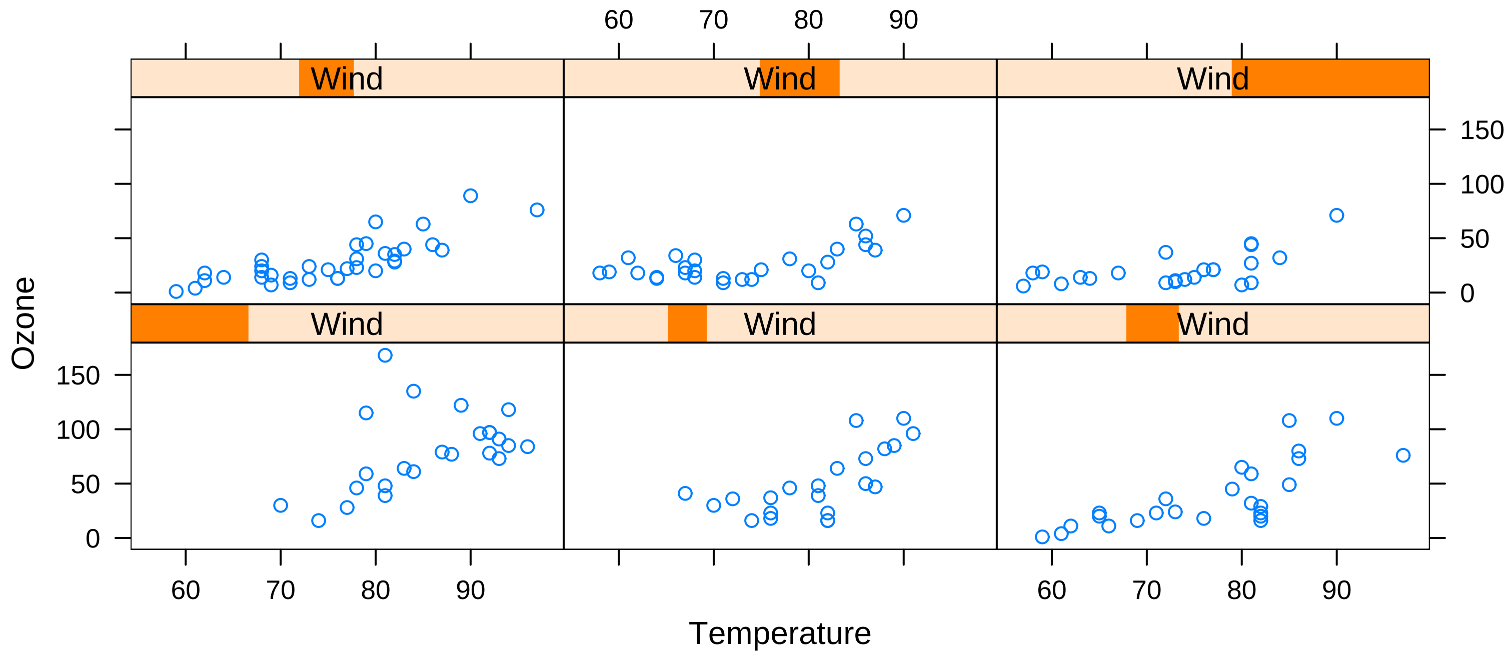
```
     print(graph, position=c(xll, yll, xur, yur), more)
```

where `(xll, yll)` are the coordinates of the lower left corner of the plot and `(xur, yur)` are the coordinates of the upper right corner of the plot.

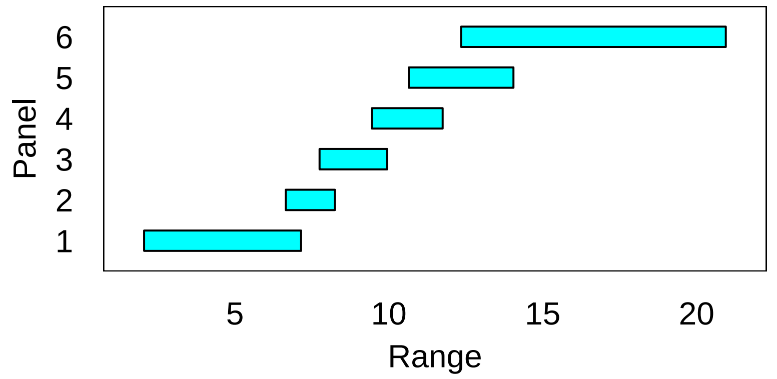
For example

```
environ.plt <- xyplot(ozone ~ temperature | Wind,
  data=environmental, ylab="Ozone",xlab="Temperature")
trellis.device("postscript", file="../environ3.eps",
  width=8, height=6, horiz=F, col=T)
print(environ.plt, position=c(0, 0.4, 1, 1), more=T)
print(wind.plt.up, position=c(0, 0, 1, 0.4), more=F)
dev.off()
```

gives



equal.count() created shingle for wind



One default that I don't like is that for many trellis devices (including `windows` and `win.metafile`) is for a gray background. To get a white background, one option is to set `theme = col.whitebg` when opening the trellis device. Note that this option will change some of the colours used in the plot. They are chosen to look good on a white background. These can be reset by playing with `trellis.par.set()` options.

Setting the theme can be done with any trellis device, including `postscript` which doesn't need it (the earlier `stripplot` example used it) . To change the background for on screen display, you must explicitly open a plot window with `trellis.device(theme=col.whitebg)`.

Lets compare

```
trellis.device(width=9, height=6)
print(environ.plt)
```

with

```
trellis.device(theme=col.whitebg, width=9, height=6)
print(environ.plt)
```