Lattice With Layers – `layer()` and `as.layer()` (from *latticeExtra*)[1]

J H Maindonald
http://www.maths.anu.edu.au/~johnm

Centre for Mathematics and Its Applications
Australian National University.

Contents

The layering features of the *latticeExtra* package, due to Felix Andrews, appeared subsequent to the final draft of Maindonald & Braun (2010), They are a powerful and useful supplement to the abilities of the *lattice* package. Examples are given that demonstrate the layering approach, in several instances offering for comparison alternatives that achieve the same end result.

1 Layering, and Alternatives to Layering

Each lattice command that creates a graph has its own panel function. Perhaps the most obvious way to gain detailed control of panel contents is to create one's own panel function, which in the simplest case could be a sequence of calls to several of the variety of panel functions that are available in *lattice*. Or the panel function can be modified in a call to the function `update()`.

The `layer()`, `as.layer()` and related functions in the *latticeExtra* package add further flexibility. Both add a new "layer" to a graphics object.

- The function `layer()` allows as arguments, passed via the `...` argument, any sequence of statements that might appear in a panel function. A `data` argument can be supplied from which new variables can be taken.

Current draft: February 17, 2011

- The function `as.layer()` makes it possible to superpose the contents of the panels of two or more lattice objects.

Several variations on `layer()` are available. These are `layer_()`, `glayer()` and `glayer_()`. Use `glayer()` when account must be taken of groups within each panel. The functions `layer_()` and `glayer_()` add the new layer "underneath" the prior layer.

First attach the `DAAG` and `latticeExtra` packages. The `lattice` package will be attached automatically along with `latticeExtra`.

```
> library(latticeExtra)
> library(DAAG)
```

Panel functions

The function `xyplot()` calls, by default, the panel function `panel.xyplot()`. The following are equivalent:

```
> xyplot(species ~ length, xlab="", data=cuckoos)
> xyplot(species ~ length, xlab="", data=cuckoos, panel=panel.xyplot)
```

NB: When a `groups` argument is supplied, `panel.xyplot()` calls `panel.superpose()`.

The user's own function can be substituted for `panel.xyplot()`. Such a function might for example call `panel.superpose()`, followed or preceded by other available panel functions. These include:

- `panel.points()` (alias `lpoints()`), `panel.lines()`, `panel.text()`, `panel.rect()`, `panel.arrows()`, `panel.segments()`, `panel.polygon()`
  (these are all documented on the same help page as `panel.points()`)

- `panel.abline()`, `panel.curve()`, `panel.loess()`, `panel.lmline()`, `panel.rug()`, `panel.refline()`, `panel.average()`, `panel.fill()`, `panel.mathdensity()`
  (these are all documented on the same help page as `panel.abline()`).

Note the aliases `lpoints()` for `panel.points()`, `llines()` for `panel.lines()`, etc.);

The following compares alternative approaches to modifying panel contents:

```
> library(DAAG)
> gph <- xyplot(Brainwt ~ Bodywt,  data=primates, xlim=c(0,280))
```

For adding labels to the points, two possibilities are:

- Use the function `layer()` to add a second layer that has the labels, thus:

  ```
  > gph + layer(panel.text(x,y, labels=rownames(primates), pos=4))
  ```

- Create a panel function that plots both points and labels, then using this to update the graphics object:

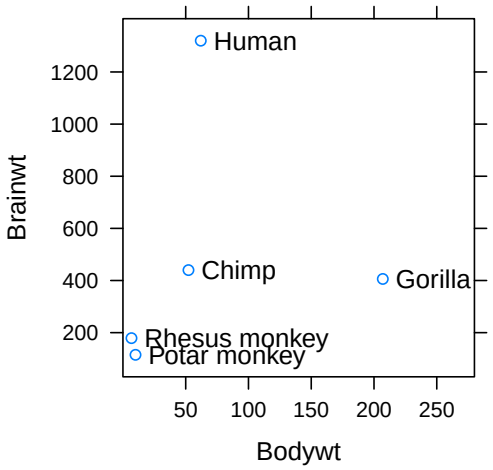

Figure 1: Graph, with points labeled.

```
> ## Code that adds layer to basic graph
> ## Code that adds a further layer
> ## Alternative to code in the text
> gph <- xyplot(Brainwt ~ Bodywt,
                data=primates,
                xlim=c(0,280))
> addlabels <-
    function(x,y,labs=rownames(primates))
      panel.text(x,y, labels=labs, pos=4)
> print(gph + layer(addlabels(x,y)))
```

```
> my.panel <- function(x,y){
      panel.xyplot(x,y)
      panel.text(x,y, labels=rownames(primates), pos=4)}
> update(gph, panel=my.panel)
```

Code that accompanies Figure 1 shows variants on the above code, in which functions are created outside the call to `layer()` or to `update()`.

Note finally that we could have supplied the argument `panel=my.panel` in a single function call, thus:

```
> xyplot(Brainwt ~ Bodywt, data=primates, xlim=c(0,280), panel=my.panel)
```

However the plot is obtained, Figure 1 shows the result.

Adding to conditioning plots

The following show haemoglobin count versus red blood cell count, distinguished by `sport` within panel, and with separate panels for females and males:

```
> aisBS <- subset(ais, sport %in% c("B_Ball", "Swim"))
> basic1 <-  xyplot(hc ~ rcc | sex, groups=sport[drop=TRUE],
                    data=aisBS)
> basic2 <- update(basic1,
      par.settings=simpleTheme(pch = c(1,3)),
      strip=strip.custom(factor.levels=c("Female","Male")),
      # In place of level names c("f", "m"),
      # use c("Female", "Male")
      scales=list(tck=0.5), lty=1:2, lwd=1.5)
```

The `xyplot()` function has provision for the addition of separate lines for the two sports, but not for parallel lines. Subsection 15.5.3 in Maindonald & Braun (2010) updates the graphics object to use a specially created panel function.

The following, which adds a new layer that has the parallel lines, is simpler. First, create the new layer:

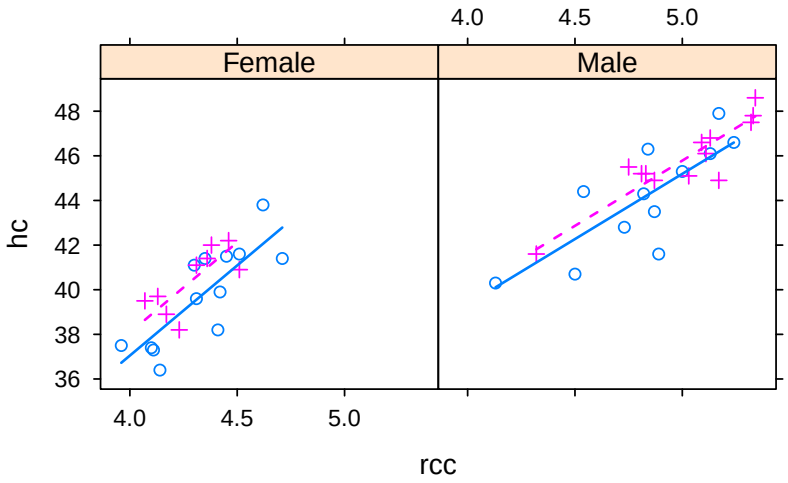

Figure 2: Lines that are parallel between sports have been added to the basic plot, as described in the text.

```
> ## Create new layer that has the parallel lines
> layer2 <-
      layer(parallel.fit <-
            fitted(lm(y ~ groups[subscripts] + x)),
            panel.superpose(x, parallel.fit, type = "a", ...))
```

Now plot the graph

```
> print(basic2+layer2)
```

Figure2 shows the plot that is required:
The following adds a new layer:

Observe that the arguments to the function `layer()` were the two commands:

```
> parallel.fit <- fitted(lm(y ~ groups[subscripts] + x))
> panel.superpose(x, parallel.fit, type = "a", ...)
```

These are executed in succession, adding the new layer to the plot. More generally, any number of arguments, each an R command, can be supplied.

2 Overlaying Spatial Data

The data frame `meuse` (*sp* package) has data from locations near the village of Stein in the Netherlands, in the floodplain of the river Meuse. Included are concentrations of various heavy metals (`cadmium`, `copper`, `lead`, `zinc`), with `x` and `y` giving Eastings and Northings in Netherlands topographical map coordinates.

The *sp* package has a function `bubble()` that uses the abilities of the *lattice* package to plot spatial measurement data. Each point (location) is shown as a bubble. By default, the area of the bubble is proportional to the value. The description of the argument `obj` looks intimidating. There is however a function `coordinates()` that can be used to turn a data frame or matrix into an object of the requisite class. Here, it will create an object of class `SpatialPointsDataFrame`.

```
> library(sp)
> data(meuse)
> coordinates(meuse) <- ~ x + y
```

The coordinates can be extracted using `coordinates(meuse)`. Remaining columns of the data are available from the data frame `meuse@data`.

Figure 3 shows a bubble plot for `zinc`.

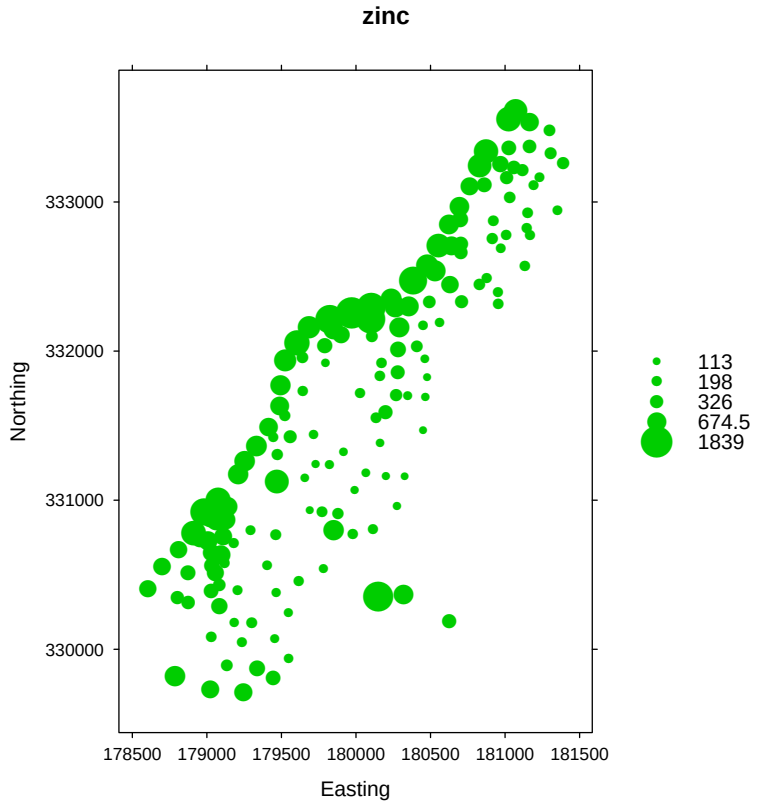

Figure 3: Bubble plot for `zinc` concentrations. Areas of bubbles are proportional to concentrations. Locations are near the village of Stein (Netherlands), in the floodplain of the river Meuse.

Code is:

```
> ## Code
> ## library(sp)
> ## data(meuse)
> ## coordinates(meuse) <- ~ x + y
> library(lattice)
> gph <- bubble(meuse, zcol="zinc", scales=list(tck=0.5),
                xlab="Easting", ylab="Northing")
> print(gph)
```

As the function `bubble()` uses the abilities of the *lattice* package, the layering abilities of the *latticeExtra* package, described earlier in Subsection 1. can be used to overlay additional information on the plot.

Figure 4 adds the river boundaries, using data from the dataset `meuse.riv`. This is a matrix, with Eastings in column 1 and Northings in column 2.

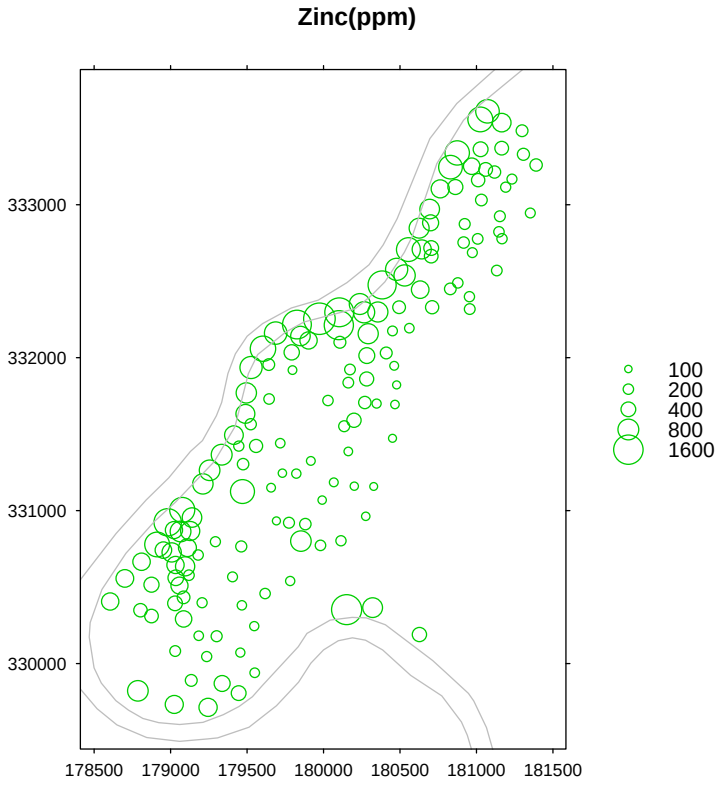

Figure 4: Bubble plot for `zinc`, with area of bubbles proportional to concentration. River Meuse boundaries are in gray.

Code is:

```
> data(meuse); data(meuse.riv)
> coordinates(meuse) <- ~ x + y
> library(latticeExtra)
> gph <- bubble(meuse, "zinc", pch=1, key.entries =  100 * 2^(0:4),
                main = "Zinc(ppm)", scales=list(axes=TRUE, tck=0.4)) +
  layer(panel.lines(meuse.riv[,1], meuse.riv[,2], col="gray"))
> print(gph)
```

3 Adding to a dotplot

The graph will add error bars to a basic dotplot. The data are:

```
> numsDF <- data.frame(
    species=rep(c("Jimajing teresaeta", "Sasquatch rutilusaeta"), c(3,3)),
    bcat=rep(c("<2 y", "5-10 y", ">20 y"), 2),
    estlog=c(0.286, 1.660, -0.616, 2.488, 3.689, 2.290),
    halfwid=rep(c(0.198, 0.464), c(3,3)))
> numsDF$bcat <- factor(numsDF$bcat, levels=c("<2 y", "5-10 y", ">20 y"))
> ## Check the data
> print(numsDF, quote=FALSE)
```

```
                species   bcat estlog halfwid
1    Jimajing teresaeta   <2 y  0.286   0.198
2    Jimajing teresaeta 5-10 y  1.660   0.198
3    Jimajing teresaeta  >20 y -0.616   0.198
4 Sasquatch rutilusaeta   <2 y  2.488   0.464
5 Sasquatch rutilusaeta 5-10 y  3.689   0.464
6 Sasquatch rutilusaeta  >20 y  2.290   0.464
```

Data are on a logarithmic ($\log_e$) scale. The following generates the numbers that will be plotted:

```
> numsDF <- within(numsDF, {
        estnum=exp(estlog)
        lolim=exp(estlog-halfwid)
        hilim=exp(estlog+halfwid)
  }
  )
```

First, create a graphics object that displays the estimated numbers.

```
> library(latticeExtra)
> uplim <- with(numsDF, sapply(split(hilim, species), max))
> gph <- dotplot(bcat ~ estnum | species, subscripts=TRUE,
                 scales=list(alternating=FALSE,relation="free"),
                 xlim=list(c(0, uplim[1]), c(0, uplim[2])),
                 data=numsDF,
                 layout=c(1,2))
```

Notice the use of the argument `subscripts=TRUE` in the call to `dotplot()`. The subscript information, identifying the panels in which points are plotted, will be required when a new layer is added.

Now create a layer that will be used to add the limits around the points:

```
> addlayer <- layer(panel.segments(lolim[subscripts], y,
                                   hilim[subscripts], y),
                    data=numsDF)
```

The argument `y` refers to the panel-specific y-values for the initial dotplot. On the other hand, the arguments `lolim` and `hilim` are taken from the data frame, and hence must be subscripted.

Then plot the graph using the command:

```
> print(gph+addlayer)
```

Figure 5 shows the result:

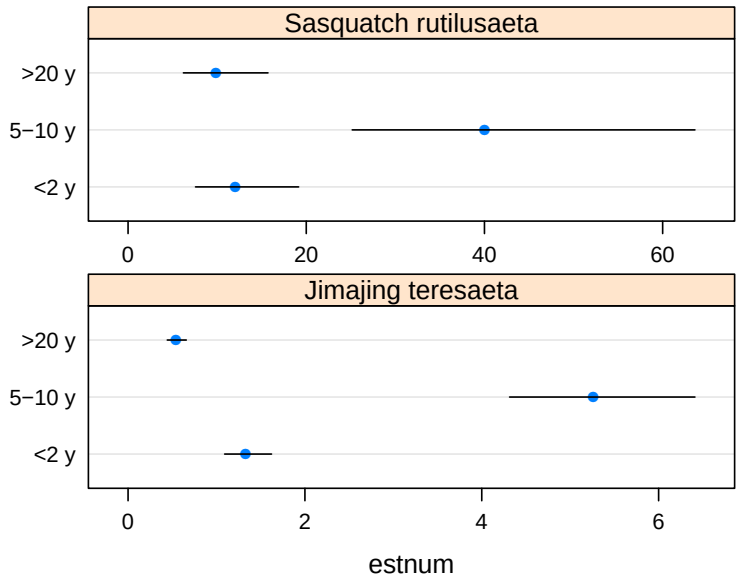

Figure 5: The limits about each point were added as a second layer to a graph that used the `dotplot()` function.

One function to do all – `segplot()`

The *latticeExtra* package has the function `segplot()` that does the complete task in one function call:

```
> segplot(bcat ~ lolim+hilim | species, centers=estnum, data=numsDF,
          scales=list(x="free"), cex=1.5,
          draw.bands=FALSE, lwd=2,
          layout=c(1,2))
```

Note the use of the argument `draw.bands=FALSE`. With the default `draw.bands=TRUE`, the lines that show the limits are replaced by rectangles.

References

Maindonald, J. H. and Braun, W.J. 2010. *Data Analysis and Graphics Using R – An Example-Based Approach*, 3rd edition, Cambridge University Press.
<URL:http://www.maths.anu.edu.au/~johnm/r-book.html>