Analysing Spatial Data in R: Vizualising Spatial Data

Roger Bivand

Department of Economics
Norwegian School of Economics and Business Administration
Bergen, Norway

31 August 2007

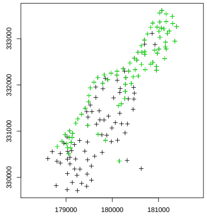
Just spatial objects

- There are base graphics plot methods for the key `Spatial*` classes, including the `Spatial` class, which just sets up the axes
- In base graphics, additional plots can be added by overplotting as usual, and the `locator()` and `identify()` functions work as expected
- In general, most `par()` options will also work, as will the full range of graphics devices (although some copying operations may disturb aspect)
- First we will display the positional data of the objects discussed in the first unit

Vizualising Spatial Data

- Displaying spatial data is one of the chief reasons for providing ways of handling it in a statistical environment
- Of course, there will be differences between analytical and presentation graphics here as well — the main point is to get a usable display quickly, and move to presentation quality cartography later
- In general, maintaining aspect is vital, and that can be done in both base and lattice graphics in R (note that both **sp** and **maps** display methods for spatial data with geographical coordinates "stretch" the y-axis)
- We'll look at the basic methods for displaying spatial data in **sp**; other packages have their own methods, but the next unit will show ways of moving data from them to **sp** classes

Plotting a `SpatialPoints` object

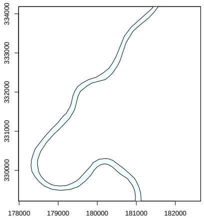


While plotting the SpatialPoints object would have called the plot method for Spatial objects internally to set up the axes, we start by doing it separately:

```
> plot(as(meuse1, "Spatial"),
+     axes = TRUE)
> plot(meuse1, add = TRUE)
> plot(meuse1[meuse1$ffreq ==
+     1, ], col = "green", add = TRUE)
```

Then we plot the points with the default plotting character, and subset, overplotting points in flood frequency class 1 in green, using the [method

Plotting a `SpatialPolygons` object



In plotting the SpatialPolygons object, we use the ylim= argument to restrict the display area to match the soil sample points.

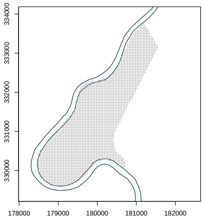
```
> plot(rivers, axes = TRUE, col = "azure1",
+     ylim = c(329400, 334000))
> box()
```

If the axes= argument is FALSE or omitted, no axes are shown — the default is the opposite from standard base graphics plot methods

Including attributes

- To include attribute values means making choices about how to represent their values graphically, known in some GIS as symbology
- It involves choices of symbol shape, colour and size, and of which objects to differentiate
- When the data are categorical, the choices are given, unless there are so many different categories that reclassification is needed for clear display
- Once we've looked at some examples, we'll go on to see how class intervals may be chosen for continuous data

Plotting a `SpatialPixels` object

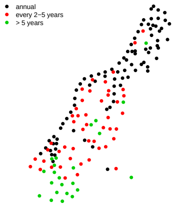


Both SpatialPixels and SpatialGrid objects are plotted like SpatialPoints objects, with plotting characters

```
> plot(rivers, axes = TRUE, col = "azure1",
+     ylim = c(329400, 334000))
> box()
> plot(meuseg1, add = TRUE, col = "grey60",
+     cex = 0.15)
```

While points, lines, and polygons are often plotted without attributes, this is rarely the case for gridded objects

Flood frequencies at soil sample sites

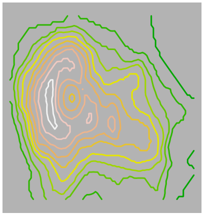


We will usually need to get the category levels and match them to colours (or plotting characters) "by hand"

```
> meuse1$ffreq1 <- as.numeric(meuse1$ffreq)
> plot(meuse1, col = meuse1$ffreq1,
+     pch = 19)
> labs <- c("annual", "every 2-5 years",
+     "> 5 years")
> cols <- 1:nlevels(meuse1$ffreq)
> legend("topleft", legend = labs,
+     col = cols, pch = 19, bty = "n")
```

It is also typical that the legend() involves more code than everything else together, but very often the same vectors are used repeatedly and can be assigned just once

Coloured contour lines



Here again, the values are represented as a categorical variable, and so do not require classification

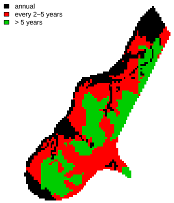
```
> volcano_sl$level1 <- as.numeric(volcano_sl$level)
> pal <- terrain.colors(nlevels(volcano_sl$level))
> plot(volcano_sl, bg = "grey70",
+     col = pal[volcano_sl$level1],
+     lwd = 3)
```

Using class membership for colour palette look-up is a very typical idiom, so that the col= argument is in fact a vector of colour values

Class intervals

- Class intervals can be chosen in many ways, and some have been collected for convenience in the **classInt** package
- The first problem is to assign class boundaries to values in a single dimension, for which many classification techniques may be used, including pretty, quantile, natural breaks among others, or even simple fixed values
- From there, the intervals can be used to generate colours from a colour palette, using the very nice `colorRampPalette()` function
- Because there are potentially many alternative class memberships even for a given number of classes (by default from `nclass.Sturges`), choosing a communicative set matters

Displaying gridded data



Since we also have 40m grid flood frequencies, we can try to display them — here we use the image() method, which first fills in the NAs, the makes a matrix of the chosen variable

```
> meuseg1$ffreq1 <- as.numeric(meuseg1$ffreq)
> image(meuseg1, "ffreq1", col = cols)
> legend("topleft", legend = labs,
+     fill = cols, bty = "n")
```

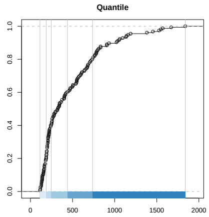
Some of the arguments here are like those we'll meet soon for lattice graphics

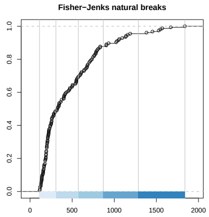
Class intervals

We will try just two styles, quantiles and Fisher-Jenks natural breaks for five classes, among the many available. They yield quite different impressions, as we will see:

```
> library(classInt)
> library(RColorBrewer)
> pal <- brewer.pal(3, "Blues")
> q5 <- classIntervals(meuse1$zinc, n = 5, style = "quantile")
> q5

style: quantile
  one of 14,891,626 possible partitions of this variable into 5 classes
  under 186.8 186.8 - 246.4 246.4 - 439.6 439.6 - 737.2    over 737.2
           31             31             31             31            31

> fj5 <- classIntervals(meuse1$zinc, n = 5, style = "fisher")
> fj5

style: fisher
  one of 14,891,626 possible partitions of this variable into 5 classes
   under 307.5  307.5 - 573.0  573.0 - 869.5 869.5 - 1286.5
            75             32             29             12
   over 1286.5
             7

> plot(q5, pal = pal)
> plot(fj5, pal = pal)
```

Class interval plots

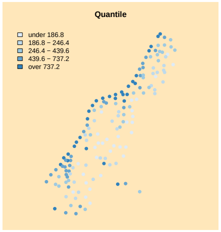


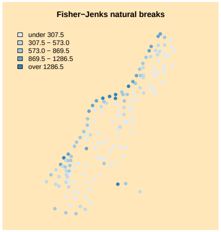


Lattice graphics

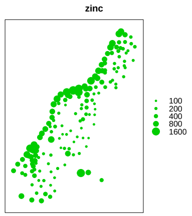
- Lattice graphics will only come into their own later on, when we want to plot several variables with the same scale together for comparison
- The workhorse method is `spplot`, which can be used as an interface to the underlying `xyplot` or `levelplot` methods, or others as suitable; overplotting must be done in the single call to `spplot` — see gallery
- It is often worthwhile to load the **lattice** package so as to have direct access to its facilities
- Please remember that lattice graphics are displayed on the current graphics device by default only in interactive sessions — in loops or functions, they must be explicitly `print`'ed

Two versions of zinc ppm





Bubble plots

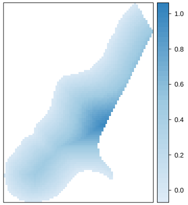


Bubble plots are a convenient way of representing the attribute values by the size of a symbol

```
> library(lattice)
> bubble(meuse1, "zinc", maxsize = 2,
+     key.entries = 100 * 2^(0:4))
```

As with all lattice graphics objects, the function can return an object from which symbol sizes can be recovered

Level plots



The use of lattice plotting methods yields easy legend generation, which is another attraction

```
> bpal <- colorRampPalette(pal)(41)
> spplot(meuseg1, "dist", col.regions = bpal,
+     cuts = 40)
```

Here we are showing the distances from the river of grid points in the study area; we can also pass in intervals chosen previously

More realism

- So far we have just used canned data and spatial objects rather than anything richer
- The vizualisation methods are also quite flexible — both the base graphics and lattice graphics methods can be extensively customised
- It is also worth recalling the range of methods available for **sp** objects, in particular the `overlay` and `spsample` methods with a range of argument signatures
- These can permit further flexibility in display, in addition to their primary uses