3

Visualising Spatial Data

A major pleasure in working with spatial data is their visualisation. Maps are amongst the most compelling graphics, because the space they map is the space we think we live in, and maps may show things we cannot see otherwise. Although one can work with all R plotting functions on the raw data, for example extracted from `Spatial` classes by methods like `coordinates` or `as.data.frame`, this chapter introduces the plotting methods for objects inheriting from class `Spatial` that are provided by package **sp**.

R has two plotting systems: the 'traditional' plotting system and the Trellis Graphics system, provided by package **lattice**, which is present in default R installations (Sarkar, 2008). The latter builds upon the 'grid' graphics model (Murrell, 2006). Traditional graphics are typically built incrementally: graphic elements are added in several consecutive function calls. Trellis graphics allow plotting of high-dimensional data by providing *conditioning plots*: organised lattices of plots with shared axes (Cleveland, 1993, 1994). This feature is particularly useful when multiple maps need to be compared, for example in case of a spatial time series, comparison across a number of species or variables, or comparison of different modelling scenarios or approaches. Trellis graphs are designed to avoid wasting space by repetition of identical information. The value of this feature, rarely found in other software, is hard to overestimate.

Waller and Gotway (2004, pp. 68–86) provide an introduction to statistical mapping, which may be deepened with reference to Slocum et al. (2005).

Package **sp** provides `plot` methods that build on the traditional R plotting system (`plot`, `image`, `lines`, `points`, etc.), as well as a 'new' generic method called `spplot` that uses the Trellis system (notably `xyplot` or `levelplot` from the **lattice** package) and can be used for conditioning plots. The `spplot` methods are introduced in a later sub-section, first we deal with the traditional plot system.

3.1 The Traditional Plot System

3.1.1 Plotting Points, Lines, Polygons, and Grids

In the following example session, we create points, lines, polygons, and grid object, from `data.frame` objects, retrieved from the **sp** package by function `data`, and plot them. The four plots obtained by the `plot` and `image` commands are shown in Fig. 3.1.

```
> library(sp)
> data(meuse)
```

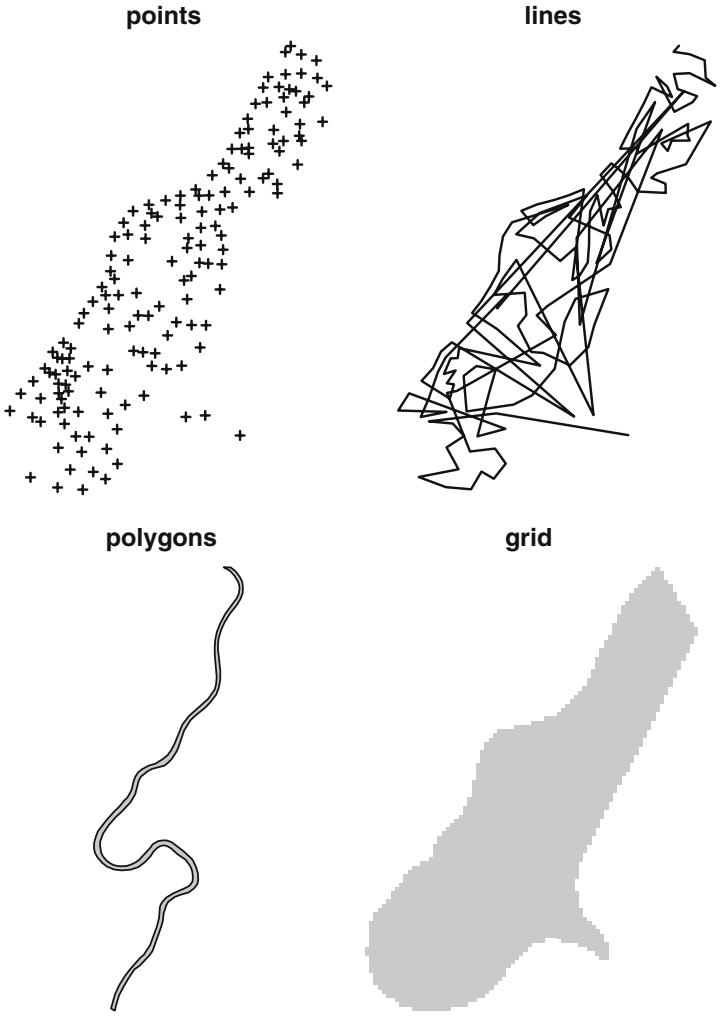

Fig. 3.1. The meuse data set: sample points, the sample path (*line*), the Meuse river (*ring*), and the gridded study area

```
> coordinates(meuse) <- c("x", "y")
> plot(meuse)
> title("points")
```

The SpatialPointsDataFrame object used is created from a data.frame provided with **sp**, and the plot method shows the points with the default symbol.

```
> cc <- coordinates(meuse)
> m.sl <- SpatialLines(list(Lines(list(Line(cc)))))
> plot(m.sl)
> title("lines")
```

A SpatialLines object is made by joining up the points in sequence, and plot draws the resulting zig-zags.

```
> data(meuse.riv)
> meuse.lst <- list(Polygons(list(Polygon(meuse.riv)),
+     "meuse.riv"))
> meuse.sr <- SpatialPolygons(meuse.lst)
> plot(meuse.sr, col = "grey")
> title("polygons")
```

We make a SpatialPolygons object from data provided with **sp** outlining the banks of the River Meuse.

```
> data(meuse.grid)
> coordinates(meuse.grid) <- c("x", "y")
> meuse.grid <- as(meuse.grid, "SpatialPixels")
> image(meuse.grid, col = "grey")
> title("grid")
```

Finally, we convert grid data for the same Meuse bank study area into a 'SpatialPixels' object and display it using the image method, with all cells set to 'grey'.

On each map, one unit in the x-direction equals one unit in the y-direction. This is the default when the coordinate reference system is not longlat or is unknown. For unprojected data in geographical coordinates (longitude/latitude), the default aspect ratio depends on the (mean) latitude of the area plotted. The default aspect can be adjusted by passing the asp argument.

A map becomes more readable when we combine several elements. We can display elements from those created above by using the add = TRUE argument in function calls:

```
> image(meuse.grid, col = "lightgrey")
> plot(meuse.sr, col = "grey", add = TRUE)
> plot(meuse, add = TRUE)
```

the result of which is shown in Fig. 3.2.

The over-plotting of polygons by points is the consequence of the order of plot commands. Up to now, the plots only show the geometry (topology, shapes) of the objects; we start plotting attributes (e.g. what has actually been measured at the sample points) in Sect. 3.1.5.

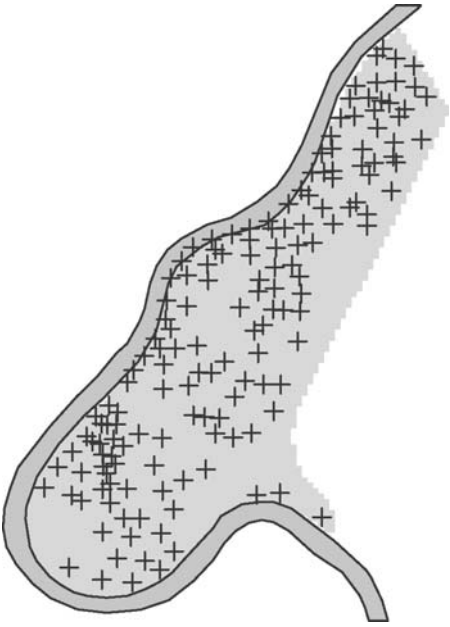

Fig. 3.2. Map elements combined into a single map

As an alternative to `plot(x,add=TRUE)`, one can use the commands `lines` for objects of class `SpatialLines` and `points` for `SpatialPoints`; text elements can be added by `text`.

3.1.2 Axes and Layout Elements

Maps often do not have axes, as the information carried in map axes can often be omitted. Especially, projected coordinates are usually long, hard to read and geographical reference is much easier when recognisable features such as administrative boundaries, rivers, coast lines, etc. are present. In the standard `plot` functions, the Boolean argument `axes` can be set to control axis plotting, and the function `axis` can be called to add axes, fine-tuning their appearance (tic placement, tic labels, and font size). The following commands result in Fig. 3.3:

```
> layout(matrix(c(1, 2), 1, 2))
> plot(meuse.sr, axes = TRUE)
> plot(meuse.sr, axes = FALSE)
> axis(1, at = c(178000 + 0:2 * 2000), cex.axis = 0.7)
> axis(2, at = c(326000 + 0:3 * 4000), cex.axis = 0.7)
> box()
```

Not plotting axes does not increase the amount of space R used for plotting the data.[1] R still reserves the necessary space for adding axes and titles later

[1] This is not true for Trellis plots; see Sect. 3.2.

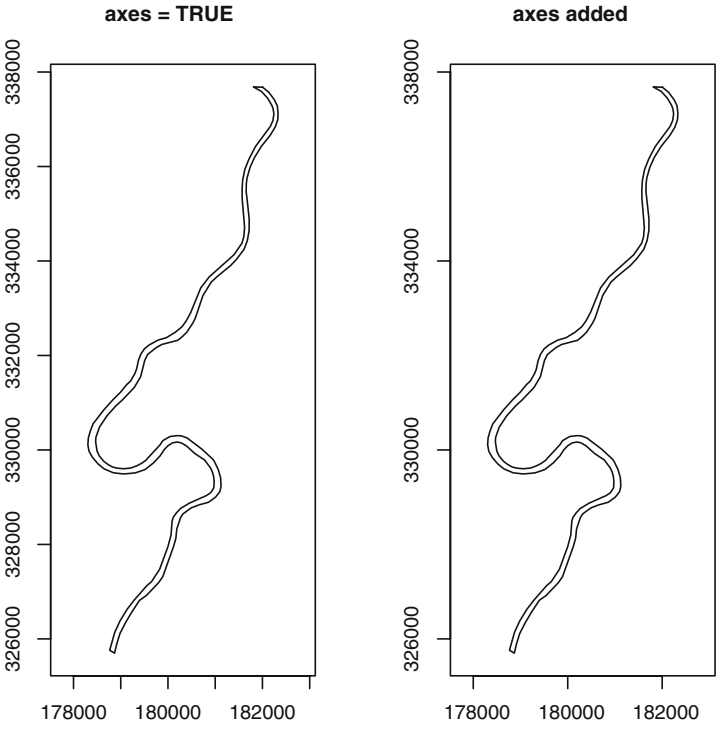

Fig. 3.3. Default axes (*left*) and custom axes (*right*) for the `meuse.riv` data

Table 3.1. Graphic arguments useful for controlling figure and plotting region

Argument	Meaning	Unit	Length
`fin`	Figure region	Inch	2
`pin`	Plotting region	Inch	2
`mai`	Plotting margins	Inch	4
`mar`	Plotting margins	Lines of text	4

see `?par` for more information

on. We can, however, explicitly instruct R not to reserve this space by using function `par`, which is intended to have side effects on the next plot on the current device. The `par`-settable arguments that are useful for controlling the physical size of the plot are listed in Table 3.1.

In Fig. 3.4, generated by

```
> oldpar = par(no.readonly = TRUE)
> layout(matrix(c(1, 2), 1, 2))
> plot(meuse, axes = TRUE, cex = 0.6)
> plot(meuse.sr, add = TRUE)
> title("Sample locations")
> par(mar = c(0, 0, 0, 0) + 0.1)
> plot(meuse, axes = FALSE, cex = 0.6)
```

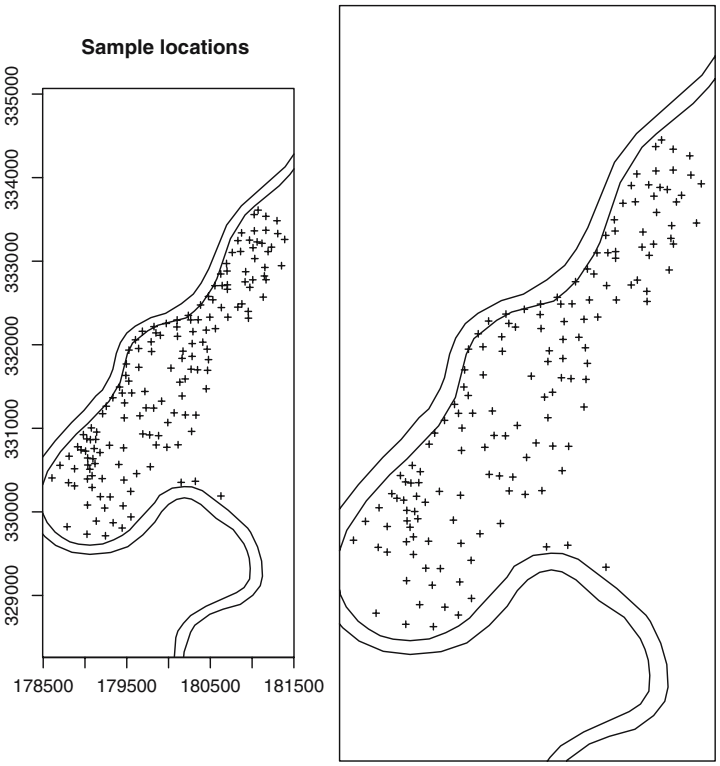

Fig. 3.4. Equal-area plots with (*left*) and without (*right*) the default space R reserves for axes and title(s)

```
> plot(meuse.sr, add = TRUE)
> box()
> par(oldpar)
```

the same data set is plotted twice within the same amount of space, at the left-hand side with R's default margins leaving space for axes, and on the right-hand side with maximised plotting space and no axes drawn.

Modifying the margins by setting `mar` in the `par` command, for example to `par(mar=c(3,3,2,1))` further optimises space usage when axes are drawn, leaving (little) space for a title. It should be noted that the margin sizes are absolute, expressed in units the height of a line of text, and so their effect on map scale decreases when the plotting region is enlarged.

The `plot` methods provided by package **sp** do not allow the printing of axis labels, such as 'Easting' and 'Northing', or 'x-coordinate' and 'y-coordinate'. The reason for this is technical, but mentioning axis names is usually obsolete once the graph is referred to as a map. The units of the coordinate reference system (such as metres) should be equal for both axes and do not need mentioning twice. Geographical coordinates are perhaps an exception, but this is made explicit by axis tic labels such as 52°N, or by adding a reference grid.

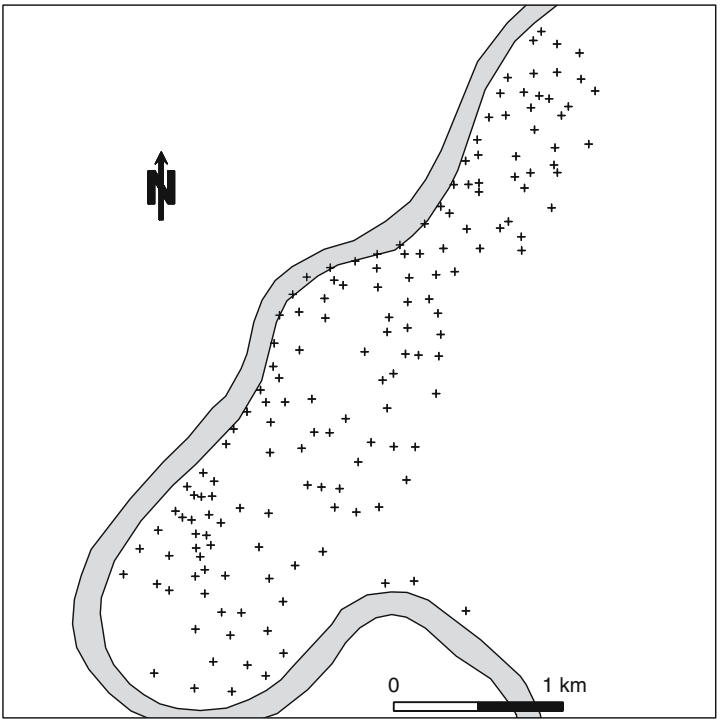

Fig. 3.5. Scale bar and north arrow as map elements

When we decide not to draw axes on a map, in addition to reference boundaries, we can provide the reader of a map with a guidance for distance and direction by plotting a scale bar and a north arrow, which can be placed interactively using `locator` followed by a few well-chosen clicks in the map (Fig. 3.5):

```
> plot(meuse)
> plot(meuse.sr, add = TRUE)
> plot(meuse)
> SpatialPolygonsRescale(layout.scale.bar(), offset = locator(1),
+   scale = 1000, fill = c("transparent", "black"), plot.grid = FALSE)
> text(locator(1), "0")
> text(locator(1), "1 km")
> SpatialPolygonsRescale(layout.north.arrow(), offset = locator(1),
+     scale = 400, plot.grid = FALSE)
```

When large numbers of maps for identical areas have to be produced with identical layout elements, it pays off to write a function that draws all layout elements. As an alternative, one may use conditioning plots; see the `spplot` method in Sect. 3.2.

3.1.3 Degrees in Axes Labels and Reference Grid

Unprojected data have coordinates in latitude and longitude degrees, with negative degrees referring to degrees west (of the prime meridian) and south (of the Equator). When unprojected spatial data are plotted using **sp** methods (`plot` or `spplot`), the axis label marks will give units in decimal degrees N/S/E/W, for example 50.5°N. An example is shown in Fig. 3.6.

When, for reference purposes, a grid needs to be added to a map, the function `gridlines` can be used to generate an object of class `SpatialLines`. By default it draws lines within the bounding box of the object at values where the default axes labels are drawn; other values can be specified. Grid lines may be latitude/longitude grids, and these are non-straight lines. This is accomplished by generating a grid for unprojected data, projecting it, and plotting it over the map shown. An example is given in Fig. 1.2. This is the code used to define and draw projected latitude/longitude grid lines and grid line labels for this figure, which uses the world map from package **maps**:

```
> library(maptools)
> library(maps)
> wrld <- map("world", interior = FALSE, xlim = c(-179,
+     179), ylim = c(-89, 89), plot = FALSE)
> wrld_p <- pruneMap(wrld, xlim = c(-179, 179))
> llCRS <- CRS("+proj=longlat +ellps=WGS84")
> wrld_sp <- map2SpatialLines(wrld_p, proj4string = llCRS)
> prj_new <- CRS("+proj=moll")
> library(rgdal)
> wrld_proj <- spTransform(wrld_sp, prj_new)
> wrld_grd <- gridlines(wrld_sp, easts = c(-179, seq(-150,
+     150, 50), 179.5), norths = seq(-75, 75, 15), ndiscr = 100)
> wrld_grd_proj <- spTransform(wrld_grd, prj_new)
> at_sp <- gridat(wrld_sp, easts = 0, norths = seq(-75,
+     75, 15), offset = 0.3)
```

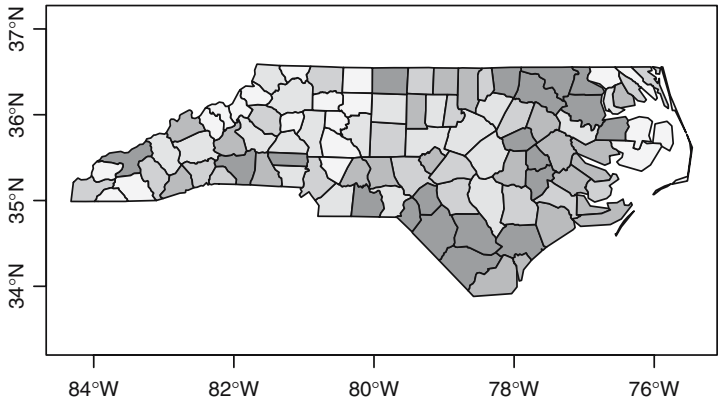

Fig. 3.6. Decimal degrees in axis labels: the North Carolina SIDS data

```
> at_proj <- spTransform(at_sp, prj_new)
> plot(wrld_proj, col = "grey60")
> plot(wrld_grd_proj, add = TRUE, lty = 3, col = "grey70")
> text(coordinates(at_proj),pos = at_proj$pos, offset = at_proj$offset,
+     labels = parse(text = as.character(at_proj$labels)),
+     cex = 0.6)
```

Here, function `gridat` returns an object to draw the labels for these 'gridded curves'.

3.1.4 Plot Size, Plotting Area, Map Scale, and Multiple Plots

R distinguishes between figure region, which is the size of the total figure including axes, title, etc., and plotting region, which is the area where the actual data are plotted. To control the total size of the figure, we can get and set the figure size in inches:

```
> par("pin")
> par(pin = c(4, 4))
```

If we want to enlarge the plotting window, we may have to close the current plotting device and re-open it specifying size, for example

```
> dev.off()
> X11(width = 10, height = 10)
```

on Unix machines; replace `X11` with `windows` on MS-Windows computers and with `quartz` on Mac OS X. When graphic output is written to files, we can use, for example

```
> postscript("file.ps", width = 10, height = 10)
```

The geographical (data) area that is shown on a plot is by default that of the data, extended with a 4% margin on each side. Because the plot size is fixed before plotting, only one of the axes will cover the entire plotting region, the other will be centred and have larger margins. We can control the data area plotted by passing `xlim` and `ylim` in a plot command, but by default they will still be extended with 4% on each side. To prevent this extension, we can set `par(xaxs="i")` and `par(yaxs="i")`. In the following example

```
> pin <- par("pin")
> dxy <- apply(bbox(meuse), 1, diff)
> ratio <- dxy[1]/dxy[2]
> par(pin = c(ratio * pin[2], pin[2]), xaxs = "i", yaxs = "i")
> plot(meuse, pch = 1)
> box()
```

we first set the aspect of the plotting region equal to that of the data points, and then we plot the points without allowing for the 4% extension of the range in all directions. The result (Fig. 3.7) is that in all four sides one plotting symbol is clipped by the plot border.

If we want to create more than one map in a single figure, as was done in Fig. 3.1, we can sub-divide the figure region into a number of sub-regions. We can split the figure into two rows and three columns either by

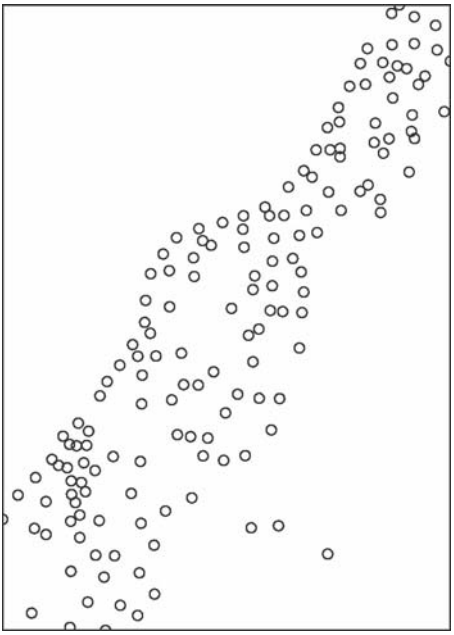

Fig. 3.7. Plotting region *exactly* equal to sample location ranges: border point symbols are clipped

```
> par(mfrow = c(2, 3))
```

or

```
> layout(matrix(1:6, 2, 3, byrow = TRUE))
```

Each time a plot command that would normally create a new plot is called (i.e. without `add = TRUE`), a plot is drawn in a new sub-area; this is done row-wise for this example, or column-wise when `byrow = FALSE`. Function `layout` also allows us to vary the height and width of the sub-areas.

Map scale is the ratio between the length of one unit on the map and one unit in the real world. It can only be controlled ahead of time when both the size of the plotting region, which is by default only a part of the figure size unless all margins are set to zero, and the plotting area are defined, or otherwise exactly known.

3.1.5 Plotting Attributes and Map Legends

Up to now we have only plotted the geometry or topology of the spatial objects. If in addition we want to show feature characteristics or *attributes* of the objects, we need to use type, size, or colour of the symbols, lines, or polygons. Grid cells are usually plotted as small adjacent squares, so their plotting is in some sense a special case of plotting polygons. Table 3.2 lists the

Table 3.2. Useful annotation arguments to be passed to `plot` or `image` methods

Class(es)	Argument	Meaning	Further help
SpatialLinesDataFrame	`col`	Colour	`?lines`
	`lwd`	Line width	`?lines`
	`lty`	Line type	`?lines`
SpatialPolygonsDataFrame	`border`	Border colour	`?polygon`
	`density`	Hashing density	`?polygon`
	`angle`	Hashing angle	`?polygon`
	`lty`	Line type	`?polygon`
	`pbg`	Hole colour	
SpatialPointsDataFrame	`pch`	Symbol	`?points`
	`col`	Colour	`?points`
	`bg`	Fill colour	`?points`
	`cex`	Symbol size	`?points`
SpatialPixelsDataFrame[a]	`zlim`	Attribute value limits	`?image.default`
and	`col`	Colours	`?image.default`
SpatialGridDataFrame	`breaks`	Break points	`?image.default`

[a]Use `image` to plot gridded data

graphic arguments that can be passed to the `plot` methods for the `Spatial` classes with attributes. When a specific colour, size, or symbol type refers to a specific numeric value or category label of an attribute, a map legend is needed to communicate this information. Example code for function `legend` is given below and shown in Fig. 3.8.

We provide `image` methods for objects of class `SpatialPixelsDataFrame` and `SpatialGridDataFrame`. As an example, we can plot interpolated (see Chap. 8) zinc concentration (`zinc.idw`) as a background image along with the data:

```
> grays = gray.colors(4, 0.55, 0.95)
> image(zn.idw, col = grays, breaks = log(c(100, 200, 400,
+     800, 1800)))
> plot(meuse.sr, add = TRUE)
> plot(meuse, pch = 1, cex = sqrt(meuse$zinc)/20, add = TRUE)
> legVals <- c(100, 200, 500, 1000, 2000)
> legend("left", legend = legVals, pch = 1, pt.cex = sqrt(legVals)/20,
+     bty = "n", title = "measured")
> legend("topleft", legend = c("100-200", "200-400", "400-800",
+     "800-1800"), fill = grays, bty = "n", title = "interpolated")
```

the result of which is shown in Fig. 3.8. This example shows how the `legend` command is used to place two legends, one for symbols and one for colours. In this example, rather light grey tones are used in order not to mask the black symbols drawn.

Fig. 3.8. Sample data points for zinc (ppm) plotted over an interpolated image, with symbol area proportional to measured concentration

3.2 Trellis/Lattice Plots with `spplot`

Apart from the traditional `plot` methods provided by package **sp**, a second method, called `spplot`, provides plotting of spatial data with attributes through the Trellis graphics system (Cleveland, 1993, 1994), which is for R provided (and extended) by package **lattice** (Sarkar, 2008). Trellis plots are a bit harder to deal with initially because plot annotation, the addition of information like `legend`, `lines`, `text`, etc., is handled differently and needs to be thought out first. The advantage they offer is that many maps can be composed into single (sets of) graphs, easily and efficiently.

3.2.1 A Straight Trellis Example

Consider the plotting of two interpolation scenarios for the zinc variable in the `meuse` data set, obtained on the direct scale and on the log scale. We can do

this either by the levelplot function from package **lattice**, or by using spplot, which is for grids a simple wrapper around levelplot:

```
> library(lattice)
> levelplot(z ~ x + y | name, spmap.to.lev(zn[c("direct",
+     "log")]), asp = "iso")
> spplot(zn[c("direct", "log")])
```

The results are shown in Fig. 3.9. Function levelplot needs a data.frame as second argument with the grid values for both maps in a single column (z) and a factor (name) to distinguish between them. Helper function spmap.to.lev converts the SpatialPixelsDataFrame object to this format by replicating the

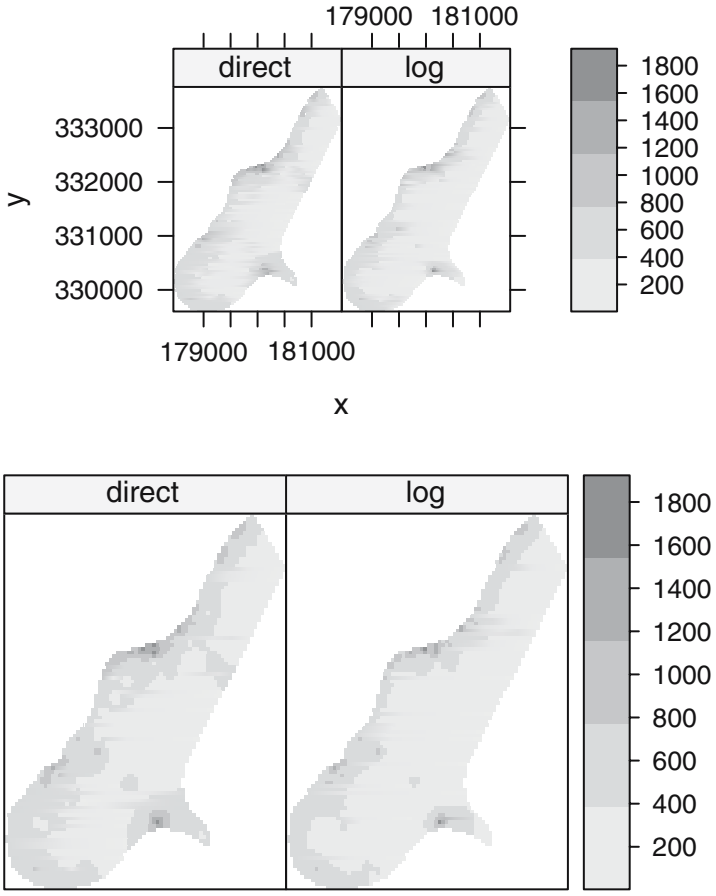

Fig. 3.9. Two interpolation scenarios for the meuse data set, plotted on the same total size. (*Top*) Example of levelplot, (*bottom*) example of the spplot wrapper, which turns off axes

coordinates, stacking the attribute variables, and adding a factor to distinguish the two maps. Function `spplot` plots each attribute passed in a single panel, which results in this case in two panels.

The `spplot` method does all this too, but hides many details. It provides a simple access to the functions provided by package **lattice** for plotting objects deriving from class `Spatial`, while retaining the flexibility offered by **lattice**. It also allows for adding geographic reference elements to maps.

Note that the plot shows four dimensions: the geographic space spanning x- and y-coordinates, the attribute values displayed in colour or grey tone, and the *panel* identifier, here the interpolation scenario but which may be used to denote, for example attribute variable or time.

3.2.2 Plotting Points, Lines, Polygons, and Grids

Function `spplot` plots spatial objects using colour (or grey tone) to denote attribute values. The first argument therefore has to be a spatial object with attributes.

Figure 3.10 shows a typical plot with four variables. If the goal is to compare the absolute levels in ppm across the four heavy metal variables, it makes

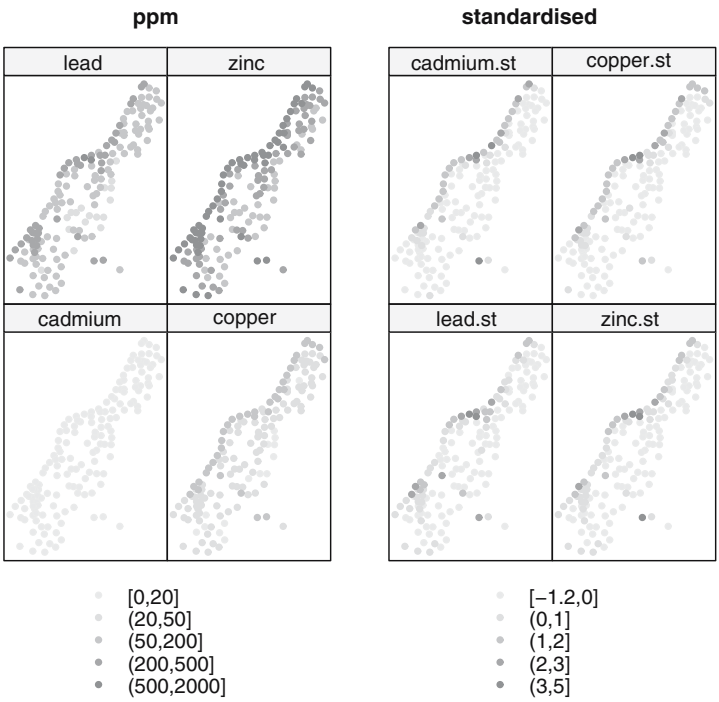

Fig. 3.10. Soil measurements for four heavy metals in the Meuse data set; (*left*) in ppm units, (*right*) each variable scaled to mean zero and unit standard variance

sense to plot them in a single figure with one legend. For such cases, the conditioning plots of `spplot` are ideal. Other cases in which multiple sub-maps are useful are, for example when different moments of time or different modelling scenarios are used to define the factor that splits the data over sub-plots (panels).

The first argument to `spplot` is a `Spatial*DataFrame` object with points, lines, polygons, or a grid. The second argument tells which attributes (column names or numbers) should be used; if omitted, all attributes are plotted. Further attributes control the plotting: colours, symbols, legend classes, size, axes, and geographical reference items to be added.

An example of a `SpatialLinesDataFrame` plot is shown in Fig. 3.11 (left). The R function `contourLines` is used to calculate the contourlines:

```
> library(maptools)
> data(meuse.grid)
> coordinates(meuse.grid) <- c("x", "y")
> meuse.grid <- as(meuse.grid, "SpatialPixelsDataFrame")
> im <- as.image.SpatialGridDataFrame(meuse.grid["dist"])
> cl <- ContourLines2SLDF(contourLines(im))
> spplot(cl)
```

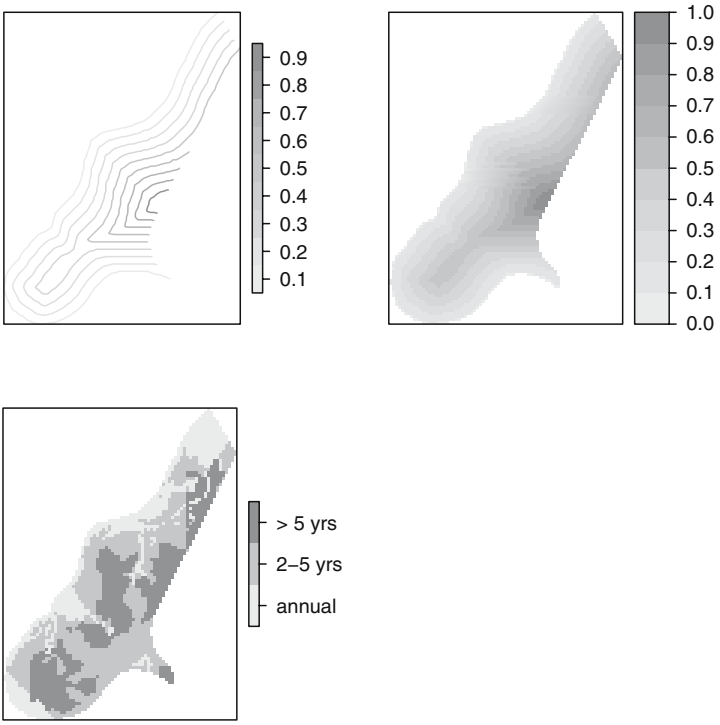

Fig. 3.11. (*Top left*) Contour lines for distance to river Meuse, levels represented by grey tones; (*top right*) grid plot of a numerical variable; (*bottom left*) plot of the factor variable flood frequency; note the different legend key

3.2.3 Adding Reference and Layout Elements to Plots

Method `spplot` takes a single argument, `sp.layout`, to annotate plots with lines, points, grids, polygons, text, or combinations of these. This argument contains either a single layout item or a list of layout items. A single layout item is a list object. Its first component is the name of the layout function to be called, followed by the object to be plotted and then optional arguments to adjust colour, symbol, size, etc. The layout functions provided are the following:

sp layout function	Object class	Useful arguments[a]
`sp.points`	`SpatialPoints`	`pch`, `cex`, `col`
`sp.polygons`	`SpatialPolygons`	`lty`, `lwd`, `col`
`sp.lines`	`SpatialLines`	`lty`, `lwd`, `col`
`sp.text`	text	(see `panel.text`)

[a]For help, see `?par`

An example of building an `sp.layout` structure is as follows:

```
> river <- list("sp.polygons", meuse.sr)
> north <- list("SpatialPolygonsRescale", layout.north.arrow(),
+     offset = c(178750, 332500), scale = 400)
> scale <- list("SpatialPolygonsRescale", layout.scale.bar(),
+     offset = c(180200, 329800), scale = 1000, fill = c("transparent",
+         "black"))
> txt1 <- list("sp.text", c(180200, 329950), "0")
> txt2 <- list("sp.text", c(181200, 329950), "1 km")
> pts <- list("sp.points", meuse, pch = 3, col = "black")
> meuse.layout <- list(river, north, scale, txt1, txt2,
+     pts)

> spplot(zn["log"], sp.layout = meuse.layout)
```

the result of which is shown in Fig. 3.12. Although the construction of this is more elaborate than annotating base plots, as was done for Fig. 3.5, this method seems better for the larger number of graphs as shown in Fig. 3.10.

A special layout element is `which` (integer), to control to which panel a layout item should be added. If `which` is present in the top-level list it applies to all layout items; in sub-lists with layout items it denotes the panel or set of panels in which the layout item should be drawn. Without `which`, layout items are drawn in each panel.

The order of items in the `sp.layout` argument matters; in principle objects are drawn in the order they appear. By default, when the object of `spplot` has points or lines, `sp.layout` items are drawn before the points to allow grids and polygons drawn as a background. For grids and polygons, `sp.layout` items are drawn afterwards (so the item will not be overdrawn by the grid and/or polygon). For grids, adding a list element `first = TRUE` ensures that the item is drawn *before* the grid is drawn (e.g. when filled polygons are

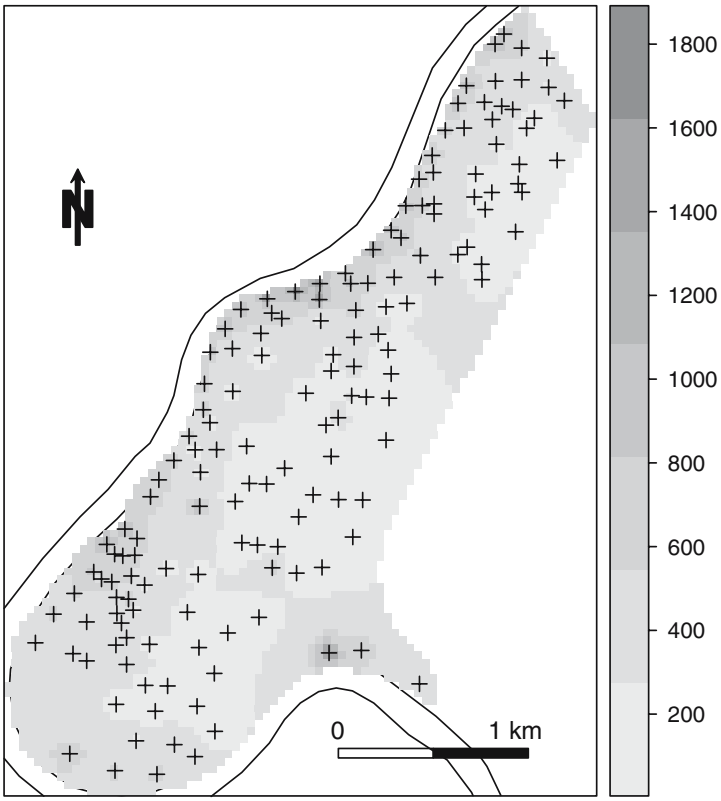

Fig. 3.12. Interpolated `spplot` image with layout elements

added). Transparency may help when combining layers; it is available for the PDF device and several other devices.

Function `sp.theme` returns a **lattice** theme that can be useful for plots made by `spplot`; use `trellis.par.set(sp.theme())` after a device is opened or changed to make this effective. Currently, this only sets the colours to `bpy.colors`.

3.2.4 Arranging Panel Layout

The default layout of `spplot` plots is computed by (i) the dimensions of the graphics device and the size of each panel and (ii) a row-wise ordering, starting top-left. The row-wise ordering can be started bottom-left if `as.table = FALSE` is passed to the `spplot` call. Note that `FALSE` is the default value for functions in **lattice**.

Besides `as.table`, panel layout can be modified with the `layout` and `skip` arguments. Argument `layout` is a numeric vector with the number of columns and the number of rows, for example `layout = c(3,4)` will result in three columns and four rows. Argument `skip` can be used to leave certain panels

blank, in plotting order: `layout = c(3,3), skip = c(F,T,T,F,F,T,F,F,F)` will plot six panels in a lower triangular 3×3 panel matrix. Figure 8.10 gives an example of this. More information about `layout`, `skip`, and `as.table` can be found in the help for **lattice** function `xyplot`.

3.3 Interacting with Plots

The interaction R allows with plots in the traditional and **lattice** plot systems is rather limited, compared with stand-alone software written for interacting with data, or GIS. The main functionality is centred around which information is present at the location where a mouse is clicked.

3.3.1 Interacting with Base Graphics

Base graphics has two functions to interact with interactive (i.e. screen) graphic devices:

`locator` returns the locations of points clicked, in coordinates of the x- and y-axis

`identify` plots and returns the labels (by default: row number) of the items nearest to the location clicked, within a specified maximum distance (0.25 inch in plot units, by default).

Both functions wait for user input; left mouse clicks are registered; a right mouse click ends the input. An example session for `identify` may look like this:

```
> plot(meuse)
> meuse.id <- identify(coordinates(meuse))
```

and the result may look like the left side of Fig. 3.13. An example digitise session, followed by selection and re-plotting of points within the area digitised may be as follows:

```
> plot(meuse)
> region <- locator(type = "o")
> n <- length(region$x)
> p <- Polygon(cbind(region$x, region$y)[c(1:n, 1), ],
+     hole = FALSE)
> ps <- Polygons(list(p), ID = "region")
> sps <- SpatialPolygons(list(ps))
> plot(meuse[!is.na(overlay(meuse, sps)), ], pch = 16,
+     cex = 0.5, add = TRUE)
```

with results in the right-hand side of Fig. 3.13. Note that we 'manually' close the polygon by adding the first point to the set of points digitised.

To identify particular polygons, we can use `locator` and overlay the points with the polygon layer shown in Fig. 3.6:

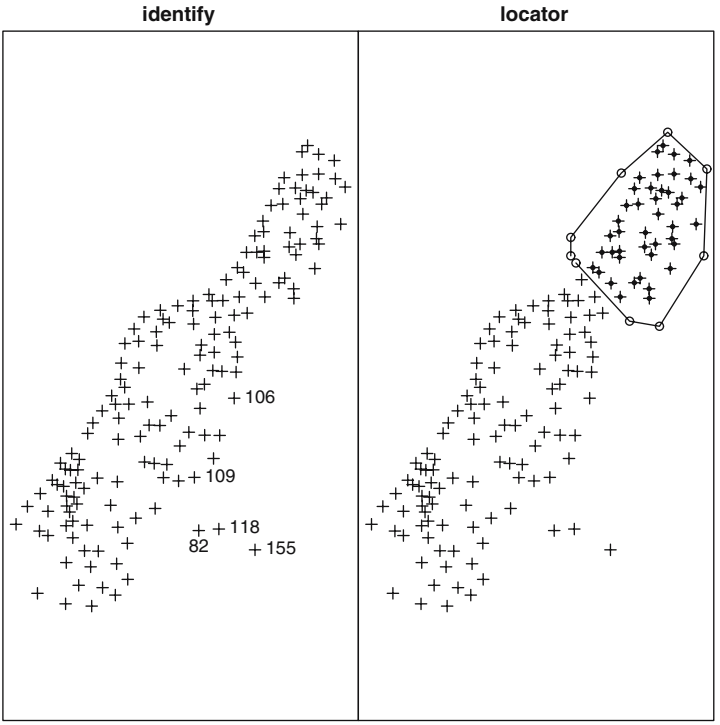

Fig. 3.13. Interaction with point plots. (*Left*) Individual identification of points; (*right*) digitising a region, highlighted points included in the region

```
> library(maptools)
> prj <- CRS("+proj=longlat +datum=NAD27")
> nc_shp <- system.file("shapes/sids.shp", package = "maptools")[1]
> nc <- readShapePoly(nc_shp, proj4string = prj)

> plot(nc)
> pt <- locator(type = "p")

> print(pt)

$x
[1] -78.69484

$y
[1] 35.8044

> overlay(nc, SpatialPoints(cbind(pt$x, pt$y), proj4string = prj))

    AREA PERIMETER CNTY_ CNTY_ID NAME  FIPS FIPSNO CRESS_ID BIR74
36 0.219      2.13  1938    1938 Wake 37183  37183       92 14484
   SID74 NWBIR74 BIR79 SID79 NWBIR79
36    16    4397 20857    31    6221
```

3.3.2 Interacting with spplot and Lattice Plots

In R, Trellis (**lattice**) plots have the same interaction functionality as base plots. However, the process is a bit more elaborate because multiple panels may be present. To select points with `spplot`, use

```
> ids <- spplot(meuse, "zinc", identify = TRUE)
```

This will show the points selected and return the selected points' row numbers.

In essence, and what the above function hides, we first select a panel, then identify within this panel, and finally unselect it, which is accomplished by the **lattice** functions

```
> library(lattice)
> trellis.focus("panel", column = 1, row = 1)
> ids <- panel.identify()
> trellis.unfocus()
```

Digitising can be done by the function `grid.locator` from package `grid`, which underlies the functionality in **lattice**. A single point is selected by

```
> library(grid)
> trellis.focus("panel", column = 1, row = 1)
> as.numeric(grid.locator())
> trellis.unfocus()
```

Package **sp** contains a simple function `spplot.locator` to return a digitised area, simulating the base plot `locator` behaviour. It returns a two-column matrix with spatial coordinates.

3.4 Colour Palettes and Class Intervals

3.4.1 Colour Palettes

R provides a number of colour palettes, and the functions providing them are self-descriptive: `rainbow`, `grey.colors`, `heat.colors`, `terrain.colors`, `topo.colors`, and `cm.colors` (`cm` for cyan-magenta) – `cm.colors` are the default palette in `spplot` and diverge from white. For quantitative data, shades in a single colour are usually preferred. These can be created by `colorRampPalette`, which creates a color interpolating function taking the required number of shades as argument, as in

```
> rw.colors <- colorRampPalette(c("red", "white"))
> image(meuse.grid["dist"], col = rw.colors(10))
```

Package **RColorBrewer** provides the palettes described (and printed) in Brewer et al. (2003) for continuous, diverging, and categorical variables. An interface for exploring how these palettes look on maps is found in the colorbrewer applet.[2]

[2] See `http://www.colorbrewer.org/`.

It also has information on suitability of each of the palettes for colour-blind people, black-and-white photo-copying, projecting by LCD projectors, use on LCD or CRT screens, and for colour printing. Another, non-interactive, overview is obtained by

```
> library(RColorBrewer)
> example(brewer.pal)
```

Package **sp** provides the ramp `bpy.colors` (blue-pink-yellow), which has the advantage that it has many colors and that it prints well both on color and black-and-white printers.

3.4.2 Class Intervals

Although we can mimic continuous variation by choosing many (e.g. 100 or more) colours, matching map colours to individual colours in the legend is approximate. If we want to communicate changes connected to certain fixed levels, for example levels related to regulation, or if we for other reasons want differentiable or identifiable class intervals, we should limit the number of classes to, for example six or less.

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, and natural breaks among others, or even simple fixed values. From there, the intervals can be used to generate colours from a colour palette as discussed earlier. 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.

We try just two styles, quantiles and Fisher-Jenks natural breaks for five classes (Slocum et al., 2005, pp. 85–86), among the many available – for further documentation see the help page of the `classIntervals` function. They yield quite different impressions, as we see:

```
> library(RColorBrewer)
> library(classInt)
> pal <- grey.colors(4, 0.95, 0.55, 2.2)
> q5 <- classIntervals(meuse$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

> diff(q5$brks)

[1]   73.8   59.6  193.2  297.6 1101.8

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

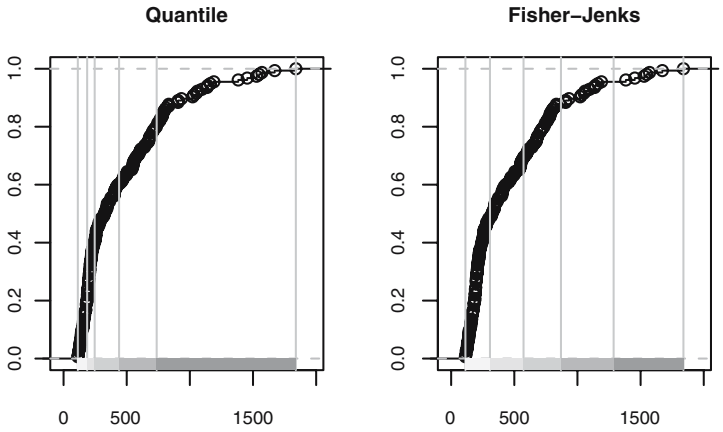

Fig. 3.14. Comparison of quantile and natural breaks methods for setting class intervals, Meuse bank zinc ppm

The empirical cumulative distribution function, used in the plot method for the `classIntervals` object returned, suggests that using quantiles is not necessarily a good idea. While of course the number of sites in each class is equal by definition, the observed values are far from uniformly distributed. Examining the widths of the classes using `diff` on the class breaks shows that many sites with moderate zinc values will be assigned to the darkest colour class. Figure 3.14 shows the plot of this class interval set compared with that for a five-class Fisher-Jenks classification. There are two implementations of this style, one named '`fisher`', the other '`jenks`'. This 'natural breaks' set of class intervals is based on minimising the within-class variance, like many of the other styles available.

```
> fj5 <- classIntervals(meuse$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

> diff(fj5$brks)

[1] 194.5 265.5 296.5 417.0 552.5

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

Once we are satisfied with the chosen class intervals and palette, we can go on to plot the data, using the `findColours` function to build a vector of colours and attributes, which can be used in constructing a legend:

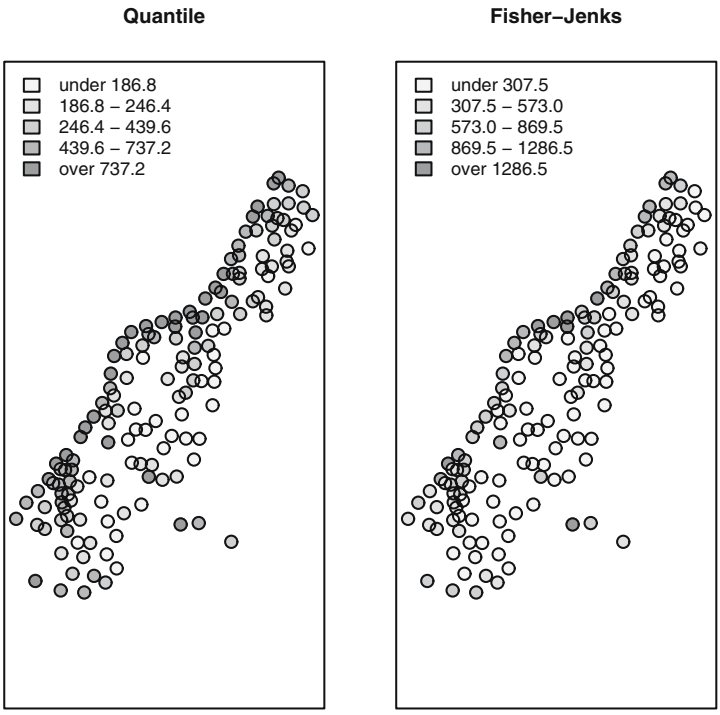

Fig. 3.15. Comparison of output maps made with quantile and natural breaks class intervals, Meuse bank zinc ppm

```
> q5Colours <- findColours(q5, pal)
> plot(meuse, col = q5Colours, pch = 19)
> legend("topleft", fill = attr(q5Colours, "palette"),
+     legend = names(attr(q5Colours, "table")), bty = "n")
```

The output for these two classifications is shown in Fig. 3.15, and does show that choice of representation matters. Using quantile-based class intervals, it appears that almost all the river bank sites are equally polluted, while the natural breaks intervals discriminate better.

For image, we can specify the `breaks` argument, as was done in Fig. 3.8. While the `classIntervals` function can be used with raster data, it may be prudent to search for class intervals using a sample of the input data, including the extremities to save time; this heuristic is used by many GIS. The default class interval style used by image is to divide the range into a number of classes of equal width (equivalent to the equal or pretty styles in `classIntervals`). With very skewed data, for example 2D density plots, this may give the impression of the data having disappeared, because almost all the cells will be in one extreme class, and only a few in other classes. Changing the class intervals will 'magically' reveal the data.

For the `spplot` methods for lines, polygons, and grids, we can pass the argument `pretty = TRUE`, which ensures that colour breaks coincide with legend values (see right-hand side of Fig. 3.11). To specify class intervals with `spplot`, for points data we can pass the `cuts` argument, and for lines, polygons, or grids we can pass the `at` argument. To also control the key tic marks and labels, we need to specify `colorkey` as well. For example, the middle plot of Fig. 3.11 was created by:

```
> cuts = (0:10)/10
> spplot(meuse.grid, "dist", colorkey = list(labels = list(at = cuts)),
+     at = cuts)
```

Having provided a framework for handling and visualising spatial data in R, we now move to demonstrate how user data may be imported into R, and the results of analysis exported.