Open Source Geospatial Software - an Introduction
Spatial Programming with R

V. Gómez-Rubio
Based on some course notes by Roger S. Bivand

Departamento de Matemáticas
Universidad de Castilla-La Mancha

17-18 November 2008
Department of Mathematics and Statistics
Lancaster University
Analysing Spatial Data in R: Why spatial data in R?
What is R?

- R is largely a ‘GNU S’, developed by some of the same people who developed S, plus a large group of public-spirited statisticians and programmers (many of whom had contributed libraries to S-PLUS).
What is R?

- R is largely a ‘GNU S’, developed by some of the same people who developed S, plus a large group of public-spirited statisticians and programmers (many of whom had contributed libraries to S-PLUS).
- ... the goal of the R project was (and remains) to take the S language to the masses, using many features of S as the foundation of an open-source and freely-available statistics environment.
What is R?

- R is largely a ‘GNU S’, developed by some of the same people who developed S, plus a large group of public-spirited statisticians and programmers (many of whom had contributed libraries to S-PLUS).
- ... the goal of the R project was (and remains) to take the S language to the masses, using many features of S as the foundation of an open-source and freely-available statistics environment.
- R: some object-orientated design features, a strong emphasis on graphics and visualizing data, and a steady flow of innovation (both computational and statistical) from the applied statistics community.
How does the community around R work?

- Jackman (The Political Methodologist, Spring 2003): “One of the great strengths of programs like S–PLUS and R: user-extensibility or ‘writing your own programs’.”
How does the community around R work?

- Jackman (The Political Methodologist, Spring 2003): “One of the great strengths of programs like S-PLUS and R: user-extensibility or ‘writing your own programs’.”

- “once methodological problems start being perceived or even defined in terms of what one’s favorite software does well, then the software has stopped being a tool, and has become a crutch, and at worse a shackle.”
How does the community around R work?

- Jackman (The Political Methodologist, Spring 2003): “One of the great strengths of programs like S-PLUS and R: user-extensibility or ‘writing your own programs’.”
- “once methodological problems start being perceived or even defined in terms of what one’s favorite software does well, then the software has stopped being a tool, and has become a crutch, and at worse a shackle.”
- R comes with a very wide range of functions for applied data analysis, and class definitions for objects like data frames, factors, etc.
How does the community around R work?

- Jackman (The Political Methodologist, Spring 2003): “One of the great strengths of programs like S–PLUS and R: user-extensibility or ‘writing your own programs’.”
- “once methodological problems start being perceived or even defined in terms of what one’s favorite software does well, then the software has stopped being a tool, and has become a crutch, and at worse a shackle.”
- R comes with a very wide range of functions for applied data analysis, and class definitions for objects like data frames, factors, etc.
- The community contributes further packages of documented code with examples — many available from CRAN.
Applied spatial data analysis with R

- Packages for importing commonly encountered spatial data formats
Applied spatial data analysis with R

- Packages for importing commonly encountered spatial data formats
- Range of contributed packages in spatial statistics and increasing awareness of the importance of spatial data analysis in the broader community
Applied spatial data analysis with R

- Packages for importing commonly encountered spatial data formats
- Range of contributed packages in spatial statistics and increasing awareness of the importance of spatial data analysis in the broader community
- Current contributed packages with spatial statistics applications (see R spatial projects):
Applied spatial data analysis with R

- Packages for importing commonly encountered spatial data formats
- Range of contributed packages in spatial statistics and increasing awareness of the importance of spatial data analysis in the broader community
- Current contributed packages with spatial statistics applications (see R spatial projects):

  **point patterns**: spatstat, VR:spatial, splancs;
Applied spatial data analysis with R

- Packages for importing commonly encountered spatial data formats
- Range of contributed packages in spatial statistics and increasing awareness of the importance of spatial data analysis in the broader community
- Current contributed packages with spatial statistics applications (see R spatial projects):
  
  **point patterns:** spatstat, VR:spatial, splancs;
  
  **geostatistics:** gstat, geoR, geoRglm, fields, spBayes, RandomFields, VR:spatial, sgeostat, vardiag;
Applied spatial data analysis with R

- Packages for importing commonly encountered spatial data formats
- Range of contributed packages in spatial statistics and increasing awareness of the importance of spatial data analysis in the broader community
- Current contributed packages with spatial statistics applications (see R spatial projects):
  - **point patterns**: spatstat, VR:spatial, splancs;
  - **geostatistics**: gstat, geoR, geoRglm, fields, spBayes, RandomFields, VR:spatial, sgeostat, vardiag;
  - **lattice/area data**: spdep, DCluster, spgwr, ade4;
Packages for importing commonly encountered spatial data formats

Range of contributed packages in spatial statistics and increasing awareness of the importance of spatial data analysis in the broader community

Current contributed packages with spatial statistics applications (see R spatial projects):

**Point patterns:** spatstat, VR:spatial, splancs;

**Geostatistics:** gstat, geoR, geoRglm, fields, spBayes, RandomFields, VR:spatial, sgeostat, vardiag;

**Lattice/area data:** spdep, DCluster, spgwr, ade4;

**GIS:** spgrass6, RSAGA
Non-standard research questions

- Because S (and its implementation R) is a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities, existing functions can be modified.
Non-standard research questions

- Because S (and its implementation R) is a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities, existing functions can be modified.

- Jackman: “if your notion of data analysis runs to more than estimating coefficients and t-statistics . . . then from time-to-time you’ll find yourself programming, if only a little . . . easy programming and flexibility is key for a serious statistical computing environment.”
Non-standard research questions

- Because S (and its implementation R) is a well-developed, simple and effective programming language which includes conditionals, loops, user-defined recursive functions and input and output facilities, existing functions can be modified.
- Jackman: “if your notion of data analysis runs to more than estimating coefficients and t-statistics . . . then from time-to-time you’ll find yourself programming, if only a little . . . easy programming and flexibility is key for a serious statistical computing environment.”
- In any case, documenting the analysis process is a “good thing”, so programming scripts are not just a burden, certainly for users doing original research and repetitive work, arguably for student classes too.
Getting up to speed in R

- R is a programming language, so using it can build on earlier experience with programming (accepting that languages do vary).
Getting up to speed in R

- R is a programming language, so using it can build on earlier experience with programming (accepting that languages do vary).
- The “Introduction to R” shipped with R is up to date, and covers many questions quite well.
Getting up to speed in R

- R is a programming language, so using it can build on earlier experience with programming (accepting that languages do vary).
- The “Introduction to R ” shipped with R is up to date, and covers many questions quite well.
- There are a number of online resources too, both linked from CRAN, and others; Gilberto câmara has assembled a systematic introduction bearing on spatial data.
Getting up to speed in R

- R is a programming language, so using it can build on earlier experience with programming (accepting that languages do vary).
- The “Introduction to R” shipped with R is up to date, and covers many questions quite well.
- There are a number of online resources too, both linked from CRAN, and others; Gilberto Câmara has assembled a systematic introduction bearing on spatial data.
- Robert Gentleman’s introduction to classes and methods in R is still one of the clearest.
Analysing Spatial Data in R: Representing Spatial Data
Object framework

To begin with, all contributed packages for handling spatial data in R had different representations of the data.
Object framework

- To begin with, all contributed packages for handling spatial data in R had different representations of the data.
- This made it difficult to exchange data both within R between packages, and between R and external file formats and applications.
Object framework

- To begin with, all contributed packages for handling spatial data in R had different representations of the data.
- This made it difficult to exchange data both within R between packages, and between R and external file formats and applications.
- The result has been an attempt to develop shared classes to represent spatial data in R, allowing some shared methods and many-to-one, one-to-many conversions.
Object framework

- To begin with, all contributed packages for handling spatial data in R had different representations of the data.
- This made it difficult to exchange data both within R between packages, and between R and external file formats and applications.
- The result has been an attempt to develop shared classes to represent spatial data in R, allowing some shared methods and many-to-one, one-to-many conversions.
- We chose to use new-style classes to represent spatial data, and are confident that this choice was justified.
Spatial objects

- The foundation object is the Spatial class, with just two slots (new-style class objects have pre-defined components called slots)
The foundation object is the \texttt{Spatial} class, with just two slots
(new-style class objects have pre-defined components called slots)
The first is a bounding box, and is mostly used for setting up plots
The foundation object is the Spatial class, with just two slots (new-style class objects have pre-defined components called slots).

The first is a bounding box, and is mostly used for setting up plots.

The second is a CRS class object defining the coordinate reference system, and may be set to CRS(as.character(NA)), its default value.
Spatial objects

- The foundation object is the Spatial class, with just two slots (new-style class objects have pre-defined components called slots)
- The first is a bounding box, and is mostly used for setting up plots
- The second is a CRS class object defining the coordinate reference system, and may be set to CRS(as.character(NA)), its default value.
- Operations on Spatial* objects should update or copy these values to the new Spatial* objects being created
The most basic spatial data object is a point, which may have 2 or 3 dimensions.
The most basic spatial data object is a point, which may have 2 or 3 dimensions.

A single coordinate, or a set of such coordinates, may be used to define a `SpatialPoints` object; coordinates should be of mode `double` and will be promoted if not.
The most basic spatial data object is a point, which may have 2 or 3 dimensions.

A single coordinate, or a set of such coordinates, may be used to define a `SpatialPoints` object; coordinates should be of mode `double` and will be promoted if not.

The points in a `SpatialPoints` object may be associated with a row of attributes to create a `SpatialPointsDataFrame` object.
The most basic spatial data object is a point, which may have 2 or 3 dimensions.

A single coordinate, or a set of such coordinates, may be used to define a `SpatialPoints` object; coordinates should be of mode `double` and will be promoted if not.

The points in a `SpatialPoints` object may be associated with a row of attributes to create a `SpatialPointsDataFrame` object.

The coordinates and attributes may, but do not have to be keyed to each other using ID values.
Spatial points classes and their slots

- SpatialPointsDataFrame
  - SpatialPoints
    - coords.nrs
    - data
  - data.frame
- Spatial
  - bbox
  - proj4string
- SpatialPoints
  - coords
  - Spatial
Spatial points

Using the Meuse bank data set of soil samples and measurements of heavy metal pollution provided with **sp**, we’ll make a `SpatialPoints` object.

```r
> library(sp)
> data(meuse)
> coords <- SpatialPoints(meuse[, c("x", "y")])
> summary(coords)
```

Object of class `SpatialPoints`
Coordinates:

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>178605</td>
<td>181390</td>
</tr>
<tr>
<td>y</td>
<td>329714</td>
<td>333611</td>
</tr>
</tbody>
</table>

Is projected: NA
proj4string : [NA]
Number of points: 155
Now we’ll add the original data frame to make a SpatialPointsDataFrame object. Many methods for standard data frames just work with SpatialPointsDataFrame objects.

```r
> meuse1 <- SpatialPointsDataFrame(coords, meuse)
> names(meuse1)
[1] "x"  "y"  "cadmium" "copper" "lead"  "zinc"
[7] "elev" "dist" "om"  "ffreq" "soil"  "lime"
[13] "landuse" "dist.m"

> summary(meuse1$zinc)

    Min. 1st Qu.  Median    Mean 3rd Qu. Max. 
   113.0   198.0   326.0   469.7  674.5  1839.0

> stem(meuse1$zinc, scale = 1/2)

The decimal point is 2 digit(s) to the right of the |

   0 | 12223333344444455566667778888899999999
  2 | 0000000111111122222334445556666788880022334455788
  4 | 00012235677001455556789
  6 | 01144678890012455678889
  8 | 0133113
 10 | 235604469
 12 | 8
 14 | 5357
 16 | 7
 18 | 4
```
Spatial lines and polygons

- A Line object is just a spaghetti collection of 2D coordinates; a Polygon object is a Line object with equal first and last coordinates.
Spatial lines and polygons

- A Line object is just a spaghetti collection of 2D coordinates; a Polygon object is a Line object with equal first and last coordinates.
- A Lines object is a list of Line objects, such as all the contours at a single elevation; the same relationship holds between a Polygons object and a list of Polygon objects, such as islands belonging to the same county.
Spatial lines and polygons

- A Line object is just a spaghetti collection of 2D coordinates; a Polygon object is a Line object with equal first and last coordinates.
- A Lines object is a list of Line objects, such as all the contours at a single elevation; the same relationship holds between a Polygons object and a list of Polygon objects, such as islands belonging to the same county.
- SpatialLines and SpatialPolygons objects are made using lists of Lines or Polygons objects respectively.
Spatial lines and polygons

- A Line object is just a spaghetti collection of 2D coordinates; a Polygon object is a Line object with equal first and last coordinates.
- A Lines object is a list of Line objects, such as all the contours at a single elevation; the same relationship holds between a Polygons object and a list of Polygon objects, such as islands belonging to the same county.
- SpatialLines and SpatialPolygons objects are made using lists of Lines or Polygons objects respectively.
- SpatialLinesDataFrame and SpatialPolygonsDataFrame objects are defined using SpatialLines and SpatialPolygons objects and standard data frames, and the ID fields are here required to match the data frame row names.
Spatial Polygons classes and slots
The Meuse bank data set also includes the coordinates of the edge of the river, linked together at the edge of the study area to form a polygon. We can make these coordinates into a `SpatialPolygons` object:

```r
> data(meuse.riv)
> str(meuse.riv)

  num [1:176, 1:2] 182004 182137 182252 182314 182332 ... 

> river_polygon <- Polygons(list(Polygon(meuse.riv)), ID = "meuse")
> rivers <- SpatialPolygons(list(river_polygon))
> summary(rivers)

Object of class SpatialPolygons
Coordinates:
    min    max
r1 178304.0 182331.5
r2 325698.5 337684.8
Is projected: NA
proj4string : [NA]
```
Spatial lines

There is a helper function `contourLines2SLDF` to convert the list of contours returned by `contourLines` into a `SpatialLinesDataFrame` object. This example shows how the data slot row names match the ID slot values of the set of `Lines` objects making up the `SpatialLinesDataFrame`, note that some `Lines` objects include multiple `Line` objects:

```r
> library(maptools)

> volcano_sl <- ContourLines2SLDF(contourLines(volcano))
> row.names(slot(volcano_sl, "data"))

[1] "C_1" "C_2" "C_3" "C_4" "C_5" "C_6" "C_7" "C_8" "C_9"
[10] "C_10"

> sapply(slot(volcano_sl, "lines"), function(x) slot(x, + "ID"))

[1] "C_1" "C_2" "C_3" "C_4" "C_5" "C_6" "C_7" "C_8" "C_9"
[10] "C_10"

> sapply(slot(volcano_sl, "lines"), function(x) length(slot(x, + "Lines")))

[1] 3 4 1 1 1 2 2 3 2 1

> volcano_sl$level

[1] 100 110 120 130 140 150 160 170 180 190
Levels: 100 110 120 130 140 150 160 170 180 190
Spatial grids and pixels

- There are two representations for data on regular rectangular grids (oriented N-S, E-W): `SpatialPixels` and `SpatialGrid`
There are two representations for data on regular rectangular grids (oriented N-S, E-W): SpatialPixels and SpatialGrid

- SpatialPixels are like SpatialPoints objects, but the coordinates have to be regularly spaced; the coordinates are stored, as are grid indices.
Spatial grids and pixels

- There are two representations for data on regular rectangular grids (oriented N-S, E-W): SpatialPixels and SpatialGrid
- SpatialPixels are like SpatialPoints objects, but the coordinates have to be regularly spaced; the coordinates are stored, as are grid indices
- SpatialPixelsDataFrame objects only store attribute data where it is present, but need to store the coordinates and grid indices of those grid cells
Spatial grids and pixels

- There are two representations for data on regular rectangular grids (oriented N-S, E-W): SpatialPixels and SpatialGrid

- SpatialPixels are like SpatialPoints objects, but the coordinates have to be regularly spaced; the coordinates are stored, as are grid indices

- SpatialPixelsDataFrame objects only store attribute data where it is present, but need to store the coordinates and grid indices of those grid cells

- SpatialGridDataFrame objects do not need to store coordinates, because they fill the entire defined grid, but they need to store NA values where attribute values are missing
Spatial grid and pixels classes and their slots
Let’s make a `SpatialPixelsDataFrame` object for the Meuse bank grid data provided, with regular points at a 40m spacing. The data include soil types, flood frequency classes and distance from the river bank:

```r
> data(meuse.grid)
> coords <- SpatialPixels(SpatialPoints(meuse.grid[, c("x",
   + "y")]))
> meuseg1 <- SpatialPixelsDataFrame(coords, meuse.grid)
> names(meuseg1)
[1] "x" "y" "part.a" "part.b" "dist" "soil" "ffreq"
> slot(meuseg1, "grid")

       x      y
cellcentre.offset 178460 329620
cells.size       40     40
cells.dim        78    104

> object.size(meuseg1)
[1] 339036
> dim(slot(meuseg1, "data"))
[1] 3103    7
```
In this case we convert the `SpatialPixelsDataFrame` object to a `SpatialGridDataFrame` by making a change in-place. In other contexts, it is much more usual to create the `GridTopology` object in the `grid` slot directly, and populate the grid from there, as we’ll see later:

```r
> meuseg2 <- meuseg1
> fullgrid(meuseg2) <- TRUE
> slot(meuseg2, "grid")

x  y

<table>
<thead>
<tr>
<th>cellcentre.offset</th>
<th>178460</th>
<th>329620</th>
</tr>
</thead>
<tbody>
<tr>
<td>cellsize</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>cells.dim</td>
<td>78</td>
<td>104</td>
</tr>
</tbody>
</table>

> class(slot(meuseg2, "grid"))

[1] "GridTopology"
attr("package")
[1] "sp"

> object.size(meuseg2)

[1] 425684

> dim(slot(meuseg2, "data"))

[1] 8112    7
Spatial classes provided by **sp**

This table summarises the classes provided by **sp**, and shows how they build up to the objects of most practical use, the Spatial*DataFrame family objects:

<table>
<thead>
<tr>
<th>data type</th>
<th>class</th>
<th>attributes</th>
<th>extends</th>
</tr>
</thead>
<tbody>
<tr>
<td>points</td>
<td>SpatialPoints</td>
<td>none</td>
<td>Spatial</td>
</tr>
<tr>
<td>points</td>
<td>SpatialPointsDataFrame</td>
<td>data.frame</td>
<td>SpatialPoints</td>
</tr>
<tr>
<td>pixels</td>
<td>SpatialPixels</td>
<td>none</td>
<td>SpatialPoints</td>
</tr>
<tr>
<td>pixels</td>
<td>SpatialPixelsDataFrame</td>
<td>data.frame</td>
<td>SpatialPixels</td>
</tr>
<tr>
<td>full grid</td>
<td>SpatialGrid</td>
<td>none</td>
<td>SpatialPixels</td>
</tr>
<tr>
<td>full grid</td>
<td>SpatialGridDataFrame</td>
<td>data.frame</td>
<td>SpatialGrid</td>
</tr>
<tr>
<td>line</td>
<td>Line</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>lines</td>
<td>Lines</td>
<td>none</td>
<td>Line list</td>
</tr>
<tr>
<td>lines</td>
<td>SpatialLines</td>
<td>none</td>
<td>Spatial, Lines list</td>
</tr>
<tr>
<td>lines</td>
<td>SpatialLinesDataFrame</td>
<td>data.frame</td>
<td>SpatialLines</td>
</tr>
<tr>
<td>polygon</td>
<td>Polygon</td>
<td>none</td>
<td>Line</td>
</tr>
<tr>
<td>polygons</td>
<td>Polygons</td>
<td>none</td>
<td>Polygon list</td>
</tr>
<tr>
<td>polygons</td>
<td>SpatialPolygons</td>
<td>none</td>
<td>Spatial, Polygons list</td>
</tr>
<tr>
<td>polygons</td>
<td>SpatialPolygonsDataFrame</td>
<td>data.frame</td>
<td>SpatialPolygons</td>
</tr>
</tbody>
</table>
This table summarises the methods provided by `sp`:

<table>
<thead>
<tr>
<th>method</th>
<th>what it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[</code></td>
<td>select spatial items (points, lines, polygons, or rows/cols from a grid) and/or attributes variables</td>
</tr>
<tr>
<td><code>$</code>, <code>$&lt;-</code>, <code>[[</code>, <code>[[&lt;-</code></td>
<td>retrieve, set or add attribute table columns</td>
</tr>
<tr>
<td><code>spsample</code></td>
<td>sample points from a set of polygons, on a set of lines or from a gridded area</td>
</tr>
<tr>
<td><code>bbox</code></td>
<td>get the bounding box</td>
</tr>
<tr>
<td><code>proj4string</code></td>
<td>get or set the projection (coordinate reference system)</td>
</tr>
<tr>
<td><code>coordinates</code></td>
<td>set or retrieve coordinates</td>
</tr>
<tr>
<td><code>coerce</code></td>
<td>convert from one class to another</td>
</tr>
<tr>
<td><code>overlay</code></td>
<td>combine two different spatial objects</td>
</tr>
<tr>
<td><code>point.in.polygon</code></td>
<td>do point(s) fall in a given polygon?</td>
</tr>
</tbody>
</table>
Using Spatial family objects

Very often, the user never has to manipulate Spatial family objects directly, as we have been doing here, because methods to create them from external data are also provided.
Using **Spatial family objects**

- Very often, the user never has to manipulate **Spatial** family objects directly, as we have been doing here, because methods to create them from external data are also provided.

- Because the **Spatial*DataFrame** family objects behave in most cases like data frames, most of what we are used to doing with standard data frames just works — like "[]" or $ (but no **merge**, etc., yet)
Using **Spatial family objects**

- Very often, the user never has to manipulate **Spatial** family objects directly, as we have been doing here, because methods to create them from external data are also provided.

- Because the **Spatial*DataFrame** family objects behave in most cases like data frames, most of what we are used to doing with standard data frames just works — like "[]" or $ (but no merge, etc., yet).

- These objects are very similar to typical representations of the same kinds of objects in geographical information systems, so they do not suit spatial data that is not geographical (like medical imaging) as such.
Using **Spatial family objects**

- Very often, the user never has to manipulate **Spatial** family objects directly, as we have been doing here, because methods to create them from external data are also provided.

- Because the **Spatial*DataFrame** family objects behave in most cases like data frames, most of what we are used to doing with standard data frames just works — like "[" or $ (but no **merge**, etc., yet).

- These objects are very similar to typical representations of the same kinds of objects in geographical information systems, so they do not suit spatial data that is not geographical (like medical imaging) as such.

- They provide a standard base for analysis packages on the one hand, and import and export of data on the other, as well as shared methods, like those for visualisation we turn to now.
Analysing Spatial Data in R: Vizualising Spatial Data
Displaying spatial data is one of the chief reasons for providing ways of handling it in a statistical environment.
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.
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).
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.
There are base graphics plot methods for the key `Spatial*` classes, including the `Spatial` class, which just sets up the axes.
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.
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).
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.
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:

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

```r
> 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.
Both SpatialPixels and SpatialGrid objects are plotted like SpatialPoints objects, with plotting characters

```r
> 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.
To include attribute values means making choices about how to represent their values graphically, known in some GIS as symbology.
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.
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.
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.
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”

```r
> 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.

```r
> 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.
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

```r
> 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
Lattice graphics will only come into their own later on, when we want to plot several variables with the same scale together for comparison.
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.
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.
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`
Bubble plots are a convenient way of representing the attribute values by the size of a symbol.

```r
> 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.
The use of lattice plotting methods yields easy legend generation, which is another attraction

```r
> 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
More realism

- So far we have just used canned data and spatial objects rather than anything richer
- The visualization methods are also quite flexible — both the base graphics and lattice graphics methods can be extensively customised
More realism

- So far we have just used canned data and spatial objects rather than anything richer
- The visualisation 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 \texttt{sp} objects, in particular the \texttt{overlay} and \texttt{spsample} methods with a range of argument signatures
More realism

- So far we have just used canned data and spatial objects rather than anything richer
- The visualisation 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
Analysing Spatial Data in R: Accessing spatial data
Having described how spatial data may be represented in R, and how to visualise these objects, we need to move on to accessing user data.
Having described how spatial data may be represented in R, and how to visualise these objects, we need to move on to accessing user data. There are quite a number of packages handling and analysing spatial data on CRAN, and others off-CRAN, and their data objects can be converted to or from `sp` object form.
Having described how spatial data may be represented in R, and how to visualise these objects, we need to move on to accessing user data. There are quite a number of packages handling and analysing spatial data on CRAN, and others off-CRAN, and their data objects can be converted to or from `sp` object form. We need to cover how coordinate reference systems are handled, because they are the foundation for spatial data integration.
Having described how spatial data may be represented in R, and how to visualise these objects, we need to move on to accessing user data. There are quite a number of packages handling and analysing spatial data on CRAN, and others off-CRAN, and their data objects can be converted to or from sp object form. We need to cover how coordinate reference systems are handled, because they are the foundation for spatial data integration. Both here, and in relation to reading and writing various file formats, things have advanced a good deal since the R News note.
Creating objects within R

As mentioned in unit 1, maptools includes ContourLines2SLDF() to convert contour lines to SpatialLinesDataFrame objects.
Creating objects within R

- As mentioned in unit 1, `maptools` includes `ContourLines2SLDF()` to convert contour lines to `SpatialLinesDataFrame` objects.
- `maptools` also allows lines or polygons from `maps` to be used as `sp` objects.
Creating objects within R

- As mentioned in unit 1, `maptools` includes `ContourLines2SLDF()` to convert contour lines to `SpatialLinesDataFrame` objects.
- `maptools` also allows lines or polygons from `maps` to be used as `sp` objects.
- `maptools` can export `sp` objects to `PBSmapping`.
Creating objects within R

- As mentioned in unit 1, `maptools` includes `ContourLines2SLDF()` to convert contour lines to `SpatialLinesDataFrame` objects.
- `maptools` also allows lines or polygons from `maps` to be used as `sp` objects.
- `maptools` can export `sp` objects to `PBSmapping`.
- `maptools` uses `gpclib` to check polygon topology and to dissolve polygons.
Creating objects within R

- As mentioned in unit 1, `maptools` includes `ContourLines2SLDF()` to convert contour lines to `SpatialLinesDataFrame` objects.
- `maptools` also allows lines or polygons from `maps` to be used as `sp` objects.
- `maptools` can export `sp` objects to `PBSmapping`.
- `maptools` uses `gpclib` to check polygon topology and to dissolve polygons.
- `maptools` converts some `sp` objects for use in `spatstat`.
Creating objects within R

- As mentioned in unit 1, **maptools** includes ContourLines2SLDF() to convert contour lines to SpatialLinesDataFrame objects.
- **maptools** also allows lines or polygons from **maps** to be used as **sp** objects.
- **maptools** can export **sp** objects to **PBSmapping**.
- **maptools** uses **gpclib** to check polygon topology and to dissolve polygons.
- **maptools** converts some **sp** objects for use in **spatstat**.
- **maptools** can read GSHHS high-resolution shoreline data into SpatialPolygon objects.
There are number of valuable geographical databases in map format that can be accessed directly — beware of IDs!

```r
> library(maptools)
> library(maps)
> ill <- map("county", regions = "illinois",
+     plot = FALSE, fill = TRUE)
> IDs <- sub("^illinois,"", 
+     ill$names)
> ill_sp <- map2SpatialPolygons(ill,
+     IDs, CRS("+proj=longlat"))
> plot(ill_sp, axes = TRUE)
```
Coordinate reference systems (CRS) are at the heart of geodetics and cartography: how to represent a bumpy ellipsoid on the plane.
Coordinate reference systems

- Coordinate reference systems (CRS) are at the heart of geodetics and cartography: how to represent a bumpy ellipsoid on the plane.
- We can speak of geographical CRS expressed in degrees and associated with an ellipse, a prime meridian and a datum, and projected CRS expressed in a measure of length, and a chosen position on the earth, as well as the underlying ellipse, prime meridian and datum.
Coordinate reference systems

- Coordinate reference systems (CRS) are at the heart of geodetics and cartography: how to represent a bumpy ellipsoid on the plane.

- We can speak of geographical CRS expressed in degrees and associated with an ellipse, a prime meridian and a datum, and projected CRS expressed in a measure of length, and a chosen position on the earth, as well as the underlying ellipse, prime meridian and datum.

- Most countries have multiple CRS, and where they meet there is usually a big mess — this led to the collection by the European Petroleum Survey Group (EPSG, now Oil & Gas Producers (OGP) Surveying & Positioning Committee) of a geodetic parameter dataset.
Coordinate reference systems

- The EPSG list among other sources is used in the workhorse PROJ.4 library, which as implemented by Frank Warmerdam handles transformation of spatial positions between different CRS.
Coordinate reference systems

- The EPSG list among other sources is used in the workhorse PROJ.4 library, which as implemented by Frank Warmerdam handles transformation of spatial positions between different CRS.
- This library is interfaced with R in the `rgdal` package, and the `CRS` class is defined partly in `sp`, partly in `rgdal`.
Coordinate reference systems

- The EPSG list among other sources is used in the workhorse PROJ.4 library, which as implemented by Frank Warmerdam handles transformation of spatial positions between different CRS.
- This library is interfaced with R in the `rgdal` package, and the `CRS` class is defined partly in `sp`, partly in `rgdal`.
- A `CRS` object is defined as a character NA string or a valid PROJ.4 CRS definition.
Coordinate reference systems

- The EPSG list among other sources is used in the workhorse PROJ.4 library, which as implemented by Frank Warmerdam handles transformation of spatial positions between different CRS.
- This library is interfaced with R in the `rgdal` package, and the `CRS` class is defined partly in `sp`, partly in `rgdal`.
- A `CRS` object is defined as a character NA string or a valid PROJ.4 CRS definition.
- The validity of the definition can only be checked if `rgdal` is loaded.
In a Dutch navigation example, a chart position in the ED50 datum has to be compared with a GPS measurement in WGS84 datum right in front of the jetties of IJmuiden, both in geographical CRS. Using the `spTransform` method makes the conversion, using EPSG and external information to set up the ED50 CRS. The difference is about 124m; lots of details about CRS in general can be found in Grids & Datums.
Let's have a look at the Meuse bank CRS — Grids & Datums gives some hints in February 2003 to search for Amersfoort in EPSG:

```r
> EPSG <- make_EPSG()
> EPSG[grep("Amersfoort", EPSG$note), 1:2]

<table>
<thead>
<tr>
<th>code</th>
<th>note</th>
</tr>
</thead>
<tbody>
<tr>
<td>209</td>
<td>4289 # Amersfoort</td>
</tr>
<tr>
<td>2985</td>
<td>28991 # Amersfoort / RD Old</td>
</tr>
<tr>
<td>2986</td>
<td>28992 # Amersfoort / RD New</td>
</tr>
</tbody>
</table>

> RD_New <- CRS("+init=epsg:28992")
> res <- CRSargs(RD_New)
> cat(strwrap(res), sep = "\n")

+init=epsg:28992 +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889 +k=0.9999079 +x_0=155000 +y_0=463000 +ellps=bessel +units=m +no_defs

> res <- showWKT(CRSargs(RD_New), morphToESRI = TRUE)
> cat(strwrap(gsub("", ", ", ", res)), sep = "\n")

PROJCS["Stereographic_North_Pole", GEOGCS["Bessel 1841", DATUM["D_unknown", SPHEROID["bessel", 6377397.155, 299.1528128]], PRIMEM["Greenwich", 0], UNIT["Degree", 0.017453292519943295]], PROJECTION["Stereographic_North_Pole"], PARAMETER["standard_parallel_1", 52.15616055555555], PARAMETER["central_meridian", 5.38763888888889], PARAMETER["scale_factor", 0.9999079], PARAMETER["false_easting", 155000], PARAMETER["false_northing", 463000], UNIT["Meter", 1]]
If you think CRS are muddled, you are right, like time zones and daylight saving time in at least two dimensions
CRS are muddled

- If you think CRS are muddled, you are right, like time zones and daylight saving time in at least two dimensions
- But they are the key to ensuring positional interoperability, and “mashups” — data integration using spatial position as an index must be able to rely on data CRS for integration integrity
If you think CRS are muddled, you are right, like time zones and daylight saving time in at least two dimensions.

But they are the key to ensuring positional interoperability, and “mashups” — data integration using spatial position as an index must be able to rely on data CRS for integration integrity.

The situation is worse than TZ/DST because there are lots of old maps around, with potentially valuable data; finding correct CRS values takes time.
CRS are muddled

- If you think CRS are muddled, you are right, like time zones and daylight saving time in at least two dimensions.
- But they are the key to ensuring positional interoperability, and “mashups” — data integration using spatial position as an index must be able to rely on data CRS for integration integrity.
- The situation is worse than TZ/DST because there are lots of old maps around, with potentially valuable data; finding correct CRS values takes time.
- On the other hand, old maps and odd choices of CRS origins can have their charm ...
Reading vectors

- GIS vector data are points, lines, polygons, and fit the equivalent `sp` classes
Reading vectors

- GIS vector data are points, lines, polygons, and fit the equivalent \texttt{sp} classes
- There are a number of commonly used file formats, all or most proprietary, and some newer ones which are partly open
Reading vectors

- GIS vector data are points, lines, polygons, and fit the equivalent \texttt{sp} classes.
- There are a number of commonly used file formats, all or most proprietary, and some newer ones which are partly open.
- GIS are also handing off more and more data storage to DBMS, and some of these now support spatial data formats.
GIS vector data are points, lines, polygons, and fit the equivalent \texttt{sp} classes.

There are a number of commonly used file formats, all or most proprietary, and some newer ones which are partly open.

GIS are also handing off more and more data storage to DBMS, and some of these now support spatial data formats.

Vector formats can also be converted outside R to formats that are easier to read.
GIS vector data can be either topological or spaghetti — legacy GIS was topological, desktop GIS spaghetti
GIS vector data can be either topological or spaghetti — legacy GIS was topological, desktop GIS spaghetti.

*sp* classes are not bad spaghetti, but no checking of lines or polygons is done for errant topology.
GIS vector data can be either topological or spaghetti — legacy GIS was topological, desktop GIS spaghetti

sp classes are not bad spaghetti, but no checking of lines or polygons is done for errant topology

A topological representation in principal only stores each point once, and builds arcs (lines between nodes) from points, polygons from arcs — GRASS 6 has a nice topological model
GIS vector data can be either topological or spaghetti — legacy GIS was topological, desktop GIS spaghetti

sp classes are not bad spaghetti, but no checking of lines or polygons is done for errant topology

A topological representation in principal only stores each point once, and builds arcs (lines between nodes) from points, polygons from arcs — GRASS 6 has a nice topological model

Only RArcInfo tries to keep some traces of topology in importing legacy ESRI ArcInfo binary vector data (or e00 format data) — maps uses topology because that was how things were done then
Reading shapefiles

- The ESRI ArcView and now ArcGIS standard(ish) format for vector data is the shapefile, with at least a DBF file of data, an SHP file of shapes, and an SHX file of indices to the shapes; an optional PRJ file is the CRS.
The ESRI ArcView and now ArcGIS standard(ish) format for vector data is the shapefile, with at least a DBF file of data, an SHP file of shapes, and an SHX file of indices to the shapes; an optional PRJ file is the CRS.

Many shapefiles in the wild do not meet the ESRI standard specification, so hacks are unavoidable unless a full topology is built.
The ESRI ArcView and now ArcGIS standard(ish) format for vector data is the shapefile, with at least a DBF file of data, an SHP file of shapes, and an SHX file of indices to the shapes; an optional PRJ file is the CRS.

Many shapefiles in the wild do not meet the ESRI standard specification, so hacks are unavoidable unless a full topology is built.

Both maptools and shapefiles contain functions for reading and writing shapefiles; they cannot read the PRJ file, but do not depend on external libraries.
The ESRI ArcView and now ArcGIS standard(ish) format for vector data is the shapefile, with at least a DBF file of data, an SHP file of shapes, and an SHX file of indices to the shapes; an optional PRJ file is the CRS.

Many shapefiles in the wild do not meet the ESRI standard specification, so hacks are unavoidable unless a full topology is built.

Both maptools and shapefiles contain functions for reading and writing shapefiles; they cannot read the PRJ file, but do not depend on external libraries.

There are many valid types of shapefile, but they sometimes occur in strange contexts — only some can be happily represented in R so far.
> library(maptools)
> list.files()

[1] "scot_BNG.dbf" "scot_BNG.prj"
[3] "scot_BNG.shp" "scot_BNG.shx"

> getinfo.shape("scot_BNG.shp")
Shapefile type: Polygon, (5), # of Shapes: 56

> scot <- readShapePoly("scot_BNG.shp")

There are readShapePoly, readShapeLines, and readShapePoints functions in the maptools package, and in practice they now handle a number of infelicities. They do not, however, read the CRS, which can either be set as an argument, or updated later with the proj4string method.
Reading vectors: rgdal

Using the OGR vector part of the Geospatial Data Abstraction Library lets us read shapefiles like other formats for which drivers are available. It also supports the handling of CRS directly, so that if the imported data have a specification, it will be read. OGR formats differ from platform to platform — the next release of rgdal will include a function to list available formats. Use FWTools to convert between formats.

```r
> scot1 <- readOGR(dsn = ".",
+     layer = "scot_BNG")

OGR data source with driver: ESRI Shapefile
Source: ".", layer: "scot_BNG"
with 56 rows and 13 columns

> cat(strwrap(proj4string(scot1)),
+     sep = "\n")

+proj=tmerc +lat_0=49 +lon_0=-2
+k=0.9996012717 +x_0=400000
+y_0=-100000 +ellps=airy
+units=m +no_defs
```
There are very many raster and image formats; some allow only one band of data, others think data bands are RGB, while yet others are flexible.
There are very many raster and image formats; some allow only one band of data, others think data bands are RGB, while yet others are flexible.

There is a simple `readAsciiGrid` function in `maptools` that reads ESRI Arc ASCII grids into `SpatialGridDataFrame` objects; it does not handle CRS and has a single band.
Reading rasters

- There are very many raster and image formats; some allow only one band of data, others think data bands are RGB, while yet others are flexible.
- There is a simple `readAsciiGrid` function in `maptools` that reads ESRI Arc ASCII grids into `SpatialGridDataFrame` objects; it does not handle CRS and has a single band.
- Much more support is available in `rgdal` in the `readGDAL` function, which — like `readOGR` — finds a usable driver if available and proceeds from there.
There are very many raster and image formats; some allow only one band of data, others think data bands are RGB, while yet others are flexible.

There is a simple `readAsciiGrid` function in `maptools` that reads ESRI Arc ASCII grids into `SpatialGridDataFrame` objects; it does not handle CRS and has a single band.

Much more support is available in `rgdal` in the `readGDAL` function, which — like `readOGR` — finds a usable driver if available and proceeds from there.

Using arguments to `readGDAL`, subregions or bands may be selected, which helps handle large rasters.
Reading rasters: rgdal

> getGDALDriverNames()$name

```
[1] AAIGrid  ADRG   AIG    AirSAR  BMP    BSB    BT    CEOS
[9] COASP   COSAR  CPG    DIMAP  DIPEx  DOQ1   DOQ2  DTED
[17] EHdr    ELAS   ENVI   ERS    ESAT   FAST   FIT   FujiBAS
[25] GenBin  GFF    GIF    GMT    GS7BG  GSAG   GSBG  GSC
[33] GTiff   GXF    HDF4   HDF4Image HDF5   HDF5Image HFA   HTTP
[41] IDA    ILWIS  INGR   ISIS2  ISIS3  JAXAPALSAR JDEM  JPEG
[49] JPEG2000 L1B    LAN    Leveller MEM    MFF    MFF2  NDF
[57] netCDF  NITF   OGD1   PAux   PCIDSK  PCRaster PDS   PNG
[65] PNM     RIK    RMF    RPFTOC RS2    RST    SAR_CEOS SDTS
[73] SGI     SRTMHGT Terragen TSX    USGSDEM VRT    WCS   WMS
[81] XPM
```

81 Levels: AAIGrid ADRG AIG AirSAR BMP BSB BT CEOS COASP COSAR CPG DIMAP DIPEx DOQ1 DOQ2 ... XPM

> list.files()

```
[1] "SP27GTIF.TIF"
```

> SP27GTIF <- readGDAL("SP27GTIF.TIF")

SP27GTIF.TIF has GDAL driver GTiff
and has 929 rows and 699 columns
This is a single band GeoTiff, mostly showing downtown Chicago; a lot of data is available in geotiff format from US public agencies, including Shuttle radar topography mission seamless data — we’ll get back to this later.

```r
> image(SP27GTIF, col = grey(1:99/100),
       axes = TRUE)
```
> summary(SP27GTIF)

Object of class SpatialGridDataFrame
Coordinates:

min        max
x  681480  704407.2
y  1882579 1913050.0
Is projected: TRUE
proj4string:

[+proj=tmerc +lat_0=36.66666666666666 +lon_0=-88.33333333333333 +k=0.99997499999999999
  +x_0=152400.3048006096 +y_0=0 +ellps=clrk66 +datum=NAD27 +to_meter=0.3048006096012192
  +no_defs +nadgrids=@conus,@alaska,@ntv2_0.gsb,@ntv1_can.dat]
Number of points: 2
Grid attributes:

  cellcentre.offset cellsize cells.dim
  x   681496.4  32.8  699
  y  1882595.2  32.8  929
Data attributes:

        Min. 1st Qu.  Median   Mean 3rd Qu.   Max.
4.0       78.0    104.0  115.1  152.0  255.0
In `rgdal`, `writeGDAL` can write for example multi-band GeoTiffs, but there are fewer write than read drivers; in general CRS and geogreferencing are supported — see `gdalDrivers`
Writing objects

- In **rgdal**, `writeGDAL` can write for example multi-band GeoTiffs, but there are fewer write than read drivers; in general CRS and geogreferencing are supported — see `gdalDrivers`
- The **rgdal** function `writeOGR` can be used to write vector files, including those formats supported by drivers, including now KML — see `ogrDrivers`
In `rgdal`, `writeGDAL` can write for example multi-band GeoTiffs, but there are fewer write than read drivers; in general CRS and geogreferencing are supported — see `gdalDrivers`

- The `rgdal` function `writeOGR` can be used to write vector files, including those formats supported by drivers, including now KML — see `ogrDrivers`

- External software (including different versions) tolerate output objects in varying degrees, quite often needing tricks - see mailing list archives
Writing objects

- In **rgdal**, `writeGDAL` can write for example multi-band GeoTiffs, but there are fewer write than read drivers; in general CRS and geogreferencing are supported — see `gdalDrivers`.

- The **rgdal** function `writeOGR` can be used to write vector files, including those formats supported by drivers, including now KML — see `ogrDrivers`.

- External software (including different versions) tolerate output objects in varying degrees, quite often needing tricks - see mailing list archives.

- In **maptools**, there are functions for writing **sp** objects to shapefiles — `writePolyShape`, etc., as Arc ASCII grids — `writeAsciiGrid`, and for using the R PNG graphics device for outputting image overlays for Google Earth.
GIS interfaces

GIS interfaces can be as simple as just reading and writing files — loose coupling, once the file formats have been worked out, that is...
GIS interfaces

- GIS interfaces can be as simple as just reading and writing files — loose coupling, once the file formats have been worked out, that is.

- Loose coupling is less of a burden than it was with smaller, slower machines, which is why the **GRASS** 5 interface was tight-coupled, with R functions reading from and writing to the GRASS database directly.
GIS interfaces

- GIS interfaces can be as simple as just reading and writing files — loose coupling, once the file formats have been worked out, that is

- Loose coupling is less of a burden than it was with smaller, slower machines, which is why the GRASS 5 interface was tight-coupled, with R functions reading from and writing to the GRASS database directly

- The GRASS 6 interface `spgrass6` on CRAN also runs R within GRASS, but uses intermediate temporary files; the package is under development but is quite usable
GIS interfaces

- GIS interfaces can be as simple as just reading and writing files — loose coupling, once the file formats have been worked out, that is.

- Loose coupling is less of a burden than it was with smaller, slower machines, which is why the **GRASS** 5 interface was tight-coupled, with R functions reading from and writing to the GRASS database directly.

- The **GRASS** 6 interface **spgrass6** on CRAN also runs R within GRASS, but uses intermediate temporary files; the package is under development but is quite usable.

- There is an interface for the SAGA GIS implemented in package **RSAGA** (Windows only!)
GIS interfaces

- GIS interfaces can be as simple as just reading and writing files — loose coupling, once the file formats have been worked out, that is.
- Loose coupling is less of a burden than it was with smaller, slower machines, which is why the GRASS 5 interface was tight-coupled, with R functions reading from and writing to the GRASS database directly.
- The GRASS 6 interface `spgrass6` on CRAN also runs R within GRASS, but uses intermediate temporary files; the package is under development but is quite usable.
- There is an interface for the SAGA GIS implemented in package `RSAGA` (Windows only!)
- Use has been made of COM and Python interfaces to ArcGIS; typical use is by loose coupling except in highly customised work situations. In addition, the manageR plugin for QGIS can read/write `sp`-objects from/to R.
GIS interfaces

- GIS interfaces can be as simple as just reading and writing files — loose coupling, once the file formats have been worked out, that is.

- Loose coupling is less of a burden than it was with smaller, slower machines, which is why the GRASS 5 interface was tight-coupled, with R functions reading from and writing to the GRASS database directly.

- The GRASS 6 interface `spgrass6` on CRAN also runs R within GRASS, but uses intermediate temporary files; the package is under development but is quite usable.

- There is an interface for the SAGA GIS implemented in package RSAGA (Windows only!).

- Use has been made of COM and Python interfaces to ArcGIS; typical use is by loose coupling except in highly customised work situations. In addition, the manageR plugin for QGIS can read/write sp-objects from/to R.

- The RGIS Project (http://r-gis.r-forge.r-project.org/) builds on existing software and tries to develop a set of tools to analyze, model, and map your spatial data in R.
Where to get more information and help?

- Course on *Spatial Data Analysis with R* (on which these slides are based)
  
  http://www.bias-project.org.uk/ASDARcourse/
Where to get more information and help?

- Course on *Spatial Data Analysis with R* (on which these slides are based)
  http://www.bias-project.org.uk/ASDARcourse/
- R Wiki on Spatial Data
  http://wiki.r-project.org/rwiki/doku.php?id=tips:spatial-data
Writing objects

GIS interfaces

Where to get more information and help?

- Course on *Spatial Data Analysis with R* (on which these slides are based)
  http://www.bias-project.org.uk/ASDARcourse/
- R Wiki on Spatial Data
  http://wiki.r-project.org/rwiki/doku.php?id=tips:spatial-data
- R-Sig-Geo mailing list
  https://stat.ethz.ch/mailman/listinfo/r-sig-geo
Where to get more information and help?

- Course on *Spatial Data Analysis with R* (on which these slides are based)
  http://www.bias-project.org.uk/ASDARcourse/
- R Wiki on Spatial Data
  http://wiki.r-project.org/rwiki/doku.php?id=tips:spatial-data
- R-Sig-Geo mailing list
  https://stat.ethz.ch/mailman/listinfo/r-sig-geo
- Book on *Spatial Data Analysis with R*
  http://www.asdar-book.org