Using R with FOSS4G, in particular with GRASS: Representing Spatial Data in R

Roger Bivand

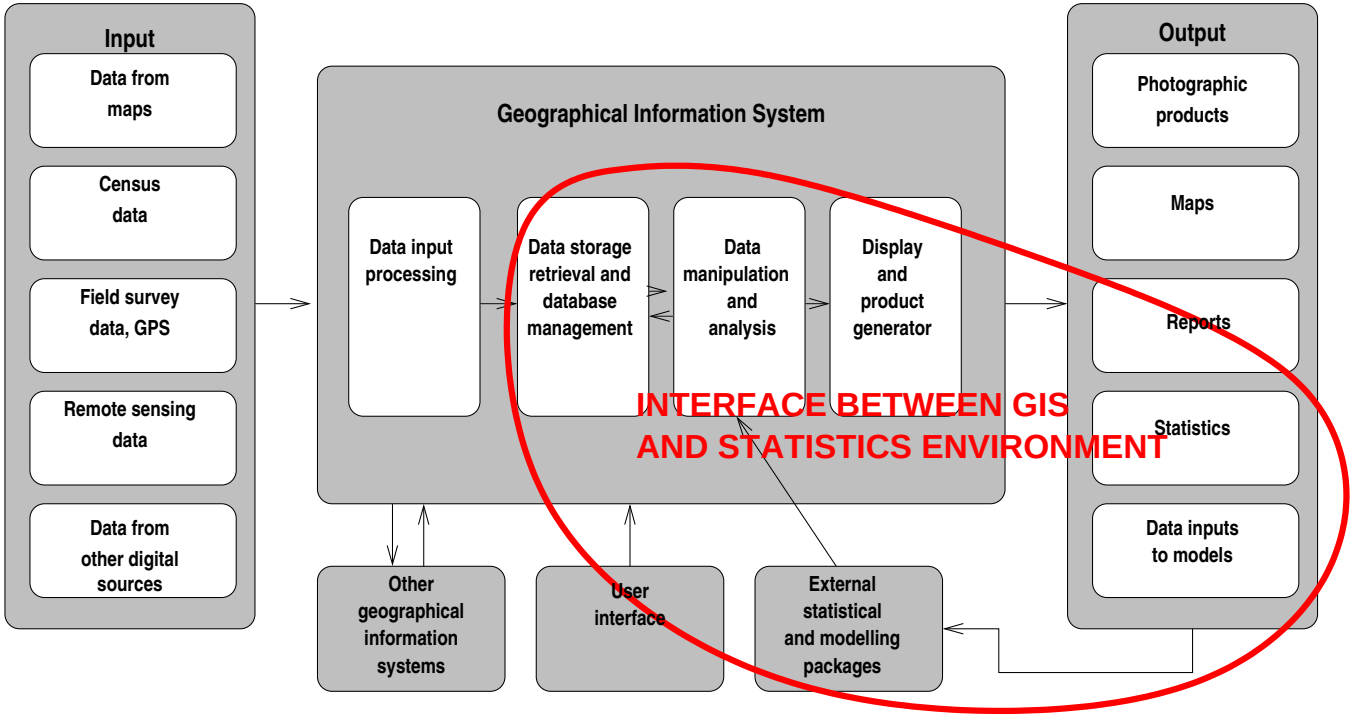
Norges Handelshøyskole
Bergen, Norway;
Roger.Bivand@nhh.no; Roger.Bivand@R-project.org

14:00 — 14:30, 12 September 2006

Why do data analysis on spatial data?

- The key reasons for doing data analysis on spatial data are to summarise the data, to visualise such summaries, and possibly to infer from it, to model and to predict using it.
- Doing exploratory data analysis on spatial data is often a helpful way of carrying out data cleaning, of getting closer to the data
- While a lot of spatial data handling quite naturally stays with the "what is where (when)" question, the "why the where (when)" question can give greater insight into what is driving the data
- So we do data analysis on spatial data when we are interested in processes driving the data

Data analysis and GIS



Why use R to do data analysis?

- Statisticians — the professionals in data analysis — speak S; R is the Free Software implementation of that language
- Statisticians implement data analysis methods using R (among others), and value access to source code — the useR to developeR continuum matters
- The R project is highly extensible by design, and support through the community and the Comprehensive R Archive Network (CRAN) for contributed packages is excellent
- R lets you get as close to the data as you need, it is an excellent environment for prototyping, it is embeddable too (in Python, in PostgreSQL, in (sorry) (D)COM)

What is CRAN?

- The Comprehensive R Archive Network (CRAN) is where the R engine can be found (source and some binaries) — there are many mirrors
- The engine ships with a handful of base packages, and a similar number of recommended packages providing a wide range of graphics and statistics functionality
- CRAN is also the main repository for contributed packages (source and some binaries), including the ones used for spatial data handling and analysis; packages are thoroughly checked before being made available from CRAN
- Packages on CRAN are also used for quality checks on the R engine, to see whether code changes in the engine break packages

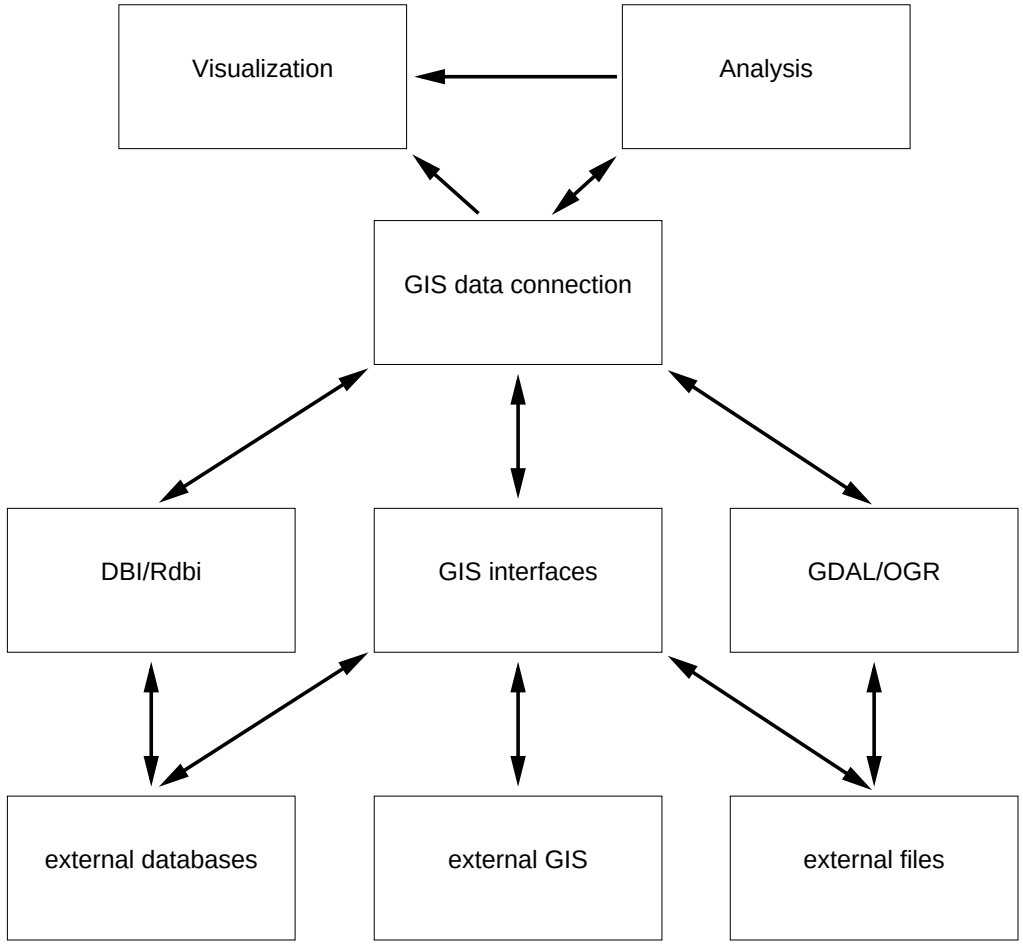
Workshop infrastructure

- Task views are one of the nice innovations on CRAN that help navigate in the jungle of contributed packages — the Spatial task view is a useful resource
- The task view is also a point of entry to the Rgeo website hosted off CRAN, and updated quite often; it tries to mention in more detail contributed packages for spatial data analysis
- It also provides a link to the **sp** development area on Sourceforge, with CVS access to **sp** and an R repository of source and Windows binary packages not yet ready for CRAN
- Finally, it links to the R-sig-geo mailing list, which is the prefered place to ask questions about analysing spatial/geographical data

Spatial data in R

- At base, spatial data can be held in GIS processes, in external databases, or in external files
- This needs to be interfaced with native R data structures in an active R process (R is single-threaded)
- The different external processes and formats need handling in different ways (although GDAL/OGR provides useful abstraction mechanisms)
- The **sp** package has been written as a spatial/GIS data connection to provide shared spatial data structures both for input/output and for use in analysis and visualisation

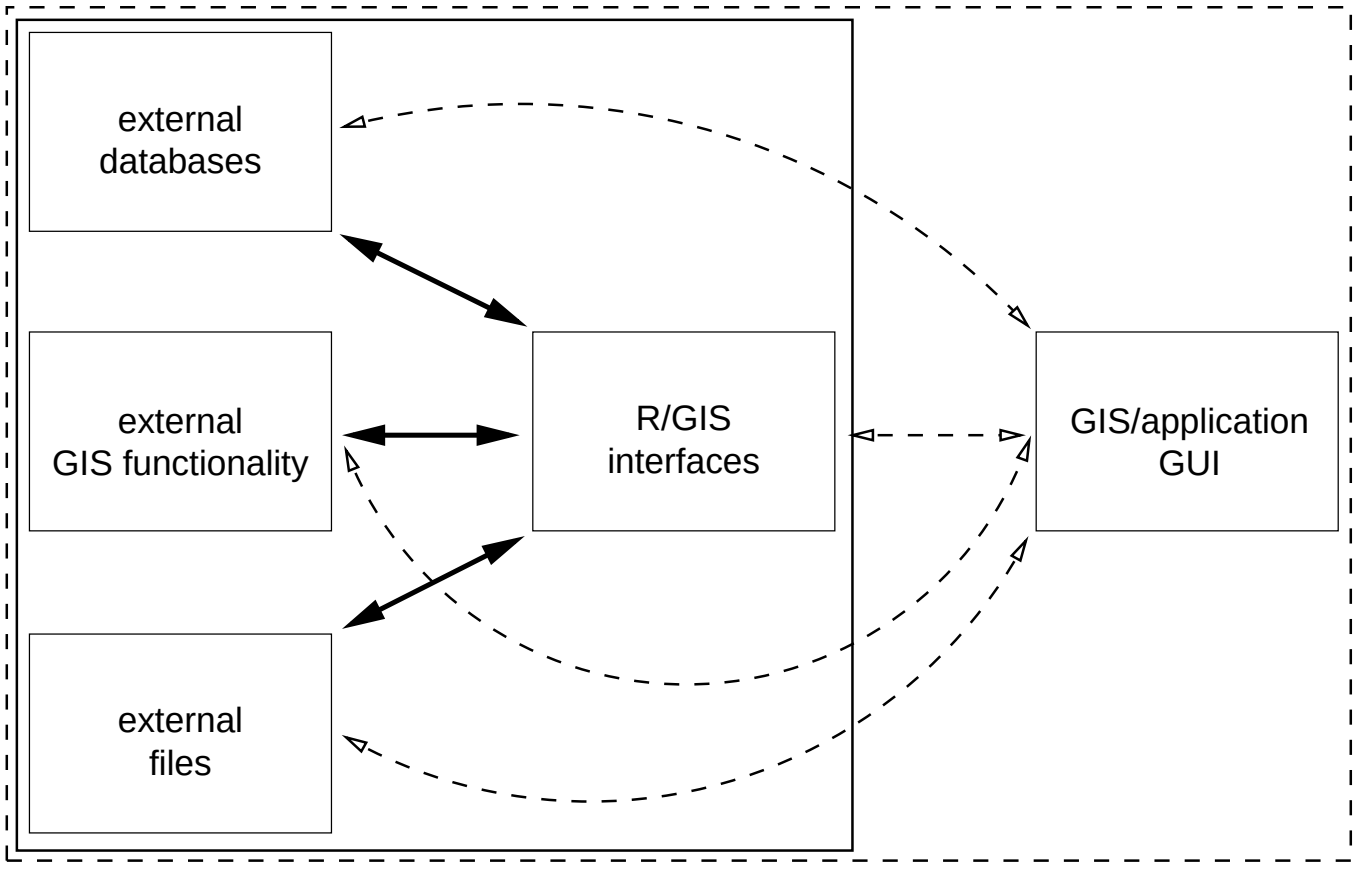
Data flow for spatial data in R



Should R be a GIS?

- A serious question which is raised repeatedly is the extent to which R contributed packages should provide GIS functionality
- Questioners on the list have asked about remote sensing, Tomlin-style cartographic modelling, vector cleaning, and other topics
- It does seem better to be minimalist, only providing routes to other software, rather than trying to do too much poorly
- R can however be integrated as part of a general data flow, involving spatial data and a front-end user interface; Rpad is a nice example

Integrating R as middleware



Short aside on classes, objects, and methods in R

- R has two class/method styles, old-style aka. `S3` and new style aka. `S4`; in neither do methods belong to classes, and no class/method system is pervasive
- Most objects returned by analysis functions are `S3` objects, lists with components and at least a `class` attribute, but no class definition
- `S4` classes are formally defined, removing the possibility that users may corrupt the contents of an object by accident
- Method dispatch for `S3` objects is on the class of the first function argument, for `S4` objects on one or more class signatures

Spatial objects

- The foundation here 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

Spatial points

- The most basic spatial data class 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

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

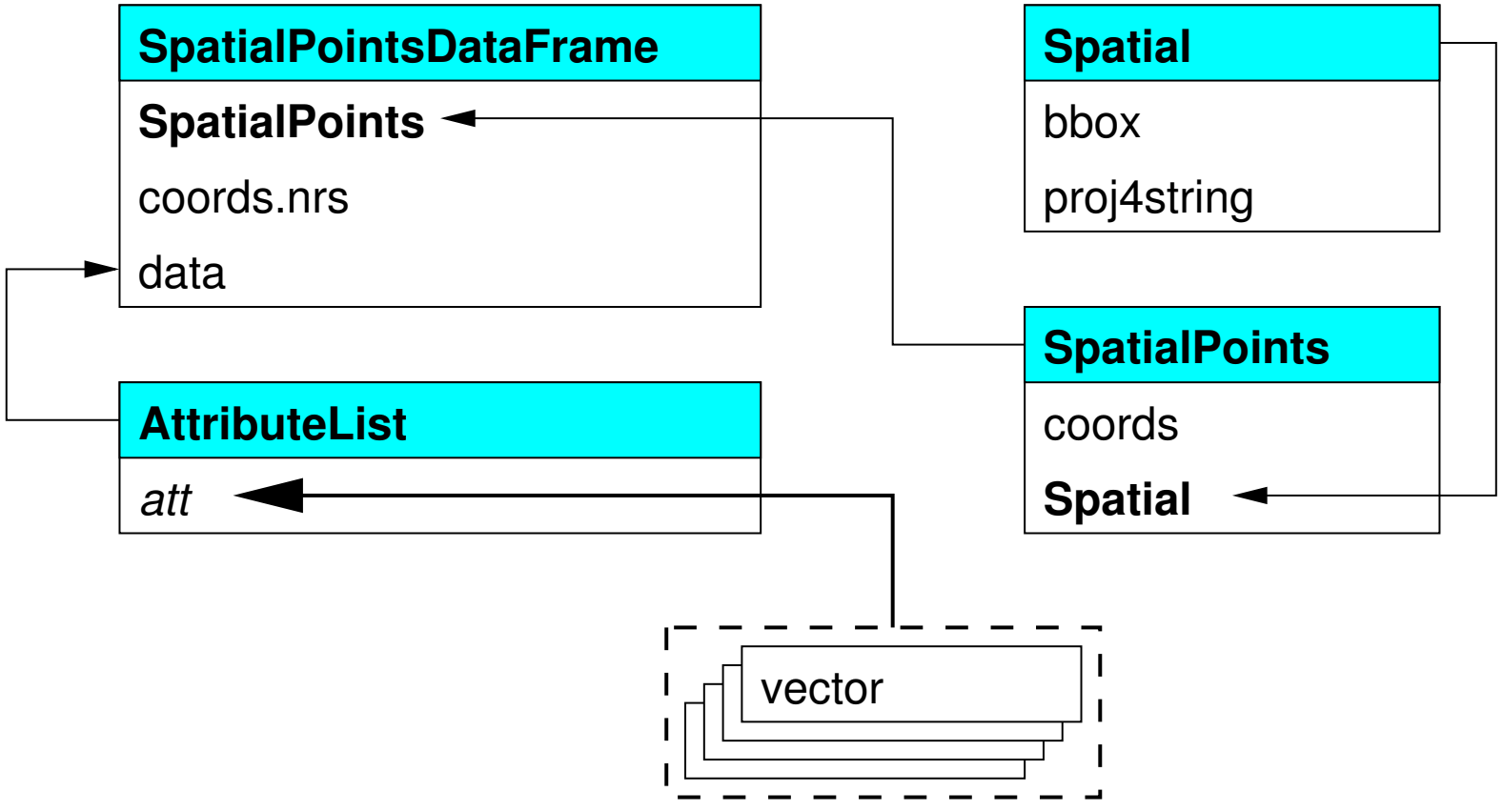
```
> library(sp)
> data(meuse)
> coords <- SpatialPoints(meuse[, c("x", "y")])
> summary(coords)

Object of class SpatialPoints
Coordinates:
     min    max
x 178605 181390
y 329714 333611
Is projected: NA
proj4string : [NA]
Number of points: 155
```

Spatial points

Now we'll add the original data frame to make a `SpatialPointsDataFrame` object. Many methods for standard data frames just work with `SpatialPointsDataFrame` objects.

```
> 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 | 12223333344444455666677778888899999999
   2 | 00000001111111222223344455566667888002233445578 8
   4 | 0001223567700145555678 9
   6 | 01144678890012455678889
   8 | 0133113
  10 | 235604469
  12 | 8
  14 | 5357
  16 | 7
  18 | 4
```

Spatial points classes and their slots



Data frames

- Note that `SpatialPointsDataFrame` objects at present have `data` slots of class `AttributeList`; this will change from R 2.4.0 to regular `data.frame`
- Data frames are the workhorses of much analysis and visualisation in S— they are lists of equal-length vectors of (possibly) different types of data
- List components are accessed either by name or number, often using the `$` operator to access by name
- `Spatial*DataFrame` family objects behave in most cases like data frames

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

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:

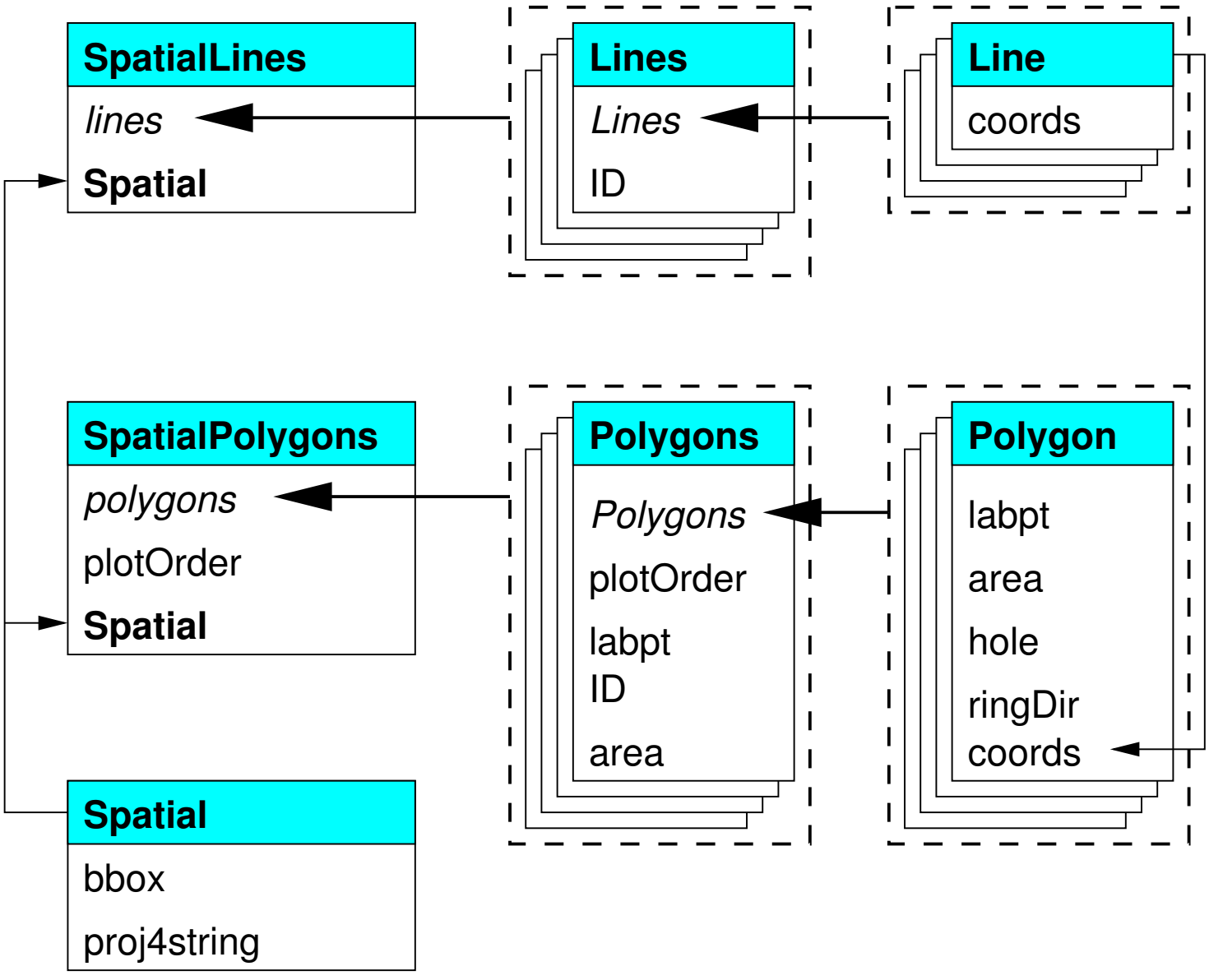
```
> data(meuse.riv)
> str(meuse.riv)

 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:

```
> 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 Polygons classes and slots



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 pixels

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:

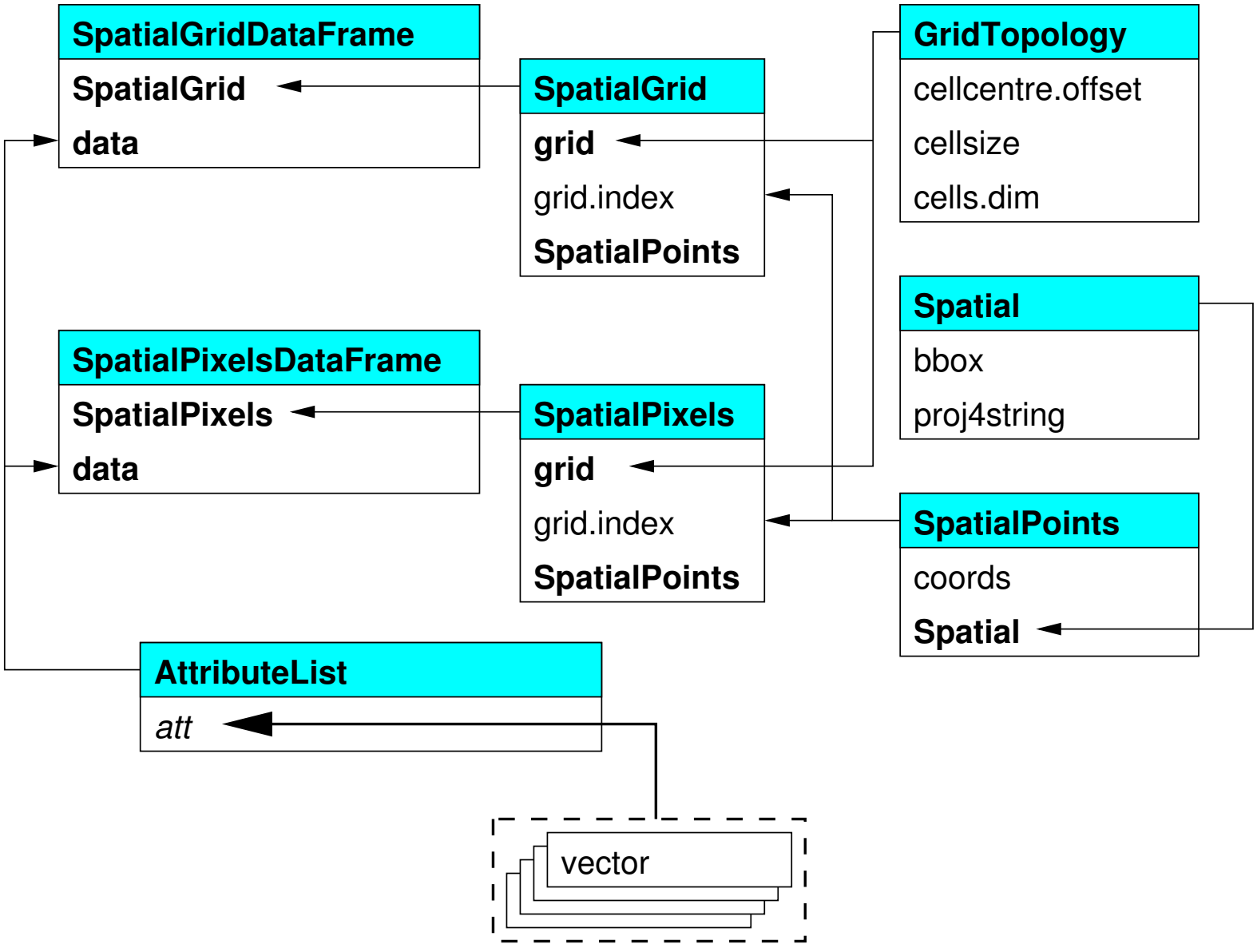
```
> data(meuse.grid)
> coords <- SpatialPixels(SpatialPoints(meuse.grid[, c("x",
+     "y")]))
> meuseg1 <- SpatialPixelsDataFrame(coords, meuse.grid)
> names(meuseg1)

[1] "x"      "y"      "part.a" "part.b" "dist"   "soil"   "ffreq"

> slot(meuseg1, "grid")

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

> object.size(meuseg1)

[1] 227712

> dim(slot(meuseg1, "data"))

[1] 3103    7
```

Spatial grids

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:

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

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

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

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

> object.size(meuseg2)

[1] 426060

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

[1] 8112    7
```

Spatial grid and pixels classes and their slots



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:

data type	class	attributes	extends
points	`SpatialPoints`	none	`Spatial`
points	`SpatialPointsDataFrame`	`AttributeList`	`SpatialPoints`
pixels	`SpatialPixels`	none	`SpatialPoints`
pixels	`SpatialPixelsDataFrame`	`AttributeList`	`SpatialPixels` `SpatialPointsDataFrame`
full grid	`SpatialGrid`	none	`SpatialPixels`
full grid	`SpatialGridDataFrame`	`AttributeList`	`SpatialGrid`
line	`Line`	none	
lines	`Lines`	none	`Line` list
lines	`SpatialLines`	none	`Spatial`, `Lines` list
lines	`SpatialLinesDataFrame`	`data.frame`	`SpatialLines`
polygon	`Polygon`	none	`Line`
polygons	`Polygons`	none	`Polygon` list
polygons	`SpatialPolygons`	none	`Spatial`, `Polygons` list
polygons	`SpatialPolygonsDataFrame`	`data.frame`	`SpatialPolygons`

Methods provided by **sp**

This table summarises the methods provided by **sp**:

method	what it does
`[`	select spatial items (points, lines, polygons, or rows/cols from a grid) and/or attributes variables
`$, $<-, [[, [[<-`	retrieve, set or add attribute table columns
`spsample`	sample points from a set of polygons, on a set of lines or from a gridded area
`bbox`	get the bounding box
`proj4string`	get or set the projection (coordinate reference system)
`coordinates`	set or retrieve coordinates
`coerce`	convert from one class to another
`overlay`	combine two different spatial objects

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