Classes and methods for spatial data in R

by Edzer J. Pebesma and Roger S. Bivand

R has been used for spatial statistical analysis for a long time: packages dedicated to several areas of spatial statistics have been reviewed in Bivand and Gebhardt (2000) and Ripley (2001), and more recent packages have been published (Ribeiro and Diggle, 2001; Pebesma, 2004; Baddeley and Turner, 2005). In addition, R links to GIS have been developed (Bivand, 2000, 2005; Gómez-Rubio and López-Quílez, 2005). A task view of spatial statistical analysis is being maintained on CRAN.

Many of the spatial packages use their own data structures (classes) for the spatial data they need or create, and provide methods, e.g. for plotting them. However, up to now R has been lacking a full-grown coherent set of classes and methods for the major spatial data types: points, lines, polygons, and grids. Package **sp**, available on CRAN since May 2005, has the ambition to fill this gap.

We believe there is a need for **sp** because

(i) with support for a single package for spatial data, it is much easier to go from one spatial statistics package to another. Its classes are either supported directly by the packages, reading and writing data in the new spatial classes, or indirectly e.g. by supplying data conversion between **sp** and the package in an interface package. This requires one-to-many links, which are easier to provide and maintain than many-to-many links,

(ii) the new package provides a well-tested set of methods (functions) for plotting, printing, subsetting and summarizing spatial objects, or combining (overlay) spatial data types,

(iii) packages with interfaces to GIS and geographic (re)projection code support the new classes,

(iv) the new package provides Lattice plots, conditioning plots, plot methods that combine points, lines, polygons and grids with map elements (reference grids, scale bars, north arrows), degree symbols (52°N) in axis labels, etc.

This article introduces the classes and methods provided by **sp**, discusses some of the implementation details, and the state and potential of linking **sp** to other packages.

Classes and implementation

The spatial classes in **sp** include points, lines, polygons and grids; each of these structures can have multiple attribute variables, describing what is actually registered (e.g. measured) for a certain location or area. Single entries in the attribute table may correspond to multiple lines or polygons; this is useful because e.g. administrative regions may consist of several polygons (mainland, islands). Polygons may further contain holes (e.g. a lake), polygons in holes (island in lake), holes in polygons in holes, etc. Each spatial object registers its coordinate reference system when this information is available. This allows transformations between latitude/longitude and/or various projections.

The **sp** package uses S4 classes, allowing for the validation of internal consistency. All `Spatial` classes in **sp** derive from an abstract class `Spatial`, which only holds a bounding box and the projection or coordinate reference system. All classes are S4 classes, so objects may be created by function `new`, but we typically envisage the use of helper functions to create instances. For instance, to create a `SpatialPointsDataFrame` object from the `meuse` data.frame, we might use:

```
> library(sp)
> data(meuse)
> coords = SpatialPoints(meuse[c("x", "y")])
> meuse = SpatialPointsDataFrame(coords, meuse)
> plot(meuse, pch=1, cex = .05*sqrt(meuse$zinc))
```

the plot of which is shown in figure 1.

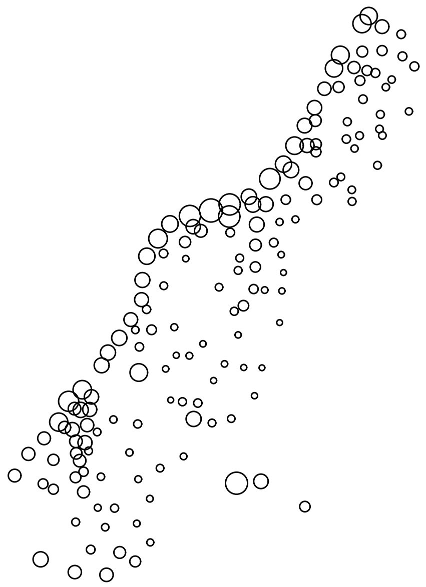

Figure 1: Bubble plot of top soil zinc concentration

The function `SpatialPoints()` creates a `SpatialPoints` object. `SpatialPointsDataFrame()` merges this object with an attribute table to creates an object of class `SpatialPointsDataFrame`.

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*

* by direct extension; ** by setIs() relationship;

Table 1: Overview of the spatial classes provided by **sp**. Classes with topology only are extended by classes with attributes.

An alternative to the calls to SpatialPoints() and SpatialPointsDataFrame() above is to use coordinates<-, as in

```
> coordinates(meuse) = c("x", "y")
```

Reading polygons from a shapefile (a commonly used format by GIS) can be done by the readShapePoly function in **maptools**, which depends on **sp**. It returns an object of class SpatialPolygonsDataFrame, for which the plot method is in **sp**. An example that uses a shapefile provided by package **maptools** is:

```
> library(maptools)
> fname = system.file("shapes/sids.shp",
+    package="maptools")
> p4s = CRS("+proj=longlat +datum=NAD27")
> nc = readShapePoly(fname, proj4string=p4s)
> plot(nc, axes = TRUE, col=grey(1-nc$SID79/57))
```

for which the plot is shown in figure 2. The function CRS defines the coordinate reference system.

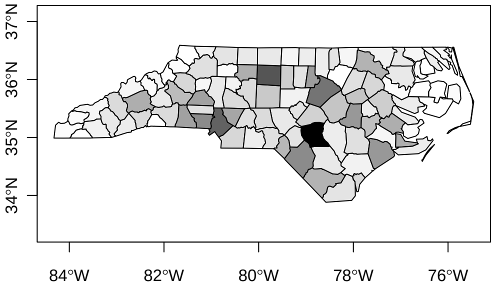

Figure 2: North Carolina sudden infant death (SID) cases in 1979 (white 0, black 57)

An overview of the classes for spatial data in **sp** is given in Table 1. It shows that the points, pixels and grid classes are highly related. We decided to implement two classes for gridded data: SpatialPixels for unordered points on a grid, SpatialGrid for a full ordered grid, e.g. needed by the image command. SpatialPixels store coordinates but may be efficient when a large proportion of the bounding box has missing values because it is outside the study area ('sparse' images).

A single Polygons object may consist of multiple Polygon elements, which could be islands, or holes, or islands in holes, etc. Similarly, a Lines entity may have multiple Line segments: think of contour lines as an example.

This table also shows that several classes store their attributes in objects of class AttributeList, instead of data.frames, as the class name would suggest. An AttributeList acts mostly like a data.frame, but stores no row.names, as row names take long to compute and use much memory to store. Consider the creation of a 1000 × 1000 grid:

```
> n = 1000
> df = data.frame(expand.grid(x=1:n, y=1:n),
+    z=rnorm(n * n))
> object.size(df)
[1] 56000556 # mostly row.names!
> library(sp)
> coordinates(df) = ~x+y
> object.size(df)
[1] 16002296 # class SpatialPointsDataFrame
> gridded(df) = TRUE
> object.size(df)
[1] 20003520 # class SpatialPixelsDataFrame
> fullgrid(df) = TRUE
> object.size(df)
[1] 8003468 # class SpatialGridDataFrame
```

Using data.frame as attribute tables for moderately sized grids (e.g. Europe on a 1 km × 1 km grid) became too resource intensive on a computer with 2 Gb RAM, and we decided to leave this path.

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, using the simple sampling methods given in Ripley (1981)
`spplot`	lattice (Trellis) plots of spatial variables (figure 3; see text)
`bbox`	give the bounding box
`proj4string`	get or set the projection (coordinate reference system)
`coordinates`	set or retrieve coordinates
`polygons`	set or retrieve polygons
`gridded`	verify whether an object is a grid, or convert to a gridded format
`dimensions`	get the number of spatial dimensions
`coerce`	convert from one class to another
`transform`	(re-)project spatial coordinates (uses spproj)
`overlay`	combine two different spatial objects (see text)
`recenter`	shift or re-center geographical coordinates for a Pacific view

Table 2: Methods provided by package **sp**

For classes `SpatialLinesDataFrame` and `SpatialPolygonsDataFrame` we expect that the spatial entities corresponding to each row in the attribute table dominate memory usage; for the other classes it is reversed. If row names are needed in a points or gridded data structure, they must be stored as a (character) attribute column.

`SpatialPolygons` and `SpatialLines` objects have IDs stored for each entity. For these classes, the ID is matched to attribute table IDs when a `Spatial*DataFrame` object is created. For points, IDs are matched when present upon creation. If the matrix of point coordinates used in creating a `SpatialPointsDataFrame` object has rownames, these are matched against the data frame row.names, and subsequently discarded. For points and both grid classes IDs are not stored because, for many points, this wastes processing time and memory. ID matching is an issue for lines and polygons when topology and attributes often come from different sources (objects, files). Points and grids are usually created from a single table, matrix, or array.

Methods

Beyond the standard `print`, `plot` and `summary` methods, methods that **sp** provides for each of the classes are shown in table 2.

Some of these methods are much for user convenience, to make the spatial data objects behave just like `data.frame` objects where possible; others are specific for spatial topology. For example, the overlay function can be used to retrieve the polygon information on point locations, or to retrieve the information for a collection of points falling inside each of the polygons. Alternatively, points and grid cells may be overlayed, resulting in a point-in-gridcell operation. Other overlay methods may be implemented when they seem useful. Method `summary` shows the data class, bounding box, projection information, number of spatial entities, and a summary of the attribute table if present.

Trellis maps

When plotting maps one usually needs to add spatial reference elements such as administrative boundaries, coastlines, cities, rivers, scale bars or north arrows. Traditional plots allow the user to add components to a plot, by using `lines`, `points`, `polygon` or `text` commands, or by using `add=TRUE` in `plot` or `image`. Package **sp** provides traditional plot methods for e.g. `points`, `lines` and `image`, but has in addition an `spplot` command which lets you plot Trellis plots (provided by the **lattice** package) to produce plots with multiple maps. Method `spplot` allows for addition of spatial elements and reference elements on all or a selection of the panels.

The example in figure 3 is created by:

```
> arrow = list("SpatialPolygonsRescale",
+   layout.north.arrow(2),
+   offset = c(-76,34), scale = 0.5, which=2)
> spplot(nc, c("SID74","SID79"),as.table=TRUE,
+   scales=list(draw=T), sp.layout = arrow)
```

where the `arrow` object and `sp.layout` argument ensure the placement of a north arrow in the second panel only.

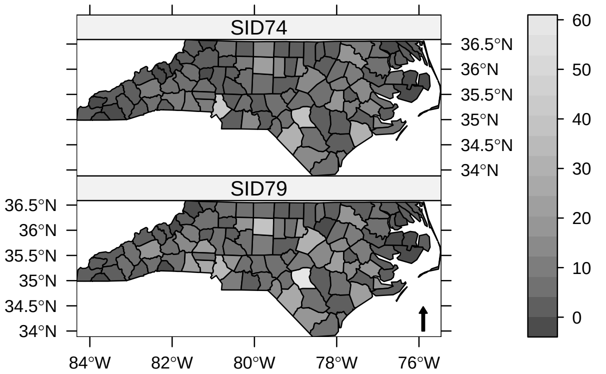

Figure 3: Trellis graph created by `spplot`

Connections to GIS

On the R-spatial web site (see below) a number of additional packages are available, linking **sp** to several external libraries or GIS:

spproj allows (re)projection of **sp** objects,

spgpc allows polygon clipping with **sp** objects, and can e.g. check whether polygons have holes,

spgrass6 read and write **sp** objects from and to a GRASS 6.0 data base,

spgdal read and write **sp** objects from gdal data sets: it provides a `"["` method that reads in (part of) a gdal data and returns an **sp** grid object.

It is not at present likely that a single package for interfacing external file and data source formats like the recommended package **foreign** will emerge, because of large and varying external software dependencies.

Connections to R packages

CRAN packages maintained by the **sp** authors do already use **sp** classes: package **maptools** has code to read shapefiles and Arc ASCII grids into **sp** class objects, and to write **sp** class objects to shapefiles or Arc ASCII grids. Package **gstat** can deal with points and grids, and will compute irregular block kriging estimates when a polygons object is provided as newdata (prediction 'regions').

The **splancs**, **DCluster**, and **spdep** packages are being provided with ways of handling **sp** objects for analysis, and **RArcInfo** for reading ArcInfo v. 7 GIS binary vector E00 files.

Interface packages present which convert to or from some of the data structures adopted by spatial statistics packages include **spPBS**, **spmaps**; packages under development are **spspatstat**, **spgeoR**, **spfields** and **spRandomFields**. These interface packages depend on both **sp** and the package they interface to, and provide methods to deal with **sp** objects in the target functions. Whether to introduce dependence on **sp** into packages is of course up to authors and maintainers. The interface framework, with easy installation from the R-spatial repository on SourceForge (see below), is intended to help users who need **sp** facilities in packages that do not use **sp** objects directly — for example creating an observation window for **spatstat** from polygons read from a shapefile.

Further information and future

The R-spatial home page is

`http://r-spatial.sourceforge.net/`

Announcements related to **sp** and interface packages are sent to the R-sig-geo mailing list.

A first point for obtaining more information on the classes and methods in **sp** is the package vignette. There is also a demo, obtained by

```
> library(sp)
> demo(gallery)
```

which gives the plots of the gallery also present at the R-spatial home page.

Development versions of **sp** and related packages are on cvs on sourceforge, as well as interface packages that are not (yet) on CRAN. An off-CRAN package repository with source packages and Windows binary packages is available from sourceforge as well, so that the package installation should be sufficiently convenient before draft packages reach CRAN:

```
> rSpatial = "http://r-spatial.sourceforge.net/R"
> install.packages("spproj", repos=rSpatial)
```

Acknowledgements

The authors want to thank Barry Rowlingson (whose ideas inspired many of the methods now in **sp**, see Rowlingson et al. (2003)), and Virgilio Gómez-Rubio.

Bibliography

A. Baddeley and R. Turner. Spatstat: an R package for analyzing spatial point patterns. *Journal of Statistical Software*, 12(6):1–42, 2005. URL: `www.jstatsoft.org`, ISSN: 1548-7660. 9

R. Bivand. Interfacing GRASS 6 and R. *GRASS Newsletter*, 3:11–16, June 2005. ISSN 1614-8746. 9

R. Bivand. Using the R statistical data analysis language on GRASS 5.0 GIS data base files. *Computers & Geosciences*, 26:1043–1052, 2000. 9

R. Bivand and A. Gebhardt. Implementing functions for spatial statistical analysis using the R language. *Journal of Geographical Systems*, 3(2):307–317, 2000. 9

V. Gómez-Rubio and A. López-Quílez. RArcInfo: using GIS data with R. *Computers & Geosciences*, 31: 1000–1006, 2005. 9

E. Pebesma. Multivariable geostatistics in S: the gstat package. *Computers & Geosciences*, 30:683–691, 2004. 9

P. Ribeiro and P. Diggle. geoR: A package for geostatistical analysis. *R-News*, 1(2), 2001. URL: www.r-project.org, ISSN: 1609-3631. 9

B. Ripley. Spatial statistics in R. *R-News*, 1(2), 2001. URL: www.r-project.org, ISSN: 1609-3631. 9

B. Ripley. *Spatial Statistics*. Wiley, New York, 1981. 11

B. Rowlingson, A. Baddeley, R. Turner, and P. Diggle. Rasp: A package for spatial statistics. In A. Z. K. Hornik, F. Leisch, editor, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20–22, Vienna, Austria.*, 2003. ISSN 1609-395X. 12

Edzer Pebesma
e.pebesma@geog.uu.nl

Roger Bivand
Norwegian School of Economics and Business Administration
Roger.Bivand@nhh.no

Running Long R Jobs with Condor DAG

by Xianhong Xie

For statistical computing, R and S-Plus have been around for quite some time. As our computational needs increase, the running time of our programs can get longer and longer. A simulation that runs for more than 12 hours is not uncommon these days. The situation gets worse with the need to run the program many times. If one has the access to a cluster of machines, he or she will definitely want to submit the programs to different machines, the management of the submitted jobs could prove to be quite challenging. Luckily, we have batch systems like Condor (http://www.cs.wisc.edu/condor), PBS (http://pbs.mrj.com/service.html), etc. available.

What is Condor?

Condor is a batch processing system for clusters of machines. It works not only on dedicated servers for running jobs, but also on non-dedicated machines, such as PCs. Some other popular batch processing systems include PBS, which has similar features as Condor. The results presented in this article should translate to PBS without much difficulties. But Condor provides some extra feature called checkpointing, which I will discuss later.

As a batch system, Condor can be divided into two parts: job management and resource management. The first part serves the users who have some jobs to run. And the second part serves the machines that have some computing power to offer. Condor matches the jobs and machines through the ClassAd mechanism which, as indicated by its name, works like the classified ads in the newspapers.

The machine in the Condor system could be a computer sitting on somebody else's desk. It serves not only the Condor jobs but also the primary user of the computer. Whenever someone begins using the keyboard or mouse, the Condor job running on the computer will get preempted or suspended depending on how the policy is set. For the first case, the memory image of the evicted job can be saved to some servers so that the job can resume from where it left off once a machine is available. Or the image is just discarded, and the job has to start all over again after a machine is available. The saving/resuming mechanism is called checkpointing, which is only available with the Condor Standard universe jobs. To be able to use this feature, one must have access to the object code files for the program, and relink the object code against the Condor library. There are some restrictions on operations allowed in the program. Currently these preclude creating a Standard universe version of R.

Another popular Condor universe is the Vanilla universe, under which most batch ready jobs should be able to run. One doesn't need to have access to the object code files, nor is relinking involved. The restrictions on what one can do in one's programs are much less too. Anything that runs under the Standard universe should be able to run under the Vanilla universe, but one loses the nice feature of checkpointing.

Condor supports parallel computing applications