Collaborative open source software development: the case of **sp**, a package of R class definitions for spatial data

Roger Bivand[*] Edzer J. Pebesma[†] Barry Rowlingson[‡]

June 2005

Abstract

In this presentation, we will describe the context in which shared foundation classes for spatial data in R were conceived, the way in which the design of the classes has progressed, and the current status. Provision is made for point, grid, line and polygon ring classes, with suitable methods for showing and displaying data contained in class objects.

The point, grid, line and polygon ring classes do not depend on the underlying data object. Initially, they have been associated with the attribute data, essentially a representation of a data table with equal-length numeric, logical or categorical columns. These attribute tables are added as slots (class components) to elementary spatial classes.

Finally, the paper will show how data may be read from files and written to files using the spatial foundation classes. The intention is that analysis packages migrate towards the foundation classes, either directly or through wrapper functions, to avoid unnecessary duplication, and to make it easier for users to transfer experience in spatial data analysis between packages.

1 Introduction

One of the attractive features of open source software development models is that they can be collaborative. In such cases, an informal group of contributors can form, drawn together by shared goals, interests, or just curiosity. The group need not be tightly defined, and often has no production schedule. While there may also be disadvantages to sharing software development in an open fashion, we would like to focus on some othe advantages we have found in writing a contributed package to the R data analysis environment with class definitions for spatial data.

The **sp** package provides classes and methods for dealing with spatial data in S (R and S-PLUS [1]). The spatial data structures implemented include points, lines, polygons and grids; each of them with or without attribute data. We have chosen to use S4 classes and methods style (Chambers, 1998) to allow validation of objects created. Although we mainly aim at using spatial data in the geographical (two-dimensional) domain, the data structures that have a straightforward implementation in higher dimensions (points, grids) do allow this.

In addition, the classes have a slot for projection information, which can be "not available", that is not defined, or can match the values used by the PROJ.4 library. The sp class package can be used with an accompanying package for projection and datum transformation, and this package will be used to demonstrate the implementation of inter-package dependencies.

[*]Economic Geography Section, Department of Economics, Norwegian School of Economics and Business Administration, Helleveien 30, N-5045 Bergen, Norway; Roger.Bivand@nhh.no

[†]Dept of Physical Geography, Faculty of Geosciences, Utrecht University, P.O. Box 80.115, 3508 TC Utrecht, The Netherlands e.pebesma@geog.uu.nl

[‡]Department of Mathematics and Statistics, Fylde College, Lancaster University, Lancaster LA1 4YF, United Kingdom B.Rowlingson@lancaster.ac.uk

[1]our primary efforts target R; depending on the needs, we will address S-Plus as well

Here we will first motivate the design of the new foundation classes and the choice to write them in S rather than rely on external resources. Next, the classes themselves will be described and their operation demonstrated. Finally, the use of these classes in conjunction with other packages will be shown with examples.

2 Motivation

The motivation to write this package was born at a pre-conference spatial data workshop during DSC 2003, the Distributed Statistical Computation meeting at which projects such as R are discussed. At that time, the advantage of having multiple R packages for spatial statistics seemed to be hindered by a lack of a uniform interface for handling spatial data. Each package had its own conventions on how spatial data were stored and returned.

Should we expect that a polygon is understood by packages in the same way (no, but maybe users are right in thinking that this is an unnecessary barrier)? In **splancs**, a polygon cannot have sub-polygons or holes.

A very fresh case shows this clearly: a user[2] wanted to carry out kriging prediction bounded for the states of Pennsylvania, Maryland and New York, and prepared a shapefile (as a single polygon after an earlier exchange indicated that multiple polygons did not work). The state boundaries were dissolved in ArcGIS, and the shapefile appears OK, has one polygon (but 7 parts, the mainland, Long Island, and five smaller islands — see Fig. 1):

```
> library(maptools)
```

```
Loading required package: foreign
```

```
> PANYMD <- read.shape("PANYMD_States.shp")
```

```
Shapefile type: Polygon, (5), # of Shapes: 1
```

```
> attr(PANYMD$Shapes[[1]], "nParts")
```

```
[1] 7
```

```
> plot(PANYMD)
```

Well, both **geoR** and **splancs** are from Lancaster, so **geoR** uses the point-in-polygon functions from **splancs** to see what is where, but also imposes limits on what a polygon is — no NA coordinates.

```
> xp <- Map2poly(PANYMD)
> load("geodata.Rda")
> load("kc.Rda")
> load("locations.Rda")
> library(geoR)
```

```
---------------------------------------------------------------
Functions for geostatistical data analysis
For an Introduction to geoR go to http://www.est.ufpr.br/geoR
geoR version 1.5-6 (built on 2005/05/10) is now loaded
---------------------------------------------------------------
```

But when shapefile data is read into polylist objects from the **maptools** package — lists of coordinate matrices, and pass the object to the function in **geoR** for visualising the results, we hit NAs (not available values) used in polylist objects to separate the subpolygons of a polygon. The standard reaction is then to do what the error message seems to suggest and omit the NAs — see Fig. 2:

[2]Thanks to Chloé Archinard for raising this problem and making her data available to demonstrate it.

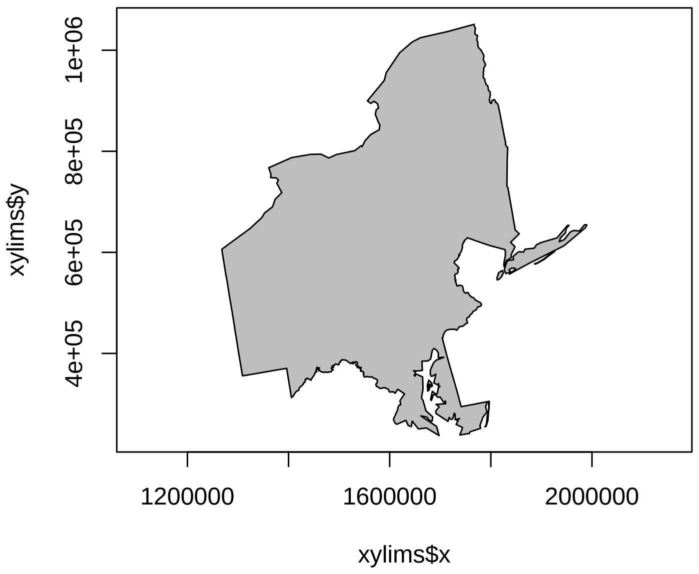

Figure 1: Polygon borders for PA, NY, MD

```
> wrong <- try(contour(kc, locations, borders = xp[[1]], filled = TRUE, coords.data = geodata$coords,
+     add = TRUE, color = terrain.colors))
```

```
Loading required package: splancs

Spatial Point Pattern Analysis Code in S-Plus

 Version 2 - Spatial and Space-Time analysis
```

```
> cat(wrong)
```

```
Error in inout(pts = locations, poly = borders, ...) :
        NA/NaN/Inf in foreign function call (arg 4)
```

```
> contour(kc, locations, borders = na.omit(xp[[1]]), filled = TRUE, coords.data = geodata$coords,
+     add = TRUE, color = terrain.colors)
```

So **splancs** polygons are a single ring with no holes, and therefore **geoR** gets confused when the data do not fit this model. For users needing to access, for example, to boundary data in typical data exchange formats, this is not a help, nor is the unpleasant fix of sticking Long Island onto the mainland — see Fig. 3:

```
> load("kc_fake.Rda")
> fake_poly <- cbind(append(xp[[1]][1:430, 1], xp[[1]][432:467, 1], 112), append(xp[[1]][1:430,
+     2], xp[[1]][432:467, 2], 112))
```

```
> contour(kc, locations, borders = fake_poly, filled = TRUE, coords.data = geodata$coords, add = TRUE,
+     color = terrain.colors)
```

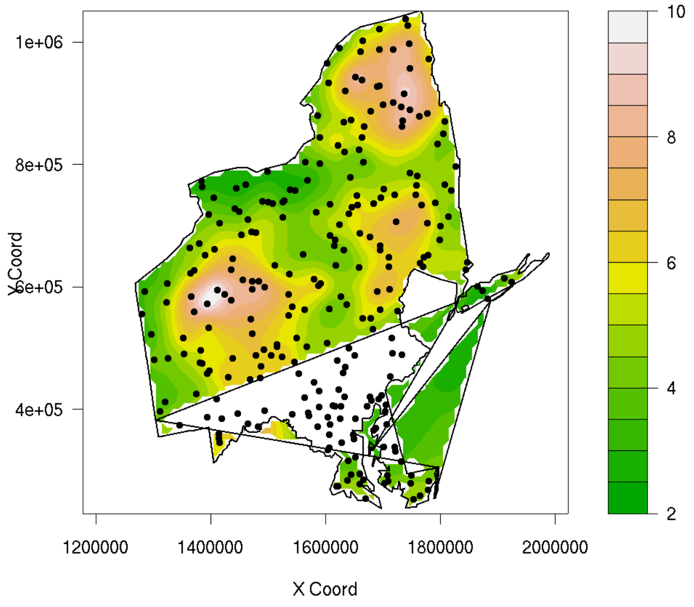

Figure 2: Not what was wanted

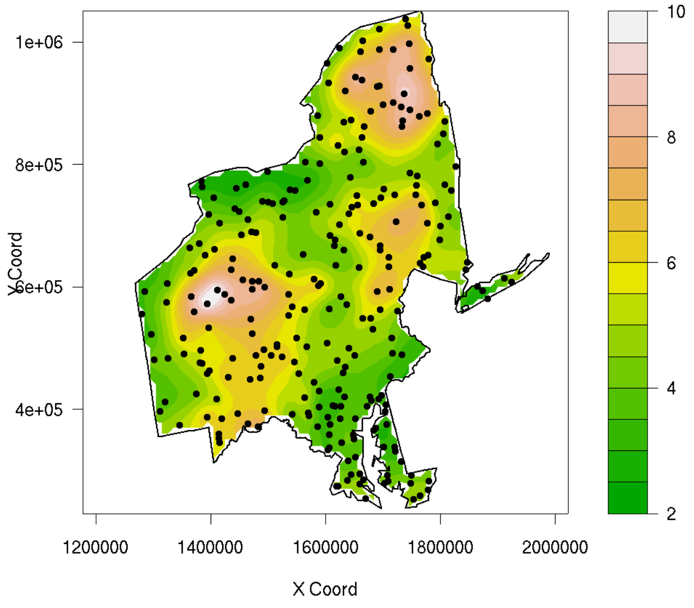

Figure 3: Well at least we got Long Island back

3 Spatial data classes

With the **sp** package, and packages supporting the classes provided here, we hope that R will become a more coherent tool for analyzing different types of spatial data; it is now available on CRAN. From the package home page, http://r-spatial.sourceforge.net/, a graph gallery with R code, and the development source tree are available, together with wrapper packages to interface other R packages.

A question needing immediate resolution is why we have not chosen to leverage the excellent work being done in other languages, such as Geotools in Java. The answer is that mechanisms for foreign language integration and object reflectance within S (R and Omegahat) have not been making progress, and many of the interfaces are very difficult to install and maintain. It seems as though some more movement on Java is coming, but both writing spatial classes and making the R-Java interface work was too challenging. Where possible, simple computational geometry functions written in C are used, but the object structure is kept to that of S. Please note that S objects and classes are far from the OO model of C++ or Java, classes do not "own" methods.

Here, we describe the classes, methods and functions provided by sp. Instead of manipulating the class slots (components) directly, we provide methods and functions to create the classes from elementary types such as matrices, data.frames or lists and to convert them back to any of these types. Also, coercion (type casting) from one class to the other is provided, where relevant.

Package **sp** is loaded by

```
> library(sp)
```

The spatial data classes implemented are points, grids, lines, rings and polygons. Package **sp** provides classes for the spatial-only information (the topology), e.g. `SpatialPoints`, and extensions for the case where attribute information is stored in either a `data.frame` or an `AttributeList` object is available for each point, e.g. `SpatialPointsDataFrame`. The available data classes are:

data type	class	attributes?	contains
points	`SpatialPoints`	No	`Spatial`*
points	`SpatialPointsDataFrame`	Yes	`SpatialPoints`*
pixels	`SpatialPixels`	No	`SpatialPoints`*
pixels	`SpatialPixelsDataFrame`	Yes	`SpatialPixels`*
			`SpatialPointsDataFrame`**
full grid	`SpatialGrid`	No	`SpatialPixels`*
full grid	`SpatialGridDataFrame`	Yes	`SpatialGrid`*
line	`SLine`	No	`Spatial`*
lines	`SLines`	No	`Spatial`*, `Sline` list
lines	`SpatialLines`	No	`Spatial`*, `SLines` list
lines	`SpatialLinesDataFrame`	Yes	`SpatialLines`*
rings	`Sring`	No	`SLine`*
rings	`Srings`	No	`Spatial`*, `Sring` list
rings	`SpatialRings`	No	`Spatial`*, `Srings` list
rings	`SpatialRingsDataFrame`	Yes	`SpatialRings`*

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

The class `Spatial` never holds actual data, it only provides the information common to all derived classes: the spatial coordinates bounding box and information about the coordinate reference system (geographic projection information).

In the following sections we will show how we can create objects of these classes from scratch or from other classes, and which methods and functions are available for them.

3.1 Manipulating spatial objects

Although entries in spatial objects are in principle accessible through their slot name, e.g. `x@coords` contains the coordinates of an object of class or extending `SpatialPoints`, we strongly encourage users to access the data by using functions and methods, in this case `coordinates(x)` to retrieve the coordinates.

3.1.1 Standard methods

Selecting, retrieving or replacing certain attributes in spatial objects with attributes is done using standard methods

- `[` select "rows" (items) and/or columns in the data attribute table; e.g. `meuse[1:2, "zinc"]` returns a `SpatialPointsDataFrame` with the first two points and an attribute table with only variable "zinc".
- `[[` select a column from the data attribute table
- `[[<-` assign or replace values to a column in the data attribute table.

Other methods available are: `plot`, `summary`, `print`, `dim` and `names` (operate on the data.frame part), `as.data.frame`, `as.matrix` and `image` (for gridded data), `lines` (for line data), `points` (for point data), `subset` (points and grids) and `stack` (point and grid data.frames).

3.1.2 Spatial methods

A number of spatial methods are available for the classes in `sp`:

- `dimensions(x)` returns number of spatial dimensions
- `transform(x, CRS("+proj=latlong +datum=WGS84"))` transform from one coordinate reference system (geographic projection) to another (requires package **spproj** — see below 4.1)
- `bbox(x)` returns the coordinate bounding box
- `coordinates(x)` returns a matrix with the spatial coordinates
- `gridded(x)` tells whether `x` derives from SpatialPixels
- `spplot(x)` plot attributes, possibly in combination with other types of data (points, lines, grids, polygons), and possibly in as a conditioning plot for multiple attributes
- `overlay(x, y)` combine two spatial layers of different type, e.g. retrieve the polygon or grid values on a set of points, or retrieve the points (or a function of their attributes) within (sets of) polygons.
- `spsample(x)` sampling of spatial points in continuous space within a polygon, a gridded area, or on a spatial line. Subsetting and `sample` can be used to subsample full spatial entities.

3.2 Spatial points

3.2.1 Points without attributes

We can generate a set of 10 points on the unit square $[0, 1] \times [0, 1]$ by

```
> set.seed(20050801)
> xc <- round(runif(10), 2)
> yc <- round(runif(10), 2)
> xy <- cbind(xc, yc)
> xy
```

```
        xc   yc
 [1,] 0.72 0.06
 [2,] 0.83 0.39
 [3,] 0.60 0.63
 [4,] 0.28 0.58
 [5,] 0.22 0.30
 [6,] 0.95 0.99
 [7,] 0.41 0.53
 [8,] 0.32 0.50
 [9,] 0.68 0.66
[10,] 0.50 0.99
```

this 10×2 matrix can be converted into a `SpatialPoints` object by

```
> xy.sp <- SpatialPoints(xy)
> xy.sp

SpatialPoints:
        xc   yc
 [1,] 0.72 0.06
 [2,] 0.83 0.39
 [3,] 0.60 0.63
 [4,] 0.28 0.58
 [5,] 0.22 0.30
 [6,] 0.95 0.99
 [7,] 0.41 0.53
 [8,] 0.32 0.50
 [9,] 0.68 0.66
[10,] 0.50 0.99
Coordinate Reference System (CRS) arguments: NA

> plot(xy.sp, pch = 2)
```

The plot is shown in figure 4.

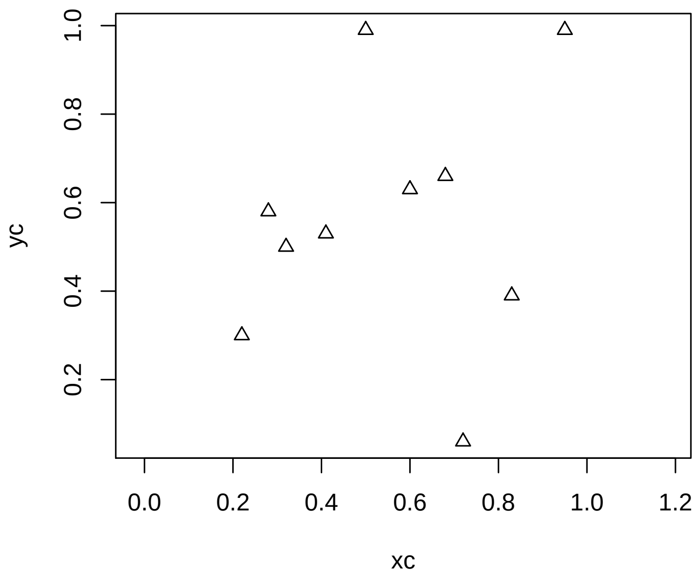

Figure 4: plot of `SpatialPoints` object; aspect ratio of x and y axis units is 1

We can retrieve the coordinates from `xy.sp` by

```
> xy.cc <- coordinates(xy.sp)
> class(xy.cc)

[1] "matrix"

> dim(xy.cc)

[1] 10  2
```

and other methods retrieve the bounding box, the dimensions, select points (not dimensions or columns), coerce to a data.frame, or print a summary.

3.2.2 Points with attributes

One way of creating a `SpatialPointsDataFrame` object is by building it from a a `SpatialPoints` object and a data.frame containing the attributes:

```
> df <- data.frame(z1 = round(5 + rnorm(10), 2), z2 = 20:29)
> df

     z1 z2
1  6.82 20
2  4.85 21
3  3.84 22
4  6.12 23
5  4.31 24
6  3.94 25
7  4.43 26
8  3.91 27
9  5.11 28
10 5.65 29

> xy.spdf <- SpatialPointsDataFrame(xy.sp, df)
> xy.spdf

    coordinates   z1 z2
1  (0.72, 0.06) 6.82 20
2  (0.83, 0.39) 4.85 21
3   (0.6, 0.63) 3.84 22
4  (0.28, 0.58) 6.12 23
5   (0.22, 0.3) 4.31 24
6  (0.95, 0.99) 3.94 25
7  (0.41, 0.53) 4.43 26
8   (0.32, 0.5) 3.91 27
9  (0.68, 0.66) 5.11 28
10  (0.5, 0.99) 5.65 29

> summary(xy.spdf)

Object of class SpatialPointsDataFrame
Coordinates:
    min  max
xc 0.22 0.95
yc 0.06 0.99
Is projected: NA
proj4string : [NA]
Number of points: 10
Data attributes:
       z1              z2
 Min.   :3.840   Min.   :20.00
 1st Qu.:4.032   1st Qu.:22.25
 Median :4.640   Median :24.50
 Mean   :4.898   Mean   :24.50
 3rd Qu.:5.515   3rd Qu.:26.75
 Max.   :6.820   Max.   :29.00
```

```
> dimensions(xy.spdf)

[1] 2

> xy.spdf[1:2, ]

   coordinates   z1 z2
1 (0.72, 0.06) 6.82 20
2 (0.83, 0.39) 4.85 21

> xy.spdf[1]

    coordinates   z1
1  (0.72, 0.06) 6.82
2  (0.83, 0.39) 4.85
3   (0.6, 0.63) 3.84
4  (0.28, 0.58) 6.12
5   (0.22, 0.3) 4.31
6  (0.95, 0.99) 3.94
7  (0.41, 0.53) 4.43
8   (0.32, 0.5) 3.91
9  (0.68, 0.66) 5.11
10  (0.5, 0.99) 5.65

> xy.spdf[1:2, "z2"]

   coordinates z2
1 (0.72, 0.06) 20
2 (0.83, 0.39) 21

> xy.df <- as.data.frame(xy.spdf)
> xy.df[1:2, ]

    z1 z2   xc   yc
1 6.82 20 0.72 0.06
2 4.85 21 0.83 0.39

> xy.cc <- coordinates(xy.spdf)
> class(xy.cc)

[1] "matrix"

> dim(xy.cc)

[1] 10  2
```

A note on selection with `[`: the behaviour is as much as possible copied from that of `data.frame`s, but coordinates are always sticky and allways a `SpatialPointsDataFrame` is returned; `drop=FALSE` is not allowed. If coordinates should be dropped, use the `as.data.frame` method and select the non-coordinate data, or use `[[` to select a single attribute column (example below).

`SpatialPointsDataFrame` objects can be created directly from `data.frames` by specifying which columns contain the coordinates:

```
> df1 <- data.frame(xy, df)
> coordinates(df1) <- c("xc", "yc")
> summary(df1)

Object of class SpatialPointsDataFrame
Coordinates:
    min  max
xc 0.22 0.95
yc 0.06 0.99
Is projected: NA
proj4string : [NA]
Number of points: 10
Data attributes:
       z1              z2
```

```
 Min.   :3.840   Min.   :20.00  
 1st Qu.:4.032   1st Qu.:22.25  
 Median :4.640   Median :24.50  
 Mean   :4.898   Mean   :24.50  
 3rd Qu.:5.515   3rd Qu.:26.75  
 Max.   :6.820   Max.   :29.00  
```

Note that in this form, `coordinates` by setting (specifying) the coordinates promotes its argument, an object of class `data.frame` to an object of class `SpatialPointsDataFrame`. The method `as.data.frame` coerces back to the original `data.frame`. When used on a right-hand side of an equation, `coordinates` *retrieves* the matrix with coordinates:

```
> df2 <- data.frame(xy, df)
> coordinates(df2) <- ~xc + yc
> coordinates(df2)[1:2, ]

       xc   yc
[1,] 0.72 0.06
[2,] 0.83 0.39
```

Elements (columns) in the data.frame part of an object can be manipulated (retrieved, assigned) directly:

```
> df2[["z2"]]

 [1] 20 21 22 23 24 25 26 27 28 29

> df2[["z2"]][10] <- 20
> df2[["z3"]] <- 1:10
> summary(df2)

Object of class SpatialPointsDataFrame
Coordinates:
    min  max
xc 0.22 0.95
yc 0.06 0.99
Is projected: NA 
proj4string : [NA]
Number of points: 10
Data attributes:
       z1              z2              z3       
 Min.   :3.840   Min.   :20.00   Min.   : 1.00  
 1st Qu.:4.032   1st Qu.:21.25   1st Qu.: 3.25  
 Median :4.640   Median :23.50   Median : 5.50  
 Mean   :4.898   Mean   :23.60   Mean   : 5.50  
 3rd Qu.:5.515   3rd Qu.:25.75   3rd Qu.: 7.75  
 Max.   :6.820   Max.   :28.00   Max.   :10.00  
```

Plotting attribute data can be done by using either `spplot` to colour symbols, or `bubble` which uses symbol size:

```
> print(bubble(df2, "z1", key.space = "bottom"), split = c(1, 1, 2, 1), more = TRUE)
> print(spplot(df2, "z1", key.space = "bottom"), split = c(2, 1, 2, 1), more = FALSE)
```

the resulting plots are shown in figure 5.

More is actually going on here than might appear to be the case, because the attribute data are actually being converted to the lightweight `data.frame` object defined in **sp** and named `AttributeList`. This is because `data.frame` objects generate character row names, which is not a big overhead for thousands of points, lines or polygons, but which is a major overhead for hundreds of thousands of points or pixels.

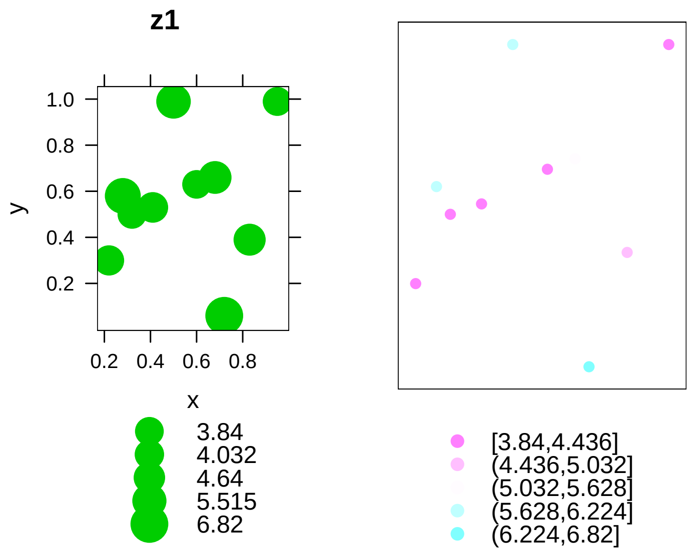

Figure 5: plot of `SpatialPointsDataFrame` object, using symbol size (`bubble`, top) or colour (`spplot`, bottom)

3.3 Grids

Package `sp` has two classes for grid topology: `SpatialPixels` and `SpatialGrid`. The pixels form stores coordinates and is for partial grids, or unordered points; the `SpatialGrid` form does not store coordinates but holds full grids (i.e., `SpatialGridDataFrame` holds attribute values for each grid cell). Objects can be coerced from one representation to the other. The metadata for the grid are held in a `GridTopology` object.

3.3.1 Creating grids from topology

When we know the offset, the cell sizes and the dimensions of a grid, we can specify this by using the function `GridTopology`:

```
> gt <- GridTopology(cellcentre.offset = c(1, 1, 2), cellsize = c(1, 1, 1), cells.dim = c(3,
+     4, 6))
> grd <- SpatialGrid(gt)
> summary(grd)

Object of class SpatialGrid
Coordinates:
          min max
coords.x1   1   3
coords.x2   1   4
coords.x3   2   7
Is projected: NA
proj4string : [NA]
Number of points: 2
Grid attributes:
  cellcentre.offset cellsize cells.dim
1                 1        1         3
2                 1        1         4
3                 2        1         6
```

3.3.2 Creating grids from points

In the following example a three-dimensional grid is constructed from a set of point coordinates:

```
> pts <- expand.grid(x = 1:3, y = 1:4, z = 2:7)
> grd.pts <- SpatialPixels(SpatialPoints(pts))
> summary(grd.pts)

Object of class SpatialPixels
Coordinates:
  min max
x   1   3
y   1   4
z   2   7
Is projected: NA
proj4string : [NA]
Number of points: 72

> grd <- as(grd.pts, "SpatialGrid")
> summary(grd)

Object of class SpatialGrid
Coordinates:
  min max
x   1   3
y   1   4
z   2   7
Is projected: NA
proj4string : [NA]
Number of points: 2
Grid attributes:
  cellcentre.offset cellsize cells.dim
x                 1        1         3
y                 1        1         4
z                 2        1         6
```

Note that when passed a points argument, SpatialPixels accepts a tolerance (default 10 * .Machine$double.eps) to specify how close the points have to be to being exactly on a grid. For very large coordinates, this value may have to be increased. A warning is issued if full rows and/or columns are missing.

3.3.3 Gridded data with attributes

Spatial, gridded data are data with coordinates on a regular lattice. To form such a grid we can go from coordinates:

```
> attr = expand.grid(xc = 1:3, yc = 1:3)
> grd.attr = data.frame(attr, z1 = 1:9, z2 = 9:1)
> coordinates(grd.attr) = ~xc + yc
> gridded(grd.attr)

[1] FALSE

> gridded(grd.attr) = TRUE
> gridded(grd.attr)

[1] TRUE

> summary(grd.attr)

Object of class SpatialPixelsDataFrame
Coordinates:
   min max
xc   1   3
yc   1   3
Is projected: NA
proj4string : [NA]
```

```
Number of points: 9
Data attributes:
       z1          z2
 Min.   :1   Min.   :1
 1st Qu.:3   1st Qu.:3
 Median :5   Median :5
 Mean   :5   Mean   :5
 3rd Qu.:7   3rd Qu.:7
 Max.   :9   Max.   :9
```

3.3.4 Are grids stored as points or as matrix/array?

The form in which gridded data comes depends on whether the grid was created from a set of points or from a matrix or external grid format (e.g. read through **rgdal**, see Section 4.2). Retrieving the form, or conversion to another can be done by `as(x, "Class")`, or by using the function `fullgrid`.

The advantage of having grids in cell form is that when a large part of the grid contains missing values, these cells do not have to be stored; also, no ordering of grid cells is required. For plotting a grid with `levelplot`, this form is required and `spplot` (for grids a front-end to `levelplot`) will convert grids that are not in this form. In contrast, `image` requires a slightly altered version of the the full grid form. A disadvantage of the cell form is that the coordinates for each point have to be stored, which may be prohibitive for large grids. Grids in cell form do have an index to allow for fast transformation to the full grid form.

Besides `print`, `summary`, `plot`, objects of class `SpatialGridDataFrame` have methods for

- `[` select rows (points) or columns (variables)
- `[[` retrieve a column from the attribute table (data.frame part)
- `[[<-` assign or replace a column in the attribute table (data.frame part)
- `coordinates` retrieve the coordinates of grid cells
- `as.matrix` retrieve the data as a matrix. The first index (rows) is the x-column, the second index (columns) the y-coordinate. Row index 1 is the smallest x-coordinate; column index 1 is the larges y-coordinate (top-to-bottom).
- `as` coercion methods for `data.frame`, `SpatialPointsDataFrame`
- `image` plot an image of the grid

Finally, `spplot`, a front-end to `levelplot` allows the plotting of a single grid plot or a lattice of grid plots.

3.3.5 Row and column selection of a region

Rows/columns selection can be done when gridded data is in the full grid form (as `SpatialGrid-DataFrame`). In this form also rows and/or columns can be de-selected (in which case a warning is issued):

```
> fullgrid(grd.attr) = FALSE
> grd.attr[1:5, "z1"]

Object of class SpatialPixelsDataFrame
Object of class SpatialPixels
Grid topology:
   cellcentre.offset cellsize cells.dim
xc                 1        1         3
yc                 1        1         2
SpatialPoints:
     xc yc
[1,]  1  1
[2,]  2  1
[3,]  3  1
[4,]  1  2
[5,]  2  2
Coordinate Reference System (CRS) arguments: NA

Data:
       z1
 Min.   :1
 1st Qu.:2
 Median :3
 Mean   :3
 3rd Qu.:4
 Max.   :5

> fullgrid(grd.attr) = TRUE
> grd.attr[1:2, -2, c("z2", "z1")]

SpatialPoints:
     xc yc
[1,]  1  2
[2,]  3  3
Coordinate Reference System (CRS) arguments: NA
```

3.4 Lines

3.4.1 Building line objects from scratch

In many instances, line coordinates will be retrieved from external sources. The following example shows how to build an object of class `SpatialLines` from scratch. Note that the `Slines` objects have to be given character ID values, and that these values must be unique for `Slines` objects combined in a `SpatialLines` object.

```
> l1 <- cbind(c(1, 2, 3), c(3, 2, 2))
> l1a <- cbind(l1[, 1] + 0.05, l1[, 2] + 0.05)
> l2 <- cbind(c(1, 2, 3), c(1, 1.5, 1))
> Sl1 <- Sline(l1)
> Sl1a <- Sline(l1a)
> Sl2 <- Sline(l2)
> S1 <- Slines(list(Sl1, Sl1a), ID = "a")
> S2 <- Slines(list(Sl2), ID = "b")
> Sl <- SpatialLines(list(S1, S2))
> summary(Sl)

Object of class SpatialLines
Coordinates:
   min  max
r1   1 3.05
r2   1 3.05
Is projected: NA
proj4string : [NA]
```

3.4.2 Building line objects with attributes

The class `SpatialLinesDataFrame` is designed for holding lines data that have an attribute table (data.frame) attached to it:

```
> df <- data.frame(z = c(1, 2), row.names = getSLSlinesIDSlots(Sl))
> Sldf <- SpatialLinesDataFrame(Sl, data = df)
> summary(Sldf)
```

```
Object of class SpatialLinesDataFrame
Coordinates:
   min  max
r1   1 3.05
r2   1 3.05
Is projected: NA
proj4string : [NA]
Data attributes:
       z
 Min.   :1.00
 1st Qu.:1.25
 Median :1.50
 Mean   :1.50
 3rd Qu.:1.75
 Max.   :2.00
```

Not many useful methods for lines are available yet. The `plot` method only plots the lines, ignoring attribute table values. Suggestions for useful methods are welcome, although most likely these classes need to be extended to handle more complex phenomena, such as cruises or animal radio tag data, which may actually be better represented as ordered points, because each point will at least have a timestamp.

3.5 Polygons

3.5.1 Building from scratch

The following example shows how a set of polygons are built from scratch. Note that `Sr4` has the opposite direction (anti-clockwise) as the other three (clockwise); it is meant to represent a hole in the `Sr3` polygon. The default value for the hole colour `pbg` is `"transparent`, which will not show, but which often does not matter, because another polygon fills the hole — here it is set to `"white"`. Note that the `Srings` objects have to be given character ID values, and that these values must be unique for `Srings` objects combined in a `SpatialRings` object.

```
> Sr1 <- Sring(cbind(c(2, 4, 4, 1, 2), c(2, 3, 5, 4, 2)))
> Sr2 <- Sring(cbind(c(5, 4, 2, 5), c(2, 3, 2, 2)))
> Sr3 <- Sring(cbind(c(4, 4, 5, 10, 4), c(5, 3, 2, 5, 5)))
> Sr4 <- Sring(cbind(c(5, 6, 6, 5, 5), c(4, 4, 3, 3, 4)), hole = TRUE)
> Srs1 <- Srings(list(Sr1), "s1")
> Srs2 <- Srings(list(Sr2), "s2")
> Srs3 <- Srings(list(Sr3, Sr4), "s3/4")
> SR <- SpatialRings(list(Srs1, Srs2, Srs3), 1:3)

> plot.SpatialRings(SR, col = 1:3, pbg = "white")
```

The plot is shown in figure 6.

3.5.2 Polygons with attributes

Polygons with attributes, objects of class `SpatialRingsDataFrame`, are built from the `SpatialRings` object (topology) and the attributes (data.frame). The `row.names` of the attributes data frame are matched with the ID slots of the `SpatialRings` object, and the rows of the data frame will be re-ordered if necessary.

```
> attr <- data.frame(a = 1:3, b = 3:1, row.names = c("s3/4", "s2", "s1"))
> SrDf <- SpatialRingsDataFrame(SR, attr)
> as(SrDf, "data.frame")

     a b
s1   3 1
s2   2 2
s3/4 1 3
```

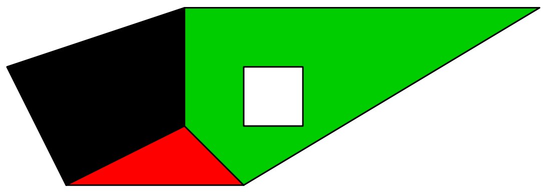

Figure 6: Constructed polygon rings with an internal hole

```
> print(spplot(SrDf, at = seq(0.5, 3.5, 1), col.regions = rainbow(3)))
```

The plot is shown in figure 7.

or, as another way to create the `SpatialRingsDataFrame` object:

```
> SrDf = attr
> rings(SrDf) = SR
```

4 Interfaces to GIS and external formats

For an interim period, it seems sensible to try to write wrappers for existing packages interfacing spatial data and R, using the new **sp** classes. This allows the interface packages to continue with their regular codebase, and existing users of these packages to be able to choose when or if they will migrate to **sp** classes. It also permits us to develop and refine the **sp** package without dependence on interface packages. The wrapper packages will be released as R source packages and R Windows binary packages on SourceForge using the standard R repository mechanism. This section will try to show how to use several of these wrapper packages in conjunction with **sp**:

- **spproj**: Interface to PROJ.4 cartographic projections library
- **spgpc**: Polygon clipping for Spatial Rings using the gpclib package
- **spGDAL**: Interface the Geospatial Data Abstraction Library using the **rgdal** package
- **spmaptools**: Interface to the shapelib library for reading and writing shapefiles, using the **maptools** package
- **spgrass6**: New version of the R/GRASS interface for GRASS 6

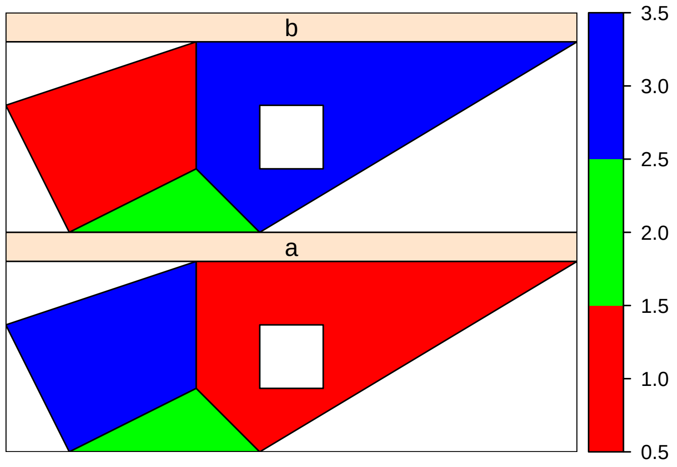

Figure 7: Trellis (lattice) graphics plots of the two attributes of the set of polygons

Work is progressing on a package to write polygon files for WinBUGS, with code on SourceForge already. Extended examples of the use of the wrapper packages is given on the SourceForge website. In addition, the **gstat** package is almost **sp**-ready, and other CRAN packages are following.

The first two examples will also show how data can be exchanged with file formats typically used with ArcGIS: shapefiles and ASCII grids.

4.1 Transforming coordinate reference systems

It is often the case that input data is not in the required coordinate reference system (CRS). If the coordinate reference system can be documented, the projection argument slot in objects inheriting from the Spatial class may assigned the values used in the PROJ.4 library. Then a suitable transform method may be used to change to a new coordinate reference system. Methods are available for points, lines and rings (polygons).

This example uses a shapefile distributed with the **maptools** package, giving the centre-lines of major rivers in Norway in the UTM zone 33 coordinate reference system. If we would like to use these with a coastline, for example taken from the high resolution database (based on WDB II) in the **mapdata** package (with geographical not projected coordinates), we will need to transform to match (see Fig 8):

```
> library(maptools)
> library(mapdata)
> library(spproj)

> lns <- read.shape(system.file("shapes/fylk-val.shp", package = "maptools")[1])

Shapefile type: PolyLine, (3), # of Shapes: 97

> rivers_utm <- shp2SLDF(lns, proj4string = CRS("+proj=utm +zone=33 +datum=WGS84"))
> nor_coast <- map("worldHires", "norway", fill = TRUE, col = "transparent", plot = FALSE, ylim = c(50,
+     73))
> ncSR_ll <- as.SpatialRings.map(nor_coast, IDs = nor_coast$names, proj4string = CRS("+proj=latlong +datum=WGS84"))
> ncSR_utm <- transform(ncSR_ll, CRS("+proj=utm +zone=33 +datum=WGS84"))
> plot(ncSR_utm)
> plot(rivers_utm, add = TRUE, col = "blue")
> axis(1)
> axis(2)
```

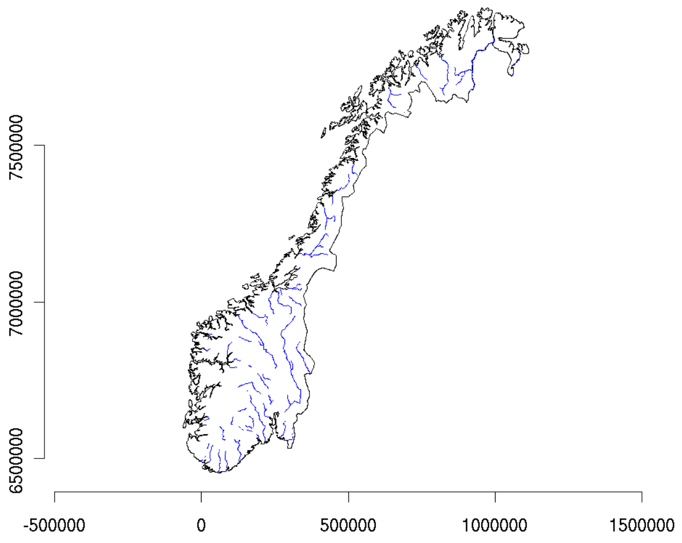

Figure 8: Norway rivers UTM zone 33

The gaps in the rivers are where lakes should be; they are not included in either of the data sources, but their absence should not suggest that Norway is an arid country! The equivalent output for geographical coordinates can also be obtained (see Fig 8):

```
> rivers_ll <- transform(rivers_utm, CRS("+proj=latlong +datum=WGS84"))
```

```
> plot(ncSR_ll)
> plot(rivers_ll, add = TRUE, col = "blue")
> axis(1)
> axis(2)
```

4.2 Reading grids through GDAL

GDAL is an important resource widely used for reading (and writing) many raster data formats. The **rgdal** package (on CRAN) was originally written by Tim Keitt, and provides bindings for GDAL in R. Because GDAL, like PROJ.4, is external software, the R source packages can only be built and installed when the external software is available. Here, the functions in **rgdal** are used to read data into a class provided in the **sp** package (data taken from http://geodata.grid.unep.ch/, 2.5 minute world population density 1990); the plot is shown in figure 10:

```
> library(spGDAL)
```

```
> popden_Nor <- read.GDAL("popd_90.txt", band = 1, offset = c(320, 4400), region.dim = c(350,
+     700))
```

```
popd_90.txt has GDAL driver AAIGrid
and has 3432 rows and 8640 columns
Closing GDAL dataset handle 0xaf971f0...  destroyed ... done.
```

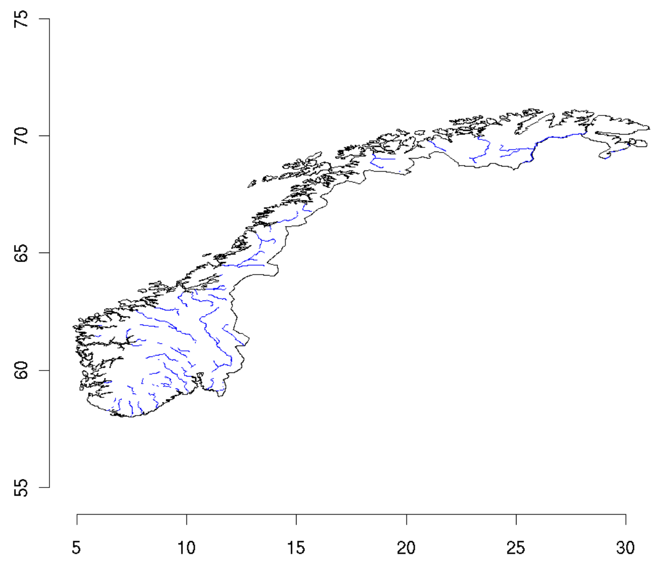

Figure 9: Norway rivers geographical coordinates

```
> proj4string(popden_Nor) <- CRS("+proj=latlong +datum=WGS84")
> summary(popden_Nor)

Closing GDAL dataset handle (nil)... done.
Object of class SpatialGridDataFrame
Coordinates:
        min      max
x  3.354167 32.47917
y 57.104167 71.64583
Is projected: FALSE
proj4string : [+proj=latlong +datum=WGS84]
Number of points: 2
Grid attributes:
  cellcentre.offset   cellsize cells.dim
x          3.354167 0.04166667       700
y         57.104167 0.04166667       350
Data attributes:
     band1
 Min.   :    0.00
 1st Qu.:    1.00
 Median :    4.00
 Mean   :   19.69
 3rd Qu.:   12.00
 Max.   :12878.00
 NA's   :96683.00

> brks <- unique(quantile(popden_Nor@data[[1]], seq(0, 1, 1/10), na.rm = TRUE))
> cols <- grey((1:(length(brks) - 1)/length(brks)))

> image(popden_Nor, col = rev(cols), breaks = brks)
> legend(c(5, 30), c(53, 57), legend = leglabs(brks), fill = rev(cols), ncol = 3, bty = "n",
+     cex = 0.7, y.intersp = 0.7)
```

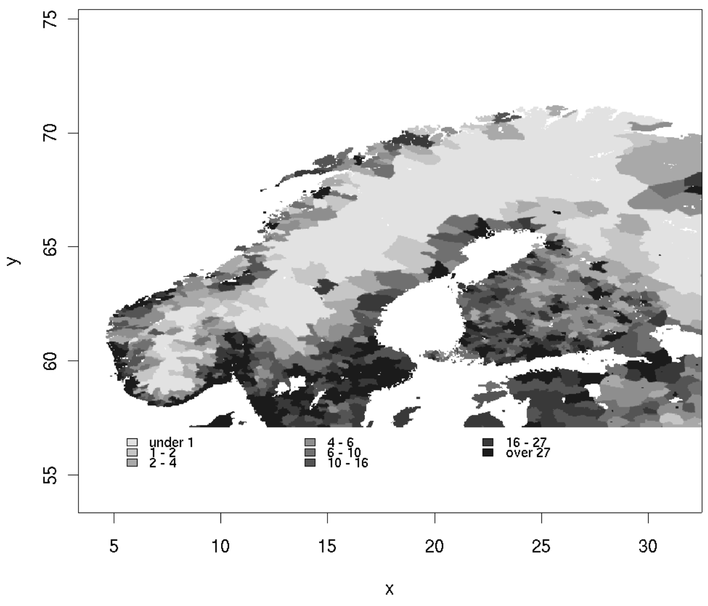

Figure 10: Population density 1990, persons per square kilometer, Norway and neighbours

4.3 Exchanging data with GRASS 6

The **GRASS** package provided close-coupled integration between GRASS 5 and R for sites and raster data. As machine capacity has increased, it seems more robust to connect R and GRASS 6 without local copies of GRASS libraries. The work that has been done so far shows that links for raster data through temporary files work adequately, as does the use of the GRASS plugin in GDAL through **rgdal** and **spGDAL**, at least for reading GRASS data into R. Writing from R to GRASS is presently done by Arc ASCII grid temporary files. There are still a number of problems to resolve, such as the transparent movement of category labels, and determining which window and resultion should apply. The **GRASS** package always used the current region values from GRASS, while GDAL reads the database directly without cropping or resampling to the current region.

Vector integration was missing in the **GRASS** package, but is now available in both directions using **maptools** for shapefiles, and OGR formats will become available before long. In addition, the vector topology engine in GRASS 6 makes it possible to consider handing off the questions of line generalisation and polygon neighbour finding to GRASS. Examples are given in a note in the newest number of GRASS News.

5 Work in Progress

The work reported here is in progress, and while the **sp** classes have mostly found their forms, changes in internal structures may be expected. The wrapper packages are available from within an online R session using the instructions on the download page, and help in debugging and advancing these wrappers is of great value. Since **sp** was released to CRAN in late April 2005, user comments and suggestions have already led to marked improvements in speed and functionality. By the summer of 2006, we expect to see a range of data import and data analysis packages using **sp** classes directly, and relying on **sp** for basic computational geometry and visualisation functionalities.

This draft paper is also "work in progress", as the lack of adequate references shows. Because of its tight links to the software, and the speed of change in the code, we hope that the reader will bear with us in successive drafts as we move towards a more polished version. Our work is based on the insight that, given open source, a major and perhaps the decisive input to scientific progress is the coming into being and development of mutually contributing communities, in which simple wishes or bug reports can trigger major re-coding, and lead the authors to understand better the partiality of their awareness of how spatial data analysis is done in widely varying disciplines.

References

Chambers, J.M., 1998, Programming with data, a guide to the S language. Springer, New York.