Analysing Spatial Data in R: Accessing spatial data

Roger Bivand

Department of Economics
Norwegian School of Economics and Business Administration
Bergen, Norway

31 August 2007

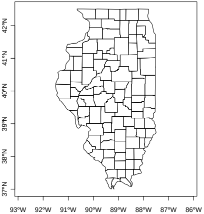
Introduction

- Having described how spatial data may be represented in R, and how to vizualise 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
- **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

Using **maps** data: Illinois counties



There are number of valuable geographical databases in map format that can be accessed directly — beware of IDs!

```
> 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

- 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

Here: neither here nor there

```
> library(rgdal)

> ED50 <- CRS(paste("+init=epsg:4230",
+     "+towgs84=-87,-96,-120,0,0,0,0"))
> IJ.east <- as(char2dms("4d31'00\"E"),
+     "numeric")
> IJ.north <- as(char2dms("52d28'00\"N"),
+     "numeric")
> IJ.ED50 <- SpatialPoints(cbind(x = IJ.east,
+     y = IJ.north), ED50)
> res <- spTransform(IJ.ED50,
+     CRS("+proj=longlat +datum=WGS84"))
> spDistsN1(coordinates(IJ.ED50),
+     coordinates(res), longlat = TRUE) *
+     1000

[1] 124.0994
```

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.

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

Meuse bank CRS

Let's have a look at the Meuse bank CRS — Grids & Datums gives some hints in February 2003 to search for Amersfoort in EPSG:

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

      code                  note
206   4289          # Amersfoort
2497 28991 # Amersfoort / RD Old
2498 28992 # Amersfoort / RD New

> 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=0.999908 +x_0=155000 +y_0=463000 +ellps=bessel
+towgs84=565.237,50.0087,465.658,-0.406857,0.350733,-1.87035,4.0812 +units=m +no_defs

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

PROJCS["Amersfoort / RD New", GEOGCS["Amersfoort", DATUM["D_Amersfoort",
SPHEROID["Bessel_1841", 6377397.155, 299.1528128]], PRIMEM["Greenwich", 0],
UNIT["Degree", 0.017453292519943295]], PROJECTION["Oblique_Stereographic"],
PARAMETER["latitude_of_origin", 52.15616055555555], PARAMETER["central_meridian",
5.38763888888889], PARAMETER["scale_factor", 0.9999079], PARAMETER["false_easting",
155000], PARAMETER["false_northing", 463000], UNIT["Meter", 1]]
```

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 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 vectors

- GIS vector data are points, lines, polygons, and fit the equivalent **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

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
- 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

Reading shapefiles: maptools

```
> 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 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
- Using arguments to `readGDAL`, subregions or bands may be selected, which helps handle large rasters

Reading vectors: rgdal

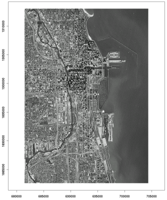
```
> scot1 <- readOGR(dsn = ".",
+     layer = "scot_BNG")

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=0.999601 +x_0=400000
+y_0=-100000 +ellps=airy
+units=m +no_defs
```

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.

Reading rasters: rgdal

```
> getGDALDriverNames()$name

 [1] AAIGrid  AIG      AirSAR   BMP      BSB      BT       CEOS     CPG      DIPEx    DOQ1
[11] DOQ2     DTED     EHdr     ELAS     ENVI     ESAT     FAST     FIT      FujiBAS  GIF
[21] GMT      GSC      GTiff    GXF      HFA      IDA      ILWIS    ISIS2    JDEM     JPEG
[31] L1B      LAN      Leveller MEM      MFF      MFF2     NDF      netCDF   NITF     PAux
[41] PCIDSK   PCRaster PDS      PNG      PNM      RIK      RMF      RS2      RST      SAR_CEOS
[51] SDTS     SGI      USGSDEM  VRT      WCS      XPM
56 Levels: AAIGrid AIG AirSAR BMP BSB BT CEOS CPG DIPEx DOQ1 DOQ2 DTED EHdr ELAS ENVI ESAT ... XPM

> list.files()

[1] "SP27GTIF.TIF"

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

SP27GTIF.TIF has GDAL driver GTiff
and has 929 rows and 699 columns
```

Reading rasters: rgdal



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

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

Reading rasters: rgdal

```
> 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=0.999975
+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:
     band1
 Min.   :  4.0
 1st Qu.: 78.0
 Median :104.0
 Mean   :115.1
 3rd Qu.:152.0
 Max.   :255.0
```

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
- 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
- Use has been made of COM and Python interfaces to ArcGIS; typical use is by loose coupling except in highly customised work situations