Spatial data in R

A. J. Rominger

02 May 2017

Without finding something I really liked for a quick crash course, I put this together

Spatial vector data I/O

There are multiple methods, from several packages, to read in vector data (points, lines, polygons) but the one I find most consistent is `rgdal`. It has simple and smart read and write functions

```
library(sp)
library(rgdal)
```

```
## rgdal: version: 1.1-10, (SVN revision 622)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 1.11.4, released 2016/01/25
##  Path to GDAL shared files: /Library/Frameworks/R.framework/Versions/3.3/Resources/library/rgdal/gda
##  Loaded PROJ.4 runtime: Rel. 4.9.1, 04 March 2015, [PJ_VERSION: 491]
##  Path to PROJ.4 shared files: /Library/Frameworks/R.framework/Versions/3.3/Resources/library/rgdal/p
##  Linking to sp version: 1.2-3
```

```
# assume we're using files from the geodata repo
setwd('~/Dropbox/hawaiiDimensions/geodata/env_data/geol')
geol <- readOGR(dsn = '.', layer = 'hawaii_state_geol_ageClean')
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "hawaii_state_geol_ageClean"
## with 9862 features
## It has 18 fields
```

Note that `readOGR` can be particular about the directories so the safest bet is to set the working directory to the one housing the data you want to read in. This is not completely unexpected, there are multiple files that together make up spatial vector data—database files (`.dbf`), projection files (`.prj`), coordinate files (`.shp`, `.shx`)—so specifying which directory to find them all is helpful. The `layer` arguement gives the common name of all those files, in the above case `hawaii_state_geol_ageClean`.

Before we go into the class and properties of the object we read in with `readOGR`, let's quickly demonstrate how we write out a spatial data object, the key thing is that we have to specify what kind of file(s) to write out. We specify that with the `driver` which can take several values. We'll start with the same driver that we read in, the ESRI shape file

```
# I tend to always specify overwrite_layer = TRUE, otherwise changes that I have
# made to the object I'm writing out won't be saved to the machine
writeOGR(geol, dsn = '.', layer = 'testWriteOut',
         driver = 'ESRI Shapefile', overwrite_layer = TRUE)
```

After confirming that the file was made (go check in your folder system!), let's make sure to delete it so we don't have stray files hanging around

```
file.remove(list.files(pattern = 'testWriteOut'))
```

```
## [1] TRUE TRUE TRUE TRUE
```

We can write out spatial data to many formats, which is really useful for collaborators who don't use `R`, for example we can write something out that they can view in Google Earth by specifying `driver = 'KML'`. However, to do that we'll first need to manipulate the object `geol`, so we'll save KML for the next section.

But in general we can see what drivers are availible like this:

```r
drvrs <- ogrDrivers()
nrow(drvrs) # there's a lot of options
```

```
## [1] 54
```

```r
head(drvrs)
```

```
##         name write
## 1 AeronavFAA FALSE
## 2     ARCGEN FALSE
## 3     AVCBin FALSE
## 4     AVCE00 FALSE
## 5        BNA  TRUE
## 6    CartoDB FALSE
```

Manipulating spatial vector data

Now that we've read some data in, let's learn a few things about it. First off its class and the properties of that class:

```r
class(geol)
```

```
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
```

We can check out what that means in the lengthy help docs of package `sp`:

```r
?`SpatialPolygonsDataFrame-class`
```

You'll see right off this thing has something called `slots`; this indicates we have a class belonging to the more advanced object-oriented system in `R` (known as *S4*). For our purposes we don't need to worry about that too much (although for more information see here). We can effectively treat these *S4* objects like lists, except we access there "elements" (again known as `slots`) with the at sign `@` not the dollar sign `$`. The most oject accessed slot is the data slot, where all the information lives about all the different polygons that we just read in:

```r
class(geol@data)
```

```
## [1] "data.frame"
```

```r
names(geol@data)
```

```
##  [1] "ID"         "ISLAND"     "VOLCANO"    "STRAT_CODE" "SYMBOL"
##  [6] "AGE_GROUP"  "AGE_RANGE"  "NAME"       "NAME0"      "UNIT"
## [11] "ROCK_TYPE"  "LITHOLOGY"  "VOLC_STAGE" "COMPOSITIO" "SOURCE"
## [16] "age_min"    "age_max"    "age_mid"
```

We can see this is just a `data.frame` with useful columns like `age_mid` which is the median estimated age of a particular lava flow.

Trying to work with the other `slots` can be tedious, in particular the `polygons` slot is a list of `Polygons` objects, which in turn contains another list, this time of `Polygon` objects, which finally is where we can find the useful information (like coordinates) about the spatial properties of the data we're working with. There are times when you need to work within this hierrarchical mess, stackoverflow is your friend in those cases. The crazy structure of these objects means that they have impressive performance, so we won't fault the authors their craziness. For example, we can treat our spatial object like a `data.frame` using familiar functions like `[` and `rbind`.

There are also functions that save us the hassel of dealing with the hierrarchical structure of `sp`'s classes. For example we can quickly extract all the coordinates that compose the polygons with the appropriately named function:

```
xy <- coordinates(geol)
nrow(xy) # there's a lot
```

```
## [1] 9862
```

This may be most useful when we've read in spatial points—in which case each coordinate corresonds to spatial point—instead of polygons as in the current case, becuase many points make up one polygon.

Other useful functions that are worth getting familiar with correspond to the common tasks of most GIS practisioners: joining spatial objects (`gUnion` family of functions from package `rgeos`) and overlaying objects to test is they overlap (`over` in package `sp`). Look at their help docs for more information on usage; pay close attention to the class of object expected as inputs into those functions—sadly it's not always our handy `SpatialPolygonsDataFrame`.

Coordinate Reference System (CRS)

A super important property of any spatial data is what *Coordinate Reference System* it is defined with. In order to compre spatial objects, or plot them all together, they need to be in the same CRS. You don't need to know much about CRS except that its basically the way of keeping track of how positions on the Earth were measured (units, official datums) and whether those positions were projected from the spheroid to a plane. We can find out an objects CRS with the function

```
proj4string(geol) # it aint pretty
```

```
## [1] "+proj=utm +zone=4 +datum=NAD83 +units=m +no_defs +ellps=GRS80 +towgs84=0,0,0"
```

Importantly we should note that we're using a UTM-based projection. If we want the coordinates as latitude and longitude (which we need, for example, to write out to a Google Earth readable file) we need to re-project this object:

```
p4s <- '+proj=longlat +ellps=WGS84 +towgs84=0,0,0,0,0,0,0 +no_defs'
geolLatLon <- spTransform(geol, CRS(p4s))
```

`p4s` holds the `PROJ.4` string, which is the standard bookkeeping format for CRS information. I just happened to know that one off the top of my head, but for more information check out this source.

For the sake of completeness we can know write out our latitude-longitude transformed object to something we can open in Google Earth:

```
writeOGR(geolLatLon, dsn = 'testKML.kml', layer = 'testKML', driver = 'KML',
         overwrite_layer = TRUE)
```

Check for yourself that it's there and then delete the file

```
file.remove('testKML.kml')
```

```
## [1] TRUE
```

Raster data I/O

The `raster` package has really made our lives a lot easier when it comes to rasters! Let's begin with simple input output

```
library(raster)
setwd('~/Dropbox/hawaiiDimensions/geodata/env_data/precip/StateRFGrids_mm2')
precip <- raster('staterf_mmann/w001001.adf') # read-in non-standard format
writeRaster(precip, 'testRaster', overwrite = TRUE) # write to standard format
KML(precip, 'testRaster.kml', overwrite = TRUE) # write to KML
```

That's all there is to it! Again, convince yourself that they're there and then remove

```
setwd('~/Dropbox/hawaiiDimensions/geodata/env_data/precip/StateRFGrids_mm2')
file.remove(list.files(pattern = 'testRaster'))
```

```
## [1] TRUE TRUE TRUE
```

Working with raster data

I will let the raster help docs do most of the explaining here, see `help('raster-package')`. Pay most attention to sections II–V and VII. Take a quick look at `stack` and `brick` from section I to think more about how rasters are structured—and can be structured in multi-dimensional ways.

One key point I want to leave you with is that its often much easier to re-project a vector object than a raster, so when dealing with multiple objects, including rasters, try to re-project all your vector data to match the rasters, not the other way around. With our two objects (`geol` and `precip`) we could do something like this:

```
geol <- spTransform(geol, CRS(proj4string(precip)))
```

Notice how clean that is because we don't have to directly mess with the `PROJ.4` strings, we can just extract the `PROJ.4` string of `precip` and use it to transform `geol`. Pretty cool.