Wrangling spatial data in R

One hindrance to wrangling spatial data in R is that there is no unified spatial toolkit. Several packages are often needed, that may overlap or clash. Spatial data work is not R's comparative advantage.

But it allows for a completely integrated workflow. If one's project focus is not spatial analysis (if it is, use a GIS software) but requires wrangling some spatial data for subsequent statistical analysis, one might be better off doing it in R.

This document contains some basic information and tools to deal with spatial data in R, notably: the main attributes of spatial data, various formats, key R packages, more specialized packages, and useful chains of commands (e.g., how to unproject projected data).

Contents

1 R objects

Everything in R is an object. An object has many features, two of particular importance are: the basic **type** of its data (low-level), and how these data are combined or **structured** (high-level).

object's feature	R function	most common options	
data type (low-level)	`typeof()`	`character`	"a", "swc" ...
		`double`	2, 15.5 ...
		`integer`	2L ...
		`logical`	TRUE, FALSE
		`complex`	1+4i ...
		`closure`	functions
		`S4`	some S4 objects
data structure (high-level)		atomic vector	1D, homogeneous (contents must be of the same type)
		list	1D, heterogeneous (contents can be of different types)
		matrix	2D, homogeneous
		data frame	2D, heterogeneous. It is a list of equal-length vectors.
		array	nD, homogeneous
		factor	a vector that can contain only predefined values.

2 Characteristics of spatial data

2.1 Type: vector vs raster

Spatial data[1] can be categorized into two **types**, based on how they express geographical features and store their information: vectors and rasters. For each type, multiple data **formats** have been developed: shapefiles, netCDFs...

R has packages to handle virtually any type×format of spatial data, i.e., they provide functions to load the data, wrangle them into desirable formats, and analyze them.

Data type	Vector Vectors consider the features as geometrical shapes: points, lines, polygons; using x,y coordinates to define their location. Vectors are useful to define centers or boundaries of features.	Raster Rasters store information of features in the form of a grid, i.e., rows and columns of cells, with each cell storing a single value. Rasters are useful to describe interiors rather than boundaries.
Data format & file extension	• GeoJSON • TIGER • ... Some split the data across multiple files: • shapefile: .shp, .shx, .dbf, .prj • MapInfo: .MIF, .MID	• JPEG • ESRI grid • netCDF: .nc • GRIB • ...

2.2 Spatial attributes

Spatial data have three *spatial* attributes: the coordinate reference system, the extent, and the resolution. One should find them in the metadata.

The Coordinate Reference System (CRS)

Spatial data represent features on the Earth's 3-D surface. A mathematical model of the shape of the Earth has been employed so as to express and store locations on the Earth as pairs of coordinates.

- Some mathematical formulas translate or 'project' the 3-D surface to a 2-D surface.
  $\Longrightarrow$ The data use {x,y} coordinates, distance is expressed in 2-D measures (e.g., meters).
- Others refer to the Earth as a 3-D object.
  $\Longrightarrow$ The data use {lat,lon} coordinates, distance is expressed in decimal degrees.

[1] Often, the term 'geospatial' is used instead of 'spatial' data. 'Geo' refers to 'geography' (the study of the surface of the earth), indicating that the data have a locational component, they refer to locations on the Earth. 'Geographic' is the right word for graphic presentation (e.g., maps) of features and phenomena on or near the Earth's surface. Geographic data are a significant subset of spatial data, although the terms geographic, spatial, and geospatial are often used interchangeably. *Source:* `http://basudebbhatta.blogspot.com/2010/02/spatial-and-geospatial.html`

All vector and raster spatial data are 3-D data that have been transformed so that they can be stored with pair of coordinates. The data's **coordinate reference system (CRS)** describes how.[2]

A CRS is either

- **geographic or 'unprojected'**: locations are defined with **{lon,lat} coordinates**.
  The axes' units are decimal degrees, which do not correspond to uniform distances on the Earth's surface. Thus, geographic CRS are not ideal for measuring distances.
  ex: `+proj=longlat +ellps=WGS84 +datum=WGS84 +towgs84=0,0,0`
- **projected**: locations are defined with **{x,y} coordinates**.
  And lat and lon are themselves functions of the coordinates. Various projections exist (*UTM, Mercator, Lambert Conformal Conic...*), units can be meters...
  ex: `+proj=utm +zone=18 +datum=WGS84 +units=m +ellps=WGS84 +towgs84=0,0,0`

To conduct any spatial analysis using several layers, we must first put them in the same CRS.[3]

[2] A CRS is composed of several elements: the coordinate system (`proj`), the ellipsoid, i.e., how the earth's roundness is calculated (`ellps`), the origin point 0,0 of the coordinate axes (`datum`), the units of the axes (`units`).

[3] One can define a CRS interchangeably using a lengthy "proj4string", like in the example, or a concise "EPSG" numeric code. It is also possible to use a EPSG code in a proj4string definition with `+init=epsg:<EPSG>`. For example, setting `+init=epsg:4326` will make the PROJ library automatically add the rest of the parameters and convert them into `+init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0`.

3 R packages

3.1 Main packages: `[sp::]`, `[sf::]`, `[raster::]`

Storage type	**Vector**		**Raster**
Package	`[sp::]`	`[sf::]`	`[raster::]`
Object class	object is `Spatial*` (`Point`, `Line`, `Polygon`), or `Spatial*DataFrame` if has attribute data	object is stored as a data frame with a 'geometry' list column that contains the geographic information	object is stored as a `RasterLayer` or `RasterBrick`
Functions			
load into R		`st_read()`	– `raster()` for single-layers – `brick()` for multi-bands – then `getValues()` to force R to import the values
unprojected?	`is.projected()`		`isLonLat()`
get CRS		`st_crs()`	`crs()`
change CRS	`spTransform()`	`st_transform()`	`projectRaster()`

Weirdly enough, many functions of the `[raster::]` package also work on *vector* data objects. Ex: to read or write a shapefile: `raster::shapefile(x, filename='', overwrite=FALSE)`

3.2 Other packages

For netCDFs: `[ncdf4::]`

NetCDF is an *array*-oriented data format commonly used in climatology, meteorology and GIS applications. NetCDF variables can be 1-dimensional vectors, 2-dimensional matrices, n-dimensional arrays.

Steps to read in the contents of a netCDF (.nc) file:

(1) `nc_open()`: open the file

(2) `print()`: see the metadata

(3) `ncvar_get()`: get the data from a variable; then store locally as array

(4) `ncatt_get()`: get variable attributes (projection, fill value...)

- replace the fill-values (i.e., grid cells with missing data) by `NA`

(5) `nc_close()`: close the file

4 Operations on spatial layers

4.1 Converting data from one class to another

Converting `sf` objects into `sp` objects
Some packages still rely on `sp` objects, so we might have to convert `sf` objects:

- Convert `sf` to `sp`: `myObject.sp <- sf::as_Spatial(myObject.sf)`
- Convert `sp` to `sf`: `sf::st_as_sf()`

Creating a spatial object from a lat,lon table

(1) Convert into a spatial object: by identifying the coordinates as such
(for point data, `coords` must be a vector that specifies the df's columns for the lon,lat coordinates)

```
myObject.sf <- sf::st_as_sf(myObject.df, coords)
```

(2) Define its projection

```
sf::st_crs(myObject.sf) <- 4326
# or
sf::st_crs(myObject.sf) <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84
    +towgs84=0,0,0"
```

4.2 Unprojecting projected data

If data are projected ({x,y} coordinates), in order to plot them over an unprojected {lon, lat} grid, we need to unproject the data. To do this, we notably need to know the original CRS (hopefully provided in the metadata).

```
# [Prelim.] convert data.frame to raster, making original CRS explicit
myVar_XY.raster <- rasterFromXYZ(myVar.df, crs = CRS("+proj=lcc +..."))

# 1. unproject: change projection to lat-long. Choose a datum (e.g., WGS84)
myVar_latlon.raster <- projectRaster(myVar_XY.raster, crs = "+proj=longlat +datum
    =WGS84 +no_defs")

# 2. convert (our new unprojected raster) back into a data.frame
#  -- save list of coordinates
coord_latlon.raster <- coordinates(myVar_latlon.raster) %>% as.data.frame()
lon_raster <- unique(coord_latlon.raster$x)
lat_raster <- unique(coord_latlon.raster$y)
#  -- convert raster to array
myVar_latlon.array <- as.array(myVar_latlon.raster)[,,1]
dimnames(myVar_latlon.array) = list(lat_raster, lon_raster)
#  -- convert array to df
myVar_latlon.df <- reshape2::melt(myVar_latlon.array, varnames=c("lat","lon"))
```

4.3 Misc. single- and multi-layer operations

Single-layer

- Compute centroids (geographic centers): `sf::st_centroid()`
- Join polygons (if they touch or overlap, internal boundaries are "dissolved"):
  - `sf::st_union()`
  - `raster::aggregate(,dissolve=TRUE)`

Multi-layer

- Crop a raster: `raster::crop(x,y)`
- Crop a raster using a polygon: `raster::mask()` then `raster::trim()`
  ⚠ `raster::mask()` removes the cells that are only partially within the polygon. If one wants to keep those, follow the steps:
  (1) get, for each cell, the fraction that's within the polygon:
  `myMask.ras <- raster::rasterize(x=myPolygon, y=myRas.ras, getCover=TRUE)`
  (2) replace 0s by NAs: `myMask.ras[myMask.ras==0] <-NA`
  (3) use that as the new mask:
  `trimmedRas.ras <-raster::mask(x=myRas.ras, mask=myMask.ras)%>% raster::trim(values=NA)`
- Find which points in `x` fall inside a polygon in `y`: `sf::st_intersects(x, y)`
  (which points fall inside a polygon = the intersection of the points with the polygons)
- Extract cell values at locations of interest, using an input vector layer: `raster::extract(x, y)`
  – with points, returns the raster values under each point
  – with polygons, return all values in each polygon, or a summary (e.g. mean)

5 Plot using [ggplot2::]

5.1 Visualizing usual spatial objects: geom_point(), _polygon(), _tile()

ggplot does not work with Spatial* or Raster* objects. They must therefore first be converted into data frames:

- for a SpatialPointsDataFrame:

```
myData.df <- data.frame(myPointData.shp)
ggplot() +
   geom_point(data = myData.df, aes(x = lon, y = lat))
```

- for a SpatialPolygonsDataFrame, fortify the object:

```
myData.df <- fortify(myPolygonData.shp, region = "id")
ggplot() +
   geom_point(data = myData.df, aes(x = lon, y = lat, group = group))

ggplot() +
   geom_polygon(data = ISR.shp, aes(x = lon, y = lat, group = group)) +
   coord_map(projection = "albers", parameters = c(25,50))
```

- for a Raster object:

```
# after having converted the Raster into a data.frame
ggplot() +
 geom_tile(data = myData.df, aes(x, y, fill = value))
```

Choose a projection with coord_map():

- Mercator: cylindrical: equally spaced straight meridians, conformal

```
+ coord_map()
```

- conformal, true scale on lat0 and lat1

```
+ coord_map(projection = "lambert", parameters = c(25,50))
```

- Albers conic equal-area projection, true scale on lat0 and lat1

```
+ coord_map(projection = "albers", parameters = c(25,50))
```

5.2 Visualizing sf objects: geom_sf()

geom_sf() is an unusual geom as it draws different geometric objects, depending on what geometries are in the data: points, lines, or polygons.

```
ggplot() +
   geom_sf(data = myData.sf, aes(geometry = geometry)) +
   coord_sf(crs = "+proj=longlat +datum=WGS84 +no_defs")
```

A Non spatial-specific commands

Below is a jumble of commands which are not specific to spatial data, but which one often has to use when wrangling spatial data.

Reshaping 3-dim data arrays into 2-dim data frames

Example: one may want to reshape some raster variables stored as 3-dim data array (*dims*={*lat,lon,time*}) into a 2-dim long-format dataframe, s.t.:
– one row corresponds to one grid cell in one time period (number of rows = $[lat] \times [lon] \times [time]$);
– columns are *lat, lon, time, variable1, variable2, ...*

```
myVar.array
dimnames(myVar.array) = list(lon_values, lat_values, time_values)
myVar.df <- reshape2::melt(myVar.array, varnames = c("lon","lat","time"))
```