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. Ex: the `sp` classes
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... Some formats can contain both, e.g., .gdb (Esri's geodatabase file format).

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, TIFF • 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 located on the Earth's 3-D surface. All vector and raster data express these locations using sets of coordinates, so they use a framework to precisely measure these locations on the Earth's surface as coordinates, called a Coordinate Reference System (CRS). To conduct any spatial analysis using several layers, we must first put them in the same CRS. The next page describes these CRSs in details.

[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

Coordinate Reference System (CRS)

A CRS is a framework to measure locations on the Earth's surface with coordinates. A CRS can be represented using either an EPSG numeric code,[2] a 'proj4string' which cites the CRS' core key elements, or a 'well-known text' representation which lists all its elements.

There are two types of coordinates and therefore of CRSs:

- **unprojected or 'geographic'**: locations are defined with **{lon, lat} coordinates.** Ex: `EPSG:4326`
  The Earth is referred to as a 3-D object.
- **projected**: locations are defined with **{x, y} coordinates.** Ex: `EPSG:3857`
  Starting from a base geographic CRS, the 3-D surface is translated to a 2-D surface using some projection (*UTM, Mercator...*). Latitude and longitude are themselves functions of {x,y}.

The table below lists the imbricated elements of unprojected and projected CRSs, using an example of each:

Element	Example of geographic CRS WGS 84 (EPSG 4326)	Example of projected CRS WGS 84 Pseudo-Mercator (EPSG 3857)
• Base geographic CRS	WGS 84 (EPSG 4326)	WGS 84 (EPSG 4326)
– Datum	WGS 84 ensemble (EPSG 6326)	''
* Model of Earth's shape	WGS 84 ellipsoid (EPSG 7030)	''
* Anchor point		
– Prime meridian	Greenwich (EPSG 8901)	
– Coordinate System[3]	Ellipsoidal, 2-dim, degree (EPSG 9122)	''
– Area of use	World	''
• Projection or "conversion"	–	
– Method		Pseudo Mercator (EPSG 1024)
– Parameters		
• Coordinate System	–	Cartesian, 2-dim, metre (EPSG 9001)
• Area of use	–	World between 85.06°S, 85.06°N

Misc. notes:

- The full list of elements can be seen using the CRS's well-known text (WKT) representation. One can find it in R with the function `sp::CRS("EPSG:<MY_EPSG>")`, or on the EPSG registry: `https://epsg.io`.
- In R, the datum `EPSG:6326` is the default assumed for a CRS declared with a PROJ string.
- One can set a CRS by using a EPSG code in a proj4string definition with `+init=epsg:<EPSG>`. This will make the PROJ library automatically add the rest of the parameters.
- As a general rule, one should always use a *projected* CRS when doing analysis. Decimal degrees are not uniform on the Earth's surface, therefore geographic CRS are not ideal for measuring distances.

[2] CRSs have EPSG codes, but so do individual elements of CRSs... Such that when one displays the full detailed representation of a CRS, one will see many imbricated EPSG codes. EPSGs are unique identifiers, whereas names can be ambiguous. For example, "World Geodetic System 1984" (WGS84) may refer to the CRS, its datum, or its ellipsoid model, but these three elements have distinct EPSG codes: 4316, 6326, and 7030.

[3] A Coordinate System is a set of axes with units. Geodetic axes (longitude and latitude) have an angle unit (e.g., degree), others, e.g., height, have a length unit (e.g., metre).

Examples of common CRSs:

- Google Earth is in the geographic CRS "WGS 84" (EPSG code: `4326`)
- Google Maps and many other popular web mapping applications are in the projected CRS "WGS 84 Pseudo-Mercator"
  - EPSG code: `3857`
  - PROJ.4 representation:[4]

```
+proj=merc +a=6378137 +b=6378137 +lat_ts=0 +lon_0=0 +x_0=0 +y_0=0 +k=1
    +units=m +nadgrids=@null +wktext +no_defs +type=crs
```

  - WKT2 representation:

```
PROJCRS["WGS 84 / Pseudo-Mercator",
  BASEGEOGCRS["WGS 84",
    ENSEMBLE["World Geodetic System 1984 ensemble",
      MEMBER["World Geodetic System 1984 (Transit)"],
      MEMBER["World Geodetic System 1984 (G730)"],
      MEMBER["World Geodetic System 1984 (G873)"],
      MEMBER["World Geodetic System 1984 (G1150)"],
      MEMBER["World Geodetic System 1984 (G1674)"],
      MEMBER["World Geodetic System 1984 (G1762)"],
      MEMBER["World Geodetic System 1984 (G2139)"],
      ELLIPSOID["WGS 84",6378137,298.257223563,
        LENGTHUNIT["metre",1]],
      ENSEMBLEACCURACY[2.0]],
    PRIMEM["Greenwich",0,
      ANGLEUNIT["degree",0.0174532925199433]],
    ID["EPSG",4326]],
  CONVERSION["Popular Visualisation Pseudo-Mercator",
    METHOD["Popular Visualisation Pseudo Mercator",
      ID["EPSG",1024]],
    PARAMETER["Latitude of natural origin",0,
      ANGLEUNIT["degree",0.0174532925199433],
      ID["EPSG",8801]],
    PARAMETER["Longitude of natural origin",0,
      ANGLEUNIT["degree",0.0174532925199433],
      ID["EPSG",8802]],
    PARAMETER["False easting",0,
      LENGTHUNIT["metre",1], ID["EPSG",8806]],
    PARAMETER["False northing",0,
      LENGTHUNIT["metre",1], ID["EPSG",8807]]],
  CS[Cartesian,2],
    AXIS["easting (X)",east, ORDER[1],
      LENGTHUNIT["metre",1]],
    AXIS["northing (Y)",north, ORDER[2],
      LENGTHUNIT["metre",1]],
  USAGE[
    SCOPE["Web mapping and visualisation."],
    AREA["World between 85.06S and 85.06N."],
    BBOX[-85.06,-180,85.06,180]],
  ID["EPSG",3857]]
```

[4]Recently, it has become discouraged to use proj4strings to represent a CRS, and recommended to use them to specify transformations between CRSs, but not specify a CRS itself.

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	`CRS()`	`st_crs()`	`crs()`
change CRS	`spTransform()`	`st_transform()`	`projectRaster()`
all of the above	`summary()`		

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

We may need to convert data from one class to another. For example, functions from the `raster` package do not work with `sf` objects, so we have to convert those to `sp` objects.

- Convert `sf` to `sp`: `sf::as_Spatial()` or `as(, "Spatial")`
- Convert `sp` to `sf`: `sf::st_as_sf()`
- Convert `data.frame` to `sf`: by identifying the coordinates as such

```
sf::st_as_sf(, coords = c("lon", "lat"), crs = 4326)
```

- Convert `raster` to `data.frame` (the stored coordinates correspond to the cell's centroid)

```
myData.df <- raster::as.data.frame(myData.ras, xy=TRUE, long=TRUE) %>%
 separate(layer, into=c("filename", "layer"), sep="\\.", convert=TRUE)
```

4.2 Misc. single-layer and multi-layer operations

Single-layer

- Compute buffers: `sf::st_buffer(, dist)`
- Compute polygon centroids: `sf::st_centroid()`
- Compute polygon areas: `sf::st_area()`
- Compute the coordinates of the bounding box: `sf::st_bbox()`
- Create a bounding box: `sf::st_make_grid(, n = 1)`
- Make a grid covering your data: `sf::st_make_grid(, n)`
- Dissolve features into a single feature
  - Polygons; Points (get bundled into a single 'multipoint' geometry; required step before creating a convex hull): `sf::st_union()`
  - Raster cells: `raster::aggregate(, dissolve = TRUE)`
- Crop a raster: `raster::crop(x,y)`

Multi-layer

- Compute relationships between objects
  - Which `x` features intersect with / are fully contained in `y` polygons: `sf::st_intersects(x, y)`, `sf::st_contains(x, y)`
  - Distance between features: `sf::st_distance(x, y)`
  - Extract cell values from `x` at locations of interest in `y` : `raster::extract(x, y, fun)`
    - with points, returns the raster values under each point

    * with polygons, returns either all values in the polygon `(fun = NULL)`, or a summary, e.g., `(fun = mean)`

- Modify objects
  - Spatial join: add attributes/information in `y` to the `x` layer `sf::st_join(x, y)`
  - Clip features in `y` to the `x` polygons: `sf::st_intersection(x, y)`
  - Crop a raster using polygons: `raster::mask()` then `raster::trim()`
    ⚠ `raster::mask()` removes cells that are only partially within a polygon. To keep those, do:
    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)
```

  - Raster math: `raster::overlay(x, y, fun)`

    Ex1: multiply a raster by another:

```
f <- function(rast1, rast2) rast1 * rast2
new <- overlay(elevation, multiplier, fun = f)
```

    Ex2: filter a raster on itself and by another one

```
f <- function(rast1, rast2) rast1 < 20 \& rast2 > 80
new <- overlay(canopy, impervious, fun = f)
```

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.sp)
ggplot() +
   geom_point(data = myData.df, aes(x = lon, y = lat))
```

- for a SpatialPolygonsDataFrame, tidy the object:

```
myData.df <- broom::tidy(myPolygonData.sp, region = "id")
ggplot() +
   geom_polygon(data = myData.df, aes(x = lon, y = lat, group = group)) +
   coord_map(projection = "albers", parameters = c(25,50))
```

- for a Raster object:

```
# a. convert into a data.frame
myData.df <- as.data.frame(myRasterData.ras, xy=TRUE)
# b. plot
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)) +
   geom_polygon(data = myOtherData.sp, aes(x=lon, y=lat, group=group)) +
   coord_sf(crs = 4326)
```

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"))
```