Stat 406, Spring 2020

Lab exercise - using the sp library

The material below illustrates some of the functions in the sp library. This library provides an extensive set of functions and classes for spatial data. It allows a rich set of GIS-like objects: points, lines, pixels, and polygons. Bivand et al. Chapter 2 describes all the classes defined by sp and Chapter 4 shows how to create sp data sets. Both have much more detail that we need. If you want to know more about anything described below, look in Bivand. We'll just use points and spatial data frames now.

We are discussing sp because it has become the standard for storing spatial data of any type (geostatistical, areal, or point). In order to analyze data, we will have to save it in one of the sp formats.

There is a new library, sf (spatial features). The sp library is not able to make certain types of data conversions. Those conversions are not an issue for us. The sf library is able to make those conversions. sf also provides even more GIS-like functionality than sp does. sf also has a more logical data structure. The way sp objects are stored in R is complex (lists of lists, and sometimes a third level of lists). There are good reasons to do this, but it makes it complicated to extract or modify one particular piece. Two years a go, I was told that sf will become the standard sometime in the next few years. It hasn't happened yet. When that happens, you will need to learn about sf instead of sp. For now, we cover the use of sp. If you want to use R as a GIS (yes, you can do that), you need to learn both sp and sf.

Reminder about libraries and how they work:

We will use a data set of rainfall at locations in Switzerland in the first part of the course. Download it from the class web site, save it in your working directory as swiss.csv, then type the following:

```
swiss <- read.csv('swiss.csv', as.is=T)
coordinates(swiss.sp) <- c('x','y')
```

You should get an error message that R could not find the function coordinates().

R has a huge number of add-on functions packaged in libraries (also called packages). Most of the functions we use will be defined in libraries, not base R. You need to attach the appropriate library before you can use the function. One of two things will happen if you forget to attach the appropriate library:

- You get the error above, because there is (not yet) a coordinates() function.
- You get something really weird and unexpected. That is because there are two (or more) versions of the coordinates() function. The desired one is not loaded; R found something else instead.

The solution is the same: attach the library before trying to use any of its functions.

```
library(sp)
coordinates(swiss.sp) $<$- c('x','y')\\
```

Now coordinates() works.

Most folks organize their code into files and put all the library() calls at the top of the code file.

If R doesn't find the sp library, you need to install it:

```
install.packages('sp')
```

Installing a package just puts it on your hard drive. You only need to do that once. You need to attach the library each time you start R. Note that the package name is in quotes when you install it and not in quotes when you attach it.
If you update your version of R, you will need to reinstall all your libraries.
You will need to install the sp package package. If sp doesn't automatically download the rgdal package, you'll also need to install that.

1. **Creating a spatial data frame:**

   A spatial point data set is just a set of coordinates. A spatial data frame has coordinates and additional information. The Swiss rain data already has X and Y in meters. You can create a spatial data frame (or point data set) many ways. Bivand et al. has all the details in Chapters and 4. Below are my simple ways to create spatial objects.

   You will need the swiss rainfall data set and the Iowa county data set from the class web site. See the Intro to R notes if you don't know how to read a data set into R. The read.csv() function is one of many ways. Use what works for you.

   Note about factors: My practice is to explicitly identify variables to be used as factors (identifying groups, instead of serving as a continuous covariate). That means I turn off the default conversion when reading character strings. The as.is=T option leaves character strings "as is", i.e. not converted to factors.

   - library(sp)
   - swiss <- read.csv('swiss.csv', as.is=T)
   - swiss.sp <- swiss
     # make a copy of the swiss data frame with the name swiss.sp
     # I often use tags .XXXX so I know two objects are related. # that is frowned upon by R gurus. They prefer swissSp (no .).
   - coordinates(swiss.sp) <- c('x','y')
     # convert swiss.sp to a spatial object by specifying the coordinate variables

     # the next two commands illustrate the change in structure from a data frame
     # to a spatial object. The output looks different because the data frame is
     # an S3 object; the spatial object is a S4 object.
   - str(swiss)
   - str(swiss.sp)

     An S4 object is like the data frames you are used to, except that much more rigorous rules are enforced about its contents. That means a function that wants an S4 object can expect specific things to be present in specific places. The consequence for a user is that is a lot harder to create an S4 object. Generally, they have to be created and manipulated using functions, not by directly accessing the components. You can legally write `swiss$new <- 5` to add a new column to the swiss data frame. You can't do that with an S4 object.

     The Formal class name identifies what type of object this is. Since the swiss data frame has more information than the coordinate values (id identifies the weather station, rain contains the annual rainfall), swiss.sp is a SpatialPointsDataFrame. That is a set of

locations with additional information. If we started with just coordinates, we would get a SpatialPoints object.

The difference between a SpatialPoints and a SpatialPointsDataFrame isn't usually a concern. sp functions do the obvious and expected things. The end of this file demonstrates how to add data to a SpatialPoints object. That is one thing that has been made much simpler in the last two years.

Components of an S4 object are called slots and identified by @. str identifies the slots and their components. You can get just the names of the slots using:

- slotNames(swiss.sp)

  You see 5 components:

  - coords: the coordinate values you specified using coordinates()
  - coords.nrs: info used by coordinates() and not needed by the user
  - bbox: the bounding box for the locations. This is the rectangle that encloses the data set
  - proj4string: the coordinate system for the locations (the "projection"). We didn't specify this when we created the spatial object, so it is currently null. Null projections will be treated as rectangular with Euclidean distances. Further down, we will see how to specify the projection and change projections.
  - data: the non-coordinate information in the swiss data frame is stored in the data slot. This is a standard R data frame. If the slot exists, it is easy to add additional columns to the data frame. You just add new columns in the usual way. This will be demonstrated later.

- names(swiss.sp@data)
  # return variable names in the data frame of the spatial object. These are the names of the additional pieces of information for each location.

2. **Plotting locations**

   This can be done multiple ways: by using the x,y values in the original data frame, by extracting coordinates from the spatial object and plotting those, or by using sp plotting function.

   - plot(swiss$x,swiss$y)
     # x, y values in the data frame
   - plot(coordinates(swiss.sp))
     # plotting the matrix returned by coordinates()
     str(coordinates(swiss.sp))
     # what coordinates extracts, a matrix with named columns.
     # Plot knows what to do because there is a column named x and a column named y.
   - plot(swiss)
     # and plot knows what to do with a data frame with x and y columns
   - plot(swiss.sp)
     # plot of locations in a spatial object, using a plot method from the sp package

3. **sp plotting refinements**

There are lots of options to the sp plotting functions. ?SpatialPoints tells you about them. There are also lots of functions that extend basic plots. I will introduce options and functions as we need them. Here are some simple, often needed, operations.

- plot(swiss.sp, axes=T)
  # add axes to the plot
- Selecting spatial locations:
- select.spatial(swiss.sp)
  # plots the locations, then allows you to select subsets of points
  # left click on the plot to indicate corners of a polygon
  # do not need to close the last side, then right click, chose stop
  # returns vector of row numbers of the points in the polygon.
- select.spatial(swiss.sp, digitize=F)
  # like identify() for non-spatial object. selects points individually.
- To stop selecting points, hit the Esc (escape key) in Rstudio or right-click the mouse in plain R
- WARNING: if you start the locator but don't select any points, then escape, Rstudio will often (usually? always?) crash. Very impolite.

4. **Various potentially useful plots**

- bubble(swiss.sp, 'rain')
  first argument is the spatial object, second is the variable to plot, in quotes. Positive values in green; negative in red. Size of circle indicates magnitude of value. Especially useful for residuals.
- spplot(swiss.sp)
  # trellis plot of all data variables, using colors to indicate level
  # to add axes to bubble() or spplot(): axes=T doesn't work
  # that's because bubble and spplot use the trellis graphics system
  # which has a very different structure for arguments
- bubble(swiss.sp, 'rain', scales=list(draw=T))
  spplot(swiss.sp, 'rain', scales=list(draw=T))
  # you can change the color scheme to whatever you like.
  # you just have to specify what you want - most are really complicated
  # the easiest way is to specify one of the predefined palettes
  # the plot in the notes used cm.colors(10)
- spplot(swiss.sp, 'rain', col.regions=cm.colors(10))
  # the number has to be at least as big as the number of categories
  # see ?rainbow to get the list of palettes
  # I have sometimes found the hcl palette very useful
  spplot(swiss.sp, 'rain', col.regions=hcl.colors(10))
  # Cynthia Brewer has done a lot of work on good palettes, especially for maps
  # She maintains a web site that allows you to play with different color schemes
  # that is colorBrewer2.org
  # The RcolorBrewer library implements her palettes as R color palettes

- plot(swiss$x,swiss$rain)
  # easiest way to look at trend for X or Y is to use the non-spatial object
- library(lattice)
  swiss$ygrp <- cut(swiss$y, 6)
  # create 6 groups of Y coordinates, store as ygrp
  xyplot(rain ~ x | ygrp, data=swiss)
  # plot rain vs x for 6 strips of Y values (6 because that's the number of groups for cut).
  # example of trellis conditioning plot in lattice package

5. **Converting between coordinate systems**

   The IAcounty.txt data file on the class web page has the lat long coordinates of some of the county seats in IA. The following assumes the file is saved in your working directory.

   - iac <- read.table('IAcounty.txt', as.is=T, header=T)
   - names(iac)
   - iac$long <- -iac$long
     # Longitudes are W, so need to be negative numbers
   - iac.sp <- iac
   - coordinates(iac.sp) <- c('long', 'lat')
     # longitude (x) MUST come before latitude
   - proj4string(iac.sp) <- CRS('+proj=longlat +datum=NAD27')
     proj4string() sets the Coordinate Reference System for the data. The CRS() function takes a character string with the information you provide and adds more stuff. This is then assigned to the proj4string slot in the spatial object. Note that the argument is a single character string with two (or more) tokens, each starting with +.

     longlat is latitude/longitude, datum is the map datum. The 3 most common choices for datum are NAD27 (old US maps), NAD83 (newer US maps) and WGS84 (GPS). Bivand has the details. Some may matter if you need to do very very careful mapping.
   - Converting between coordinate systems:
     - spTransform() does the hard work
     - but you must define the starting coordinate system (see proj4string() above)
     - spTransform() is in the rgdal library
     - the first argument is the spatial object; the second is the new coordinate system. CRS() is used to specify the new system.

     library(rgdal)
     iac.utm <- spTransform(iac.sp, CRS('+proj=utm +zone=15'))
     # convert coordinates to UTM in specified zone
   - coordinates(iac.utm)
     # print the UTM coordinates.
     # see what happens to plot axes when change from longlat to UTM
   - plot(iac.sp, axes=T) # Notice the axis labels.
     # The version of plot in the sp library understands lat long
   - plot(iac.utm, axes=T)

- omaha <- SpatialPoints(cbind(-95.9, 41.3),
  proj4string=CRS('+proj=longlat +datum=NAD83'))
  # The 'where is Omaha' (lat: 41.3N 95.9W) example
  # SpatialPoints is a second way to convert a matrix of locations to a spatial object
  # The first argument needs to be a matrix with 2 columns. That's why cbind() not c()
  # If you try to use c(), the error message is spectacularly unhelpful
- library(rgdal)
- spTransform(omaha.sp, CRS("+proj=utm +zone=15"))
  # UTM treating Omaha as in zone 15
- spTransform(omaha.sp, CRS("+proj=utm +zone=14"))
  # UTM treating Omaha as in zone 14
  # notice difference in northing (coords.x2),
  # plus you need to figure out how to match west edge of 15 to east edge of 14

6. Lat/Long in degrees, minutes, seconds
   Often lat long is recorded in degrees, minutes, seconds, not decimal degrees, 41d 18' 40", where d is degrees, ' is minutes and " is seconds. sp provides functions to convert.

   - omahalat <- "41d18'40\"N"
     omahalong <- "95d50'00\"W"
     # because " starts and ends the character string, need to protect the "
     # when you enter second symbol. That's why there is a \ before the "
     # that indicates seconds, then the quote ending the character string.
   - char2dms(omahalat)
     # char2dms() converts a character string with degree, minute and seconds info
     # to a DMS class object, which are automatically printed nicely
   - as.numeric(char2dms(omahalat))
     as.numeric(char2dms(omahalong))
     # and coercing the DMS object to a number gives you decimal degrees.
     # Note negative long (because W).
   - dd2dms(-93.6)
     # dd2dms() converts from decimal degrees to DMS, which print nicely

7. **Calculating distances between points**

   Let's compute the distance from Ames to each of the county seats in iac. We need the location of Ames in UTM. Could get from a map or by converting the latlong to UTM. Code below does it by the conversion.

   - ames.sp <- SpatialPoints(cbind(-93.6, 42.0),
     proj4string=CRS('+proj=longlat +datum=NAD83'))
   - ames.utm <- spTransform(ames.sp, CRS("+proj=utm +zone=15"))
     # ames.sp is the location of Ames in longlat; ames.utm is the location in UTM
   - utm <- spDistsN1(iac.utm, ames.utm)
     # spDistsN1() computes distance from the 2nd argument (one location)
     # to each point in the 1st argument (can be one loc. or a matrix with many locs.)
     # Because iac.utm is UTM, spDistsN1() computes Euclidean distances,

in m from Ames to each county seat, based on UTM coordinates # Distance in m because that's the UTM unit

- gc <- spDistsN1(iac.sp, ames.sp, longlat=T)
  # adding longlat=T tells spDistsN1 that locations are long lat
  # in this case, spDistsN1() knows to compute
  # great circle distances, in km from Ames to each county seat
  # distances in km because that's the default for great circle distances
- plot(gc, gc-utm/1000)
  abline(h=0)
  # compare the two distances, abline(h=) adds horizontal line
- You need to be careful that the two arguments to spDistsN1() have the same coordinate system. Many (most?) sp functions check this and throw an error if they are not the same. spDistsN1() doesn't. You would also think it would pick up the coordinate system from the proj4string slots, but it doesn't.

The plot comparing great circle and euclidean distances looks different than what I had in the lecture notes. The data file you downloaded used is a more accurate set of county seat locations. I don't know whether that is cause of the difference, or whether sp or rgdal functions have changed.

8. **Adding data to a Spatial Points object** This is much simpler than it was 2 years ago. All you need to do is create a new variable (not a coordinate name) in the SpatialPoints object.

   IAcountyLocs.csv is a data file with only long and lat coordinates. When you convert it to an sp object, it is a SpatialPoints object.

   - Read a csv file, create negative longitude (for W), and convert to a spatial object

     ```
     ialoc <- read.csv('IAcountyLocs.csv')
     ialoc.sp <- ialoc
     ialoc.sp$long <- -ialoc.sp$long
     coordinates(ialoc.sp) <- c('long', 'lat')
     ```

   - class(ialoc.sp) # this is a SpatialPoints object
   - slotNames(ialoc.sp) # and does not have a data slot
   - ialoc.sp$name <- iac$name
     iac$name is has the county name (read in a while ago from IAcounty.txt
     To add that data to the SpatialPoints object, all you need to do is assign a new variable.
   - class(ialoc.sp); slotNames(ialoc.sp)
     # ialoc.sp is now a SpatialPointsDataFrame with a data slot
   - head(ialoc.sp$name) # you can access the data easily, using its variable name
   - head(ialoc.sp) # and printing the sp object does nice things

9. **Summary** This is a very short introduction to the sp package. Bivand et al. describe all sorts of other things you can do with the sp package. A lot looks like the sorts of manipulations you might otherwise do in a GIS. We will not use GIS-like operations in this course.

   Over the last decade, the sp package has been actively developed. Every year, sp has been augmented and improved. sf is now proposed to replace sp. I don't know whether or not development of sp will continue.