GIS in R with the `terra` package

Tutorial by Sarah Namiiro & Andreas Hamann

Permanent link to video walkthrough: http://tinyurl.com/mv690/seminars/terra
Permanent link for a PDF file of this lab: http://tinyurl.com/mv690/seminars/terra/lab

This lab is an introduction to the `terra` package for R. The package provides basic GIS functionality to load, process and display spatial data files. If you have never used GIS, but you are familiar with R, then this is a very easy introduction to GIS. The `terra` package is compatible with a huge range of spatial data formats without any fuss. Because `terra` is a GIS system build on R scripts, it is great for automating complicated GIS analysis tasks. It's also easy to create basic custom maps for your study area and sample sites. However, it's not meant to output great-looking cartography. For that, it's easier to use menu-based GIS systems, such as ESRI's ArcGIS or to open-source QGIS. For that, see http://tinyurl.com/mv690/labs/qgis.

1. Opening and working with vector files

Vector files contain spatial coordinate information for points, lines or polygons. For this lab, we have already prepared a data package with such spatial data files that you can download here: http://tinyurl.com/mv690/seminars/terra/data.

There are many data formats for vector files, and here we work with one of the most common formats, ESRI shapefiles, which consist of 4 different files that have the same name but different extensions. The files with the .shp and .shx extension contains the coordinates and vectors required for drawing shapes, the .prj file contains projection information (e.g. geographic lat/long or projected systems like UTM), and the .dbf file is an attribute table that describes the properties of each point, line, or polygon.

Shapefiles for all kinds of things can be obtained from many websites, and the UofA library has an excellent guide to these resources: https://guides.library.ualberta.ca/geospatial-data-maps. If you follow the links on the left of that page, you can find almost anything you need. Another well organized and widely used source of data is https://www.naturalearthdata.com/downloads/ that covers many basic needs. You don't need to worry about the data formats of data downloads. The `terra` package will automatically detect any data format for import.

- Let's start by installing and loading the `terra` library:

```
install.packages('terra')
library(terra)
```

- Then read in a polygon file of administrative boundaries for western Canada. This would be something you can download for any part of the world from the web. To see the polygons, rather than just an outline of the polygons, you need to specify a color.

```
wcan = vect("outlines.shp")
plot(wcan, col="gray90")
head(wcan)
```

- The last header command will show you the first six rows of the attribute table, and we can use the information therein to subset the vector file. If you work with a downloaded dataset, you may only be interested in a portion of it, so let's extract just the Alberta polygon, like we are used to in R:

```
ab = wcan[wcan$postal == "AB",]
plot(ab, col="gray90")
head(ab)
```

- Can you import the road layer from the data file and add it to your Alberta map? This is the command:

```
plot(roads, add=T)
```

2. Dealing with projections for spatial data

One of the more challenging parts when working with GIS data is that data files come in different projections that attempt to faithfully represent areas, distances and shapes (i.e. projecting the round surface of earth on a flat piece of paper). The bigger the area, the more difficult this is. Therefore there are hundreds of local projections that work well for a single country, province or state. Those projections get revised and updated, so there are literally hundreds of projections in active use or needed to import legacy datasets.

The first layer you import defines the projection of your data frame, and any other layers you want to add to the data frame or plot will have to be of the same projection otherwise adding the new layer to the plot will not work.

- Let's check the coordinate reference system (CRS), of the file we just imported. It's UTM Zone 10. You can find a description and its EPSG-ID with a Google search at https://epsg.io/32610.

```
crs(ab)
```

- However, the coordinate reference system is not always included with the data, and sometimes it is entirely omitted. If it's omitted it often means that the coordinate system is unprojected and displayed in latitude and longitude, which you can also infer from horizontal and vertical boundaries. Let's load a shape file that lacks this information. To use it, we still need to defined the coordinate system:

```
geol = vect("geol_srf_latlong.shp")
head(geol)
crs(geol) # projection undefined
plot(geol, col="gray90") # looks like WGS84 (latlong)
crs(geol) = "EPSG:4326" # Search on http://epsg.io
```

- Now try adding a layer with a different projection. Our first layer was projected in a Universal Trans Mercator Region 10 projection (UTM-10) and we just gave **geol** a World Geodetic System 1984 (WGS 84) projection. Adding the layer to our plot will not work:

```
crs(ab)    # UTM-10
crs(geol) # WGS84
plot(ab, col="gray90")
plot(geol, add=T) # doesn't work
```

- First we need to re-project the layer to the same CRS as the first layer we displayed.

```
geol = project(geol, ab)
plot(geol, add=T)  # now it works!
```

3. Coloring a layer by attributes

- Now that we have our geology layer (**geol**) added, we can display the different features of our surface geology (or any other variable) using different colors. To do this with purpose, let's explore the the variables in the attribute table:

```
head(geol)
labels = sort(unique(geol$SurfGeol))
labels  # 19 classes of surface geology
ordered = factor(geol$SurfGeol, ordered=T, levels=labels)
```

- Let's set the Glaciers and Water in position 8 and 19 to "white", and for the rest use colors:

```
library(RColorBrewer); display.brewer.all()
geocol = c(brewer.pal(7, "Pastel2"), "white", brewer.pal(10,
"Set3"), "white")
```

- We then plot and color the surface geology. You can position the legend with in-built positions like "bottomleft", "topright" "center", etc. However, these may not always work well so you define the position using the x and y values in your plot.

```
plot(geol, col=geocol[ordered])
legend("bottomleft", legend=labels, fill=geocol, cex=0.6)
plot(geol, col=geocol[ordered])
legend(x = 150000, y =5800000 , legend=labels, fill=geocol,
       cex=0.4, bty="n")
```

4. GIS operations with your own data

When working with your own data in GIS, a common objective is to extract data from other spatial layers for specific points. For example, if you obtained coordinates with a GPS for your field sites (or if you are able to determine them accurately with Google Maps), you can use these coordinates to obtain additional information such as climate, elevation, or soil type, from GIS layers produced by other research groups. Subsequently, you can for example analyze if this additional information has any association with the measurements in your own study.

For this tutorial, we created a dataset similar to you might have yourself: a CSV file that contains coordinates for sample plots. Open the file "AB_Trees.csv" with tree species frequency information. We will use this example file to show you how to import this as a GIS layer and extract information for those coordinates. You should be able to do the same with your own data tables, which may contain anything between one study site and thousands of sample plots.

- Import the data table with the coordinates of your sample points and convert it into a spatial vector file. You need to define its projection, in this case they are unprojected Lat Long coordinates, and we already know the EPSG number from above. Then we reproject it to the same layer from which we like to extract data, here the bedrock geology information:

```
sites = read.csv("AB_Trees.csv")
head(sites)
sites = vect(sites, geom=c("LONG", "LAT"), crs="EPSG:4326")
sites_utm = project(sites, geol)
crs(sites_utm)
plot(sites_utm, add=T, pch=24, cex=1.4, col="white", bg="black")
```

- By converting the data table to a vector file, `terra` unfortunately removes the coordinate information. However, it's nice to keep track of this information, both in your original and projected coordinate system. It's easy to add that back to the vector file.

```
head(sites) # coordinates lost
sites_utm$LONG = crds(sites)[,1]
sites_utm$LAT = crds(sites)[,2]
sites_utm$X = crds(sites_utm)[,1]
sites_utm$Y = crds(sites_utm)[,2]
head(sites_utm)
```

- Now, let's extract information from the geology layer to our sample points and save the results. You can save them as a regular CSV and/or in your our preferred spatial data format. Format options are listed in the first column here: https://gdal.org/drivers/vector/index.html

```
head(geol) # just want cols 7 and 5 added
sites_geol = intersect(sites_utm, geol[,c(7,5)])
head(sites_geol)
write.csv(sites_geol, "sites_geol.csv", row.names=F)
writeVector(sites, "sites_geol.shp", filetype="ESRI Shapefile")
```

5. Working with raster data

Another important type of spatial file is raster data, which is essentially the same as an image file. This image file is formed by a matrix, in which each cell represents a location and a value for a variable of interest, such as climate or elevation. Rasters are widely used in remote sensing by satellites, and many interesting layers are derived from those, such as land cover, primary productivity, vegetation types, etc.

- Let's import a raster layer that contains elevation information, called a Digital Elevation Model (DEM)

```
dem = rast("DEM.asc")
plot(dem, col=terrain.colors(100))
dem
crs(dem)
head(dem)
summary(dem)
```

- Next we are going to extract the elevation values to the same point file from previous section:

```
head(sites)
crs(sites)
crs(dem) # need to match, otherwise: dem=project(sites, dem)
plot(sites, add=T)
ext = extract(dem, sites)
head(ext)
sites$LONG = crds(sites)[,1]
sites$LAT = crds(sites)[,2]
sites$ELEV = ext[,2]
head(sites)
```

- For display purposes, we can also re-projecting a raster grid but this results in some loss of quality. For other purposes, it's better to re-project your point file to the coordinate system of the raster.

```
dem # 0.01 degrees is about 1.5 km resolution
dem_utm = project(dem, ab, res=1500)  # 1500 m resolution
dem_utm
plot(dem_utm, col=terrain.colors(100))
plot(ab, add=T)
```

- If you don't need the full raster extent, you can crop and mask a raster to the same extent as a vector later (or another raster). Let's do it for Alberta and save the final product for later use:

```
dem_ab = crop(dem_utm, ab)
plot(dem_ab)
plot(ab, add=T)
dem_ab = mask(dem_ab, ab) # sets outside values to NA
plot(dem_ab)
plot(ab, add=T)

round(dem_ab, 0) # before saving, round to decimals you really need
writeRaster(dem_ab, paste0("DEM_ab.tif"), gdal=c("COMPRESS=DEFLATE"))
```

- Lastly it's sometimes a good idea to rasterize a vector layer for analysis or display purposes. This requires a source shapefile and a target raster extent that must be in the same projection. With the last command you can zoom into a specific area (click top-left then bottom right corner of the desired area on the plot window) to create a inset for your map:

```
crs(geol); crs(dem_ab)
ras_geol = rasterize(geol, dem_ab, field="SurfGeol")
plot(ras_geol, col=geocol)
plot(ab, add=T)

zoom(ras_geol)
```