Handling spatial data with R

Claudio Fronterre

29 October, 2019

Introduction

R allows to analyze, visualize and model geographic data in a open source environment. It merges the features of standard desktop Geographic Information System (GIS[1]), such as Arcmap, QGIS and GRASS, with flexible and powerful data processing, visualization and geospatial capabilities.

This tutorial will introduce you to the basics of working with geographical data in R. We will start with a description of the main types of geodata and then move to how we can manipulate and visualise them.

[1] A **G**eographic **I**nformation **S**ystem is a system for the analysis, manipulation and visualization of geographical data.

Geographical data structures

The two primary types of geospatial data are **vector** and **raster** data. Vector data are discrete objects represented by points. There are three main subtypes of vector data: points, lines and polygons. They are specially suitable for objects with well-defined borders (lakes, houses, streets, etc.). Raster data can be seen as a continous field represented by pixels, where each pixel (cell) is associated with a specific geographical location. The value of a pixel can be continuous (e.g. elevation) or categorical (e.g. land use). Only one attribute is assigned to each cell.

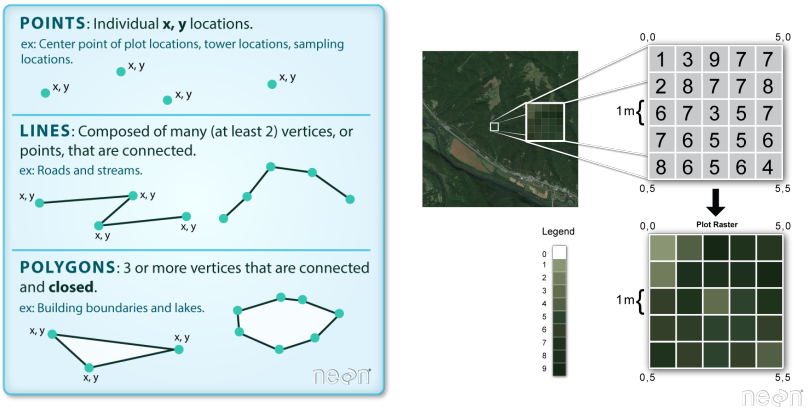


Figure 1: The three different types of vector data (left). An example of raster data (right). Source: National Ecological Observatory Network.

In this tutorial we will be using a mix of vector data (mainly points and polygons, i.e. the locations of villages where a survey was conducted) and raster data (usually satellite images that represents environmental variables, i.e. precipitation, temperature, elevation).

Coordinate Reference Systems (CRS)

Points in geographic vector data and cells in raster data are located on Earth with a **coordinate reference system** (CRS). We distinguish between **geographical** CRS and **projected** CRS.

Geographical CRS span the entire world. They are usually in decimal degrees (longitude and latitude). The most common geographical CRS is called **WGS 84**. Great for locating a place on Earth and by far the best for global analysis. However they are less suitable if you want to measure distance and heavily distorted towards the poles

Projected CRS are (usually) localized to minimize visual distortion in a particular region (use a specific ellipsoid which is especially suitable for a particular part of the Earth).

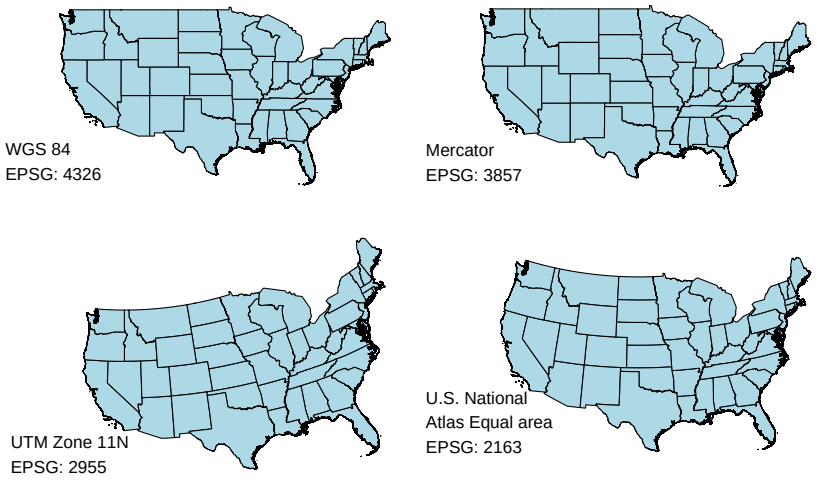


Figure 2: Maps of the united states in different CRS including WGS 84 geographic (top left), Mercator (top right), UTM (bottom left) and Alberts Equal Area (botto right). Notice the differences in shape and orientation associated with each CRS. These differences are direct result of the calculations used to flatten the data onto a two dimensional map.

This video highlights how map projections can make continents seem proportionally larger or smaller than they actually are.

There are lots of great resources that describe coordinate reference systems and projections in greater detail. For the purposes of this tutorial, what is important to understand is that data from the same location but saved in different projections will not line up in any GIS or other program. Thus, it's important when working with spatial data to identify the coordinate reference system applied to the data and retain it throughout data processing and analysis.

CRS in R can be described with either a **EPSG** code or a **proj4string** definition. Both of these approaches have advantages and disadvantages. An EPSG code

is usually shorter, and therefore easier to remember. The code also refers to a unique coordinate reference system. This website shows a list of EPSG codes along with its corresponding CRS. On the other hand, a proj4string definition allows you more flexibility when it comes to specifying different parameters such as the projection type, the datum and the ellipsoid. This way you can specify many different projections, and modify existing ones. This also makes the proj4string approach more complicated.

Spatial data in R

R has a big suite of packages that allow the manipulation, visualisation and analyis of spatial data. The CRAN Task View: Analysis of Spatial Data contains a list of R packages useful to perform different types of spatial operations. `sf` (simple features) and `sp` are the most important R packages to handle vector data. `sf` is a successor of `sp` and offers many advantages, including faster data input/output, more geometry types supported, a simpler structure and compatibility with the *tidyverse*. The package `raster` is an extension of spatial data classes to work with rasters. In this tutorial we will us the `sf` package as it is the state of the art for spatial data handling.

Importing and exporting vector data

In this section we will show how to load vector data into R and how to save them locally. First of all we will need to load[2] the `sf` package with the following command:

[2] This assumes that you already installed the `sf` package. If that is note the case you can install it with `install.packages("sf"")`

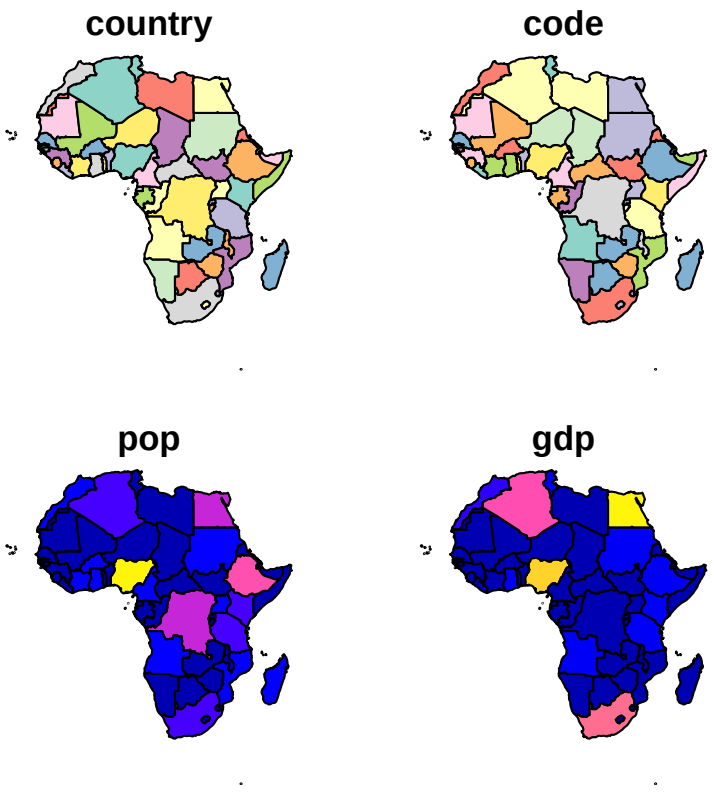
```
# Load the sf package
library(sf)
```

The main function to load vector data into R is `st_read()`. To find out which data formats `sf` supports, run `st_drivers()`. The first argument of `st_read()` is `dsn`, which should be a text string with the name of the spatial data that we want to load[3]. We now read-in a shapefile containing the administrative boundaries of african coutries. You will find a file called `africa.shp` in your `data`[4] folder.

[3] Remember that when loading that into R you always need to specify the extension of the file (i.e `"mydata.csv"` instead of `"mydata"`). Keep also in mind that if the file is not located in your working directory you need to provide the full file path.

[4] The data folder is contained in the resources folder of week 1 in the github repository of the spatial statistics reading group `https://github.com/claudiofronterre/spatstatReading`

```
# Loading a shapefile
africa <- st_read("data/africa.shp")

# Visualise the data
plot(africa)
```



We then used the base R `plot()` function to visualsie our shapefile[5].

The first thing that we need to do after loading spatial data is to check if the CRS is specified. We can do this using the `st_crs()` function. If the output of this function is `NA` the CRS is not included in the shapefile metadata. In these situations, it is possible to add the missing information using the `st_set_crs()` function (assuming that we now the true CRS). The shapefile we are using has the geographical CRS WGS 84[6]. If we then wish to project the shapefile or, if projected, to re-project it using another CRS we can use the `st_transform()` function. Below there is an example on how to project our spatial data object to the Mercator CRS.

```
# Check the CRS
st_crs(africa)
```

```
## Coordinate Reference System: NA
```

```
# Set the CRS to WGS 84
africa <- st_set_crs(africa, 4326)

# Project using Mercator CRS (EPSG:3857)
africa_merc <- st_transform(africa, crs = 3857)
```

[5] Note that, by default, when plotting an `sf` object all the features contained in it will be visualised. To visualise only the geomtry use `plot(africa$geometry)`.

[6] Since this CRS is the most common it is useful to remember the correpsonding EPSG code 4326.

To save an `sf` object locally we can use the `st_write()` function. The first argument of the function will be the `sf` that we want to save. The second argument is a character string with the name of the file and the extension; `sf` will automatically select the driver to use based on the extension provided. If no file path is provided the spatial object will be saved in the working directory. Let's now save our africa spatial dataset as a shapefile with the right CRS info we added in our `data` folder with the name `africa_wgs.shp`.

```
st_write(africa, "data/africa_wgs.shp")
```

A lot of times spatial data is provided as a `.csv` file containing the coordinates as variables. In this case we will need to use the `st_af_sf()` function to convert our data set to a `sf` object. Your `data` folder contains a file called `lf.csv`. It contains georeferenced village level data about *lymphatic filariasis*[7] (LF) prevalence collected in Africa by different national programmes. We will load the `readr` package to read in the csv file and then we will convert it to a `sf` object and plot it.

[7] Considered globally as a neglected tropical disease (NTD), it is a parasitic disease caused by microscopic, thread-like worms.

```
# Load the readr package
library(readr)

# Read in the csv file
lf <- read_csv("data/lf.csv")

# Convert it to an sf object
lf_sp <- lf %>%
  st_as_sf(coords = c("Longitude", "Latitude"), crs = 4326)

# Plot the points
plot(lf_sp$geometry, cex = .05)
```

Note that we needed to specify two arguments in the `st_as_sf()` function. The first one `coords` wants the names of the columns where the coordinates are stored. With the `crs` argument we specify the CRS of the data. When

plotting the points we used the `cex` argument to regulate their size.

Importing and exporting raster data

Similar to vector data, raster data comes in many file formats with some of them supporting even multilayer files. The `raster()` command from the `raster` package reads in a single layer. In your `data` folder you will find a `.tif` file containing precipitation data for Africa. Let's load it to R and visualise it. Remember that you first need to install and load the `raster` package. To visualise our `raster` object we can use the standard `plot()`. However, to get a better output I suggest you to install the `rasterVis` package and use the `levelplot()` function.

```
# Load the raster package
library(raster)

# Read the raster containing rainfall data
precip <- raster("data/precipitation.tif")

# Let's have a look at his structure
precip
```

```
## class      : RasterLayer
## dimensions : 10143, 10663, 108154809  (nrow, ncol, ncell)
## resolution : 0.008333333, 0.008333333  (x, y)
## extent     : -25.35875, 63.49959, -46.97893, 37.54607  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## source     : /home/claudio/Dropbox/LSHTM/intro_to_R/data/precipitation.tif
## names      : precipitation
```

```
# Visualise it
plot(precip)

# Visualise it using rasterVis package
library(rasterVis)
levelplot(precip, margin = F)
```

To check the CRS of a raster you need to use the `crs()` function from the raster package. Let's see which CRS is our `raster` object using

```
# Check the CRS of the raster
crs(precip)
```

```
## CRS arguments:
##  +proj=longlat +datum=WGS84 +no_defs
## +ellps=WGS84 +towgs84=0,0,0
```

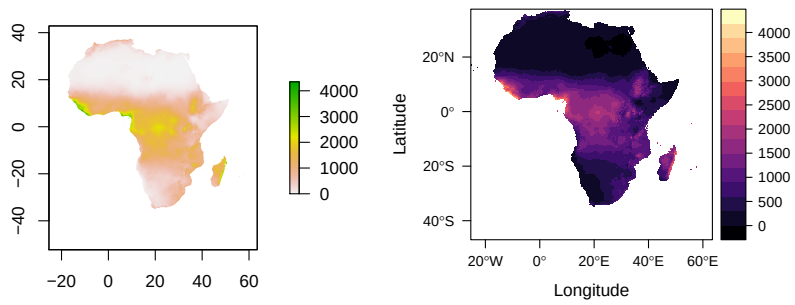


Figure 3: Outputs of the plot() function (left) and of the levelplot() function (right).

Note that with rasters the CRS is expressed as `proj4string`. If we look at the `proj` and `datum` arguments we see that the raster CRS is WGS 84. Since we want all of our spatial objects aligned we need to re-project it to Mercatori using the appropriate `proj4string`.

```
precip_merc <- projectRaster(precip, crs = crs("+init=epsg:3857"))
```

To write locally the re-projected raster we use the `writeRaster()` function:

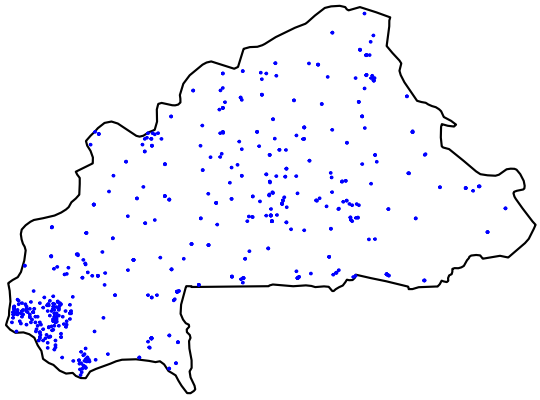
```
writeRaster(precip_merc, "data/precip_merc.tif")
```

Tidying spatial data

One of the nice features of `sf` objects is that they are *tidyverse friendly*. This means that great part of the `dplyr` functions can be used as well on `sf` objects. We will try some of them and see how they can help us during our analyses. Let's say that we are interested in Burkina Faso and we want to subeset both the Africa country boundaries and the LF points accordingly. We can use the `filter()` function for this task:

```
# Load the dplyr package
library(dplyr)

# Let's filter using the country column
burkina <- africa_merc %>%
  filter(country == "Burkina Faso")

lf_burkina <- lf_merc %>%
  filter(Country == "Burkina Faso")
```

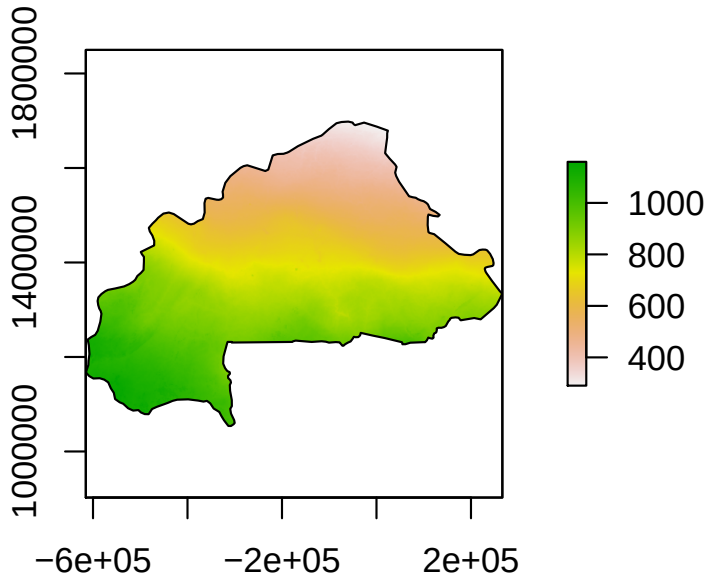
```r
# Let's plot the resulting map with points on top
plot(lf_burkina$geometry, add = T, cex = 0.1, col = "blue")
```



Spatial operations

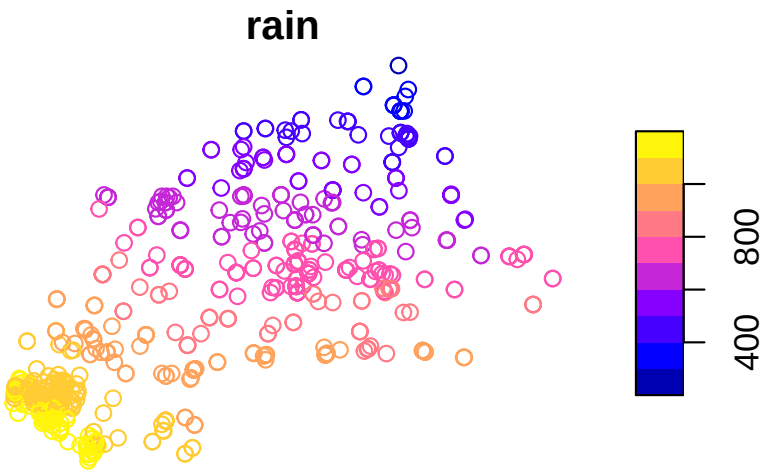
In this section we will show some of the most useful spatial operations to combine different types of spatial data. Immagine that we want to do a spatial analysis of LF in Burundi and we want to use precipitation as one of the variables during the analysis. We have a continent wide raster of precipitation but we want to crop it according tu our country boundaries and then extract the values at each of the observed village location. To do this we will use the `crop()` and `extract()` functions from the `raster` package.

```r
# Crop the precipitation raster according to Burkina faso boundaries
precip_burkina <- crop(precip_merc, burkina)
precip_burkina <- mask(precip_burkina, burkina)

# Visualise the result
plot(precip_burkina)
plot(burkina$geometry, add = T)
```



The `crop()` function retrieve the sptial extent of the `sf` object and crop the raster accordingly. If you want it to get only the values inside the country then you can use the `mask()` function. We will now extract values from the precipitation raster generatered at the locations where the villages in Burkina Faso were sampled. We will add them to our `lf_burkina` object generating a new variable.

```
# Extract values from raster using points
lf_burkina$rain <- extract(precip_burkina, lf_burkina)

# Visualise the reuslt
plot(lf_burkina["rain"])
```



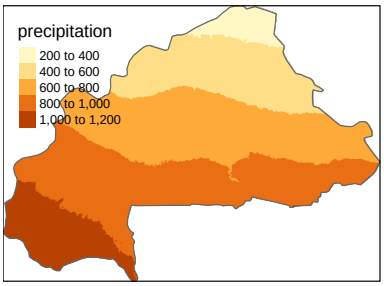
Create maps

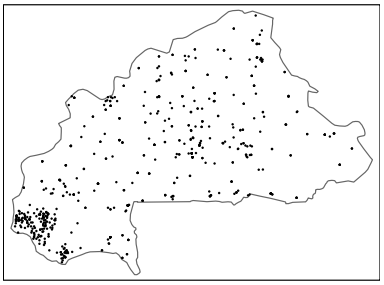
R provides a lof of functionalities to visualise spatial data and create very beautiful maps. Until now we have used basic plotting functions. Here we introduce the `tmap` package that allows to combine different types of geographic data in a map.

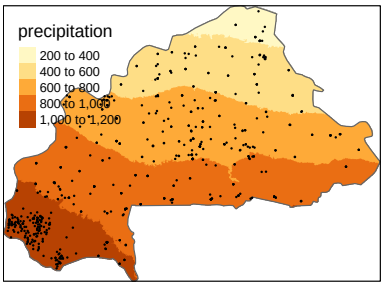
Like `ggplot2`, `tmap` is based on the idea of a "grammar of graphics". This involves a separation between the input data and the aesthetics (how data are visualised): each input dataset can be "mapped" in a range of different ways including location on the map (defined by data's geometry), color, and other visual variables. The basic building block is `tm_shape()` (which defines input data, raster and vector objects), followed by one or more layer elements such as `tm_fill()` and `tm_dots()`. This layering is demonstrated in the code below:

```
# Load the tmap package
library(tmap)

# Add fill layer to africa shape
tm_shape(africa_merc) +
  tm_fill() +

# Add border layer to africa shape
tm_shape(africa_merc) +
  tm_borders() +
# Add fill and border layers to nz shape
tm_shape(africa_merc) +
  tm_fill() +
  tm_borders()
```

This is an intuitive approach to map making: the common task of adding new layers is undertaken by the addition operator +, followed by `tm_*()`. The asterisk (*) refers to a wide range of layer types which have self-explanatory names including fill, borders (demonstrated above), bubbles, text and raster (see `help("tmap-element")` for a full list). Following this layering approach it is possible to combine different shapes together:

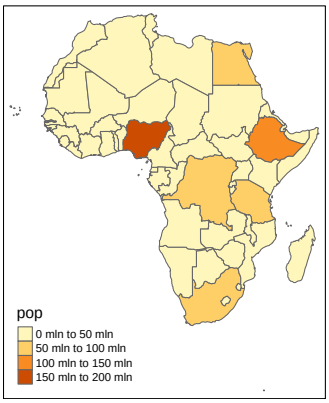






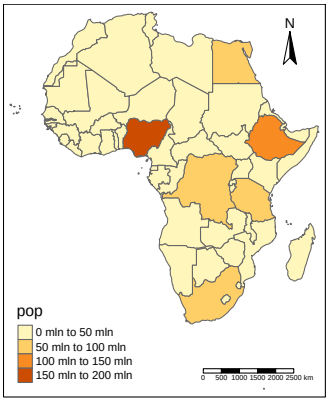
If the shape that we are plotting contain a variable that can be plottted the color of each layer can be visualised according to the values of that variable. Let's see an example with `tm_fill()`.

```
tm_shape(africa_merc) +
  tm_fill(col = "pop") +
  tm_borders()
```



Two important features that should never miss from a map are a legend with the scale of the map and a compass. With `tmap` this can be easility added:

```
tm_shape(africa_merc) +
  tm_fill(col = "pop") +
  tm_borders() +
tm_compass(position = c("right", "top")) +
tm_scale_bar()
```



Extra resources

The web is full of very nice resources related to R and spatial data. Here there is a list of my favourite references, they are either books or introudcitons to R packages for spatial analysis:

- Geocomputation with R
- Spatial Data Science
- `sf` webpage
- `sf` cheatsheet
- `tmap`: get started!