GIS And Remote Sensing For Ecologists

Emilio Berti

2024-08-07

Contents

Preface

I started learning GIS in 2017 during my PhD. As most of the programming I know, I learnt it by myself making many mistakes. I still make mistakes, which means I am still learning. My aim with this course is to be the guide I did not have when I started learning GIS and that would have saved me some of the mistakes. This course is intended for newcomers to GIS; experienced users should definitely look somewhere else (most likely on stackoverflow.com).

The course is divided into three main parts:

1. General introduction to GIS.
2. Fundamentals for GIS analysis.
3. GIS analysis with Google Earth Engine.

Software requirements

In the first part of this course, I will introduce the basic concepts of GIS using R and Python. For this, I will use the *terra* package for R and the *shapely* module for Python. I will focus more often on R than on python.

In the second part, I will cover the fundamentals of GIS analysis. I will explain the theory behind it and complement it with examples from actual analyses, introducing the most common workflows for R and Python. I will use the R package *terra* and the Python modules *shapely*, *geopandas*, *fiona*, and *plotly*.

In the third part, I will focus on Google Earth Engine and explain how to move heavy GIS analyses on the cloud. This will require a basic understanding of JavaScript and of Python object-oriented programming, as well as to interface with GEE using the *gee* Python module. This part may seem redundant at first (why should I move my computation on Google servers?), but, as your analyses expand in scale and need to access several databases, GEE shines as a great trade-off between complexity and effectiveness.

To install all packages and modules used in this book, you can use the conda environment management system. The environment file is at https://github.com/emilio-berti/gis-course/blob/master/conda/gis-course.yml.

```
conda env create --file=gis-course.yml
```

A history of GIS

Humans have always been interested in mapping out the Earth. Anaximander was among the first to publish a map of the known world in the 6th century BC.

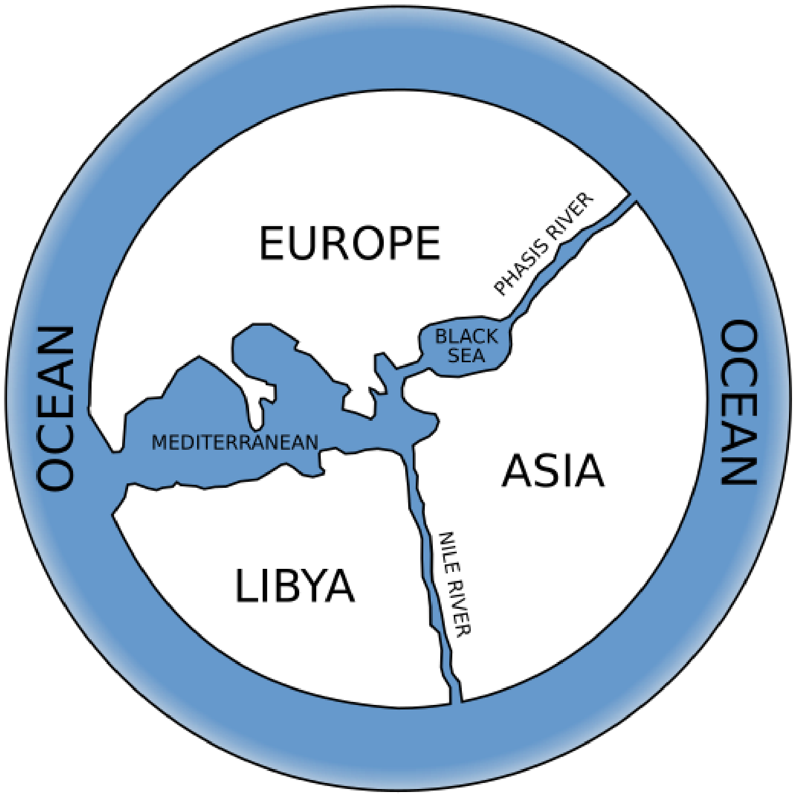


Figure 1: Reconstruction of the map of the known world from Anaximander.

It was not until the "Age of Exploration", where accurate maps could make the difference between riches and starving at sea, that such Tolkien-like style maps became more realistic.

Figure 2: A *portolan* map from the Age of Discovery.

Jump forward another 400 years and many major technological advancements and we get to the Information Age, where cartographic software started to be developed. GIS comprise both the hardware and the software to collect, store, manage, and analyze geographic data. Today, there are several GIS hardware and software that are publicly accessible and that are used in many fields: urban planning, climate modeling, mining, warfare, etc. It may be surprising for an ecologist, but current GIS tools are tightly linked to mineral and oil exploration, such as the European Petroleum Survey Group (EPSG), and to warfare, such as GPS and GLONASS. Nevertheless, we will use GIS for the well-being of humanity and stick to biogeographic analyses.

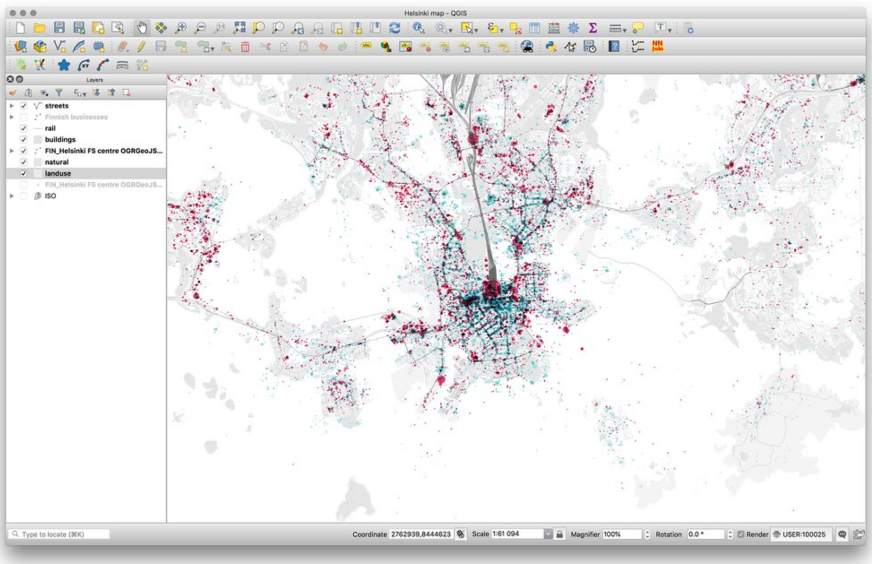


Figure 3: An example map produced using QGIS, a free GIS software.

Chapter 1

Projections

A map projection is a 2D representation of the Earth surface. There are several type of projections, each approximating Earth surface in different ways. Importantly, each spatial data must have an associated coordinate reference system (CRS), which defines how the 2D map and the Earth surface are related. In GIS courses, CRS is usually only briefly discussed or not talked at all. However, CRS is quite a complicated topic and one that can introduce many errors in your analysis: if you choose the wrong CRS, then your calculations can be extremely biased.

I will give only a brief introduction to CRS for two reasons:

1. They are a very complex topic and going into details takes a very long time.
2. Honestly, I am still figuring out some of the most obscure details of some CRS and I do not want to say incorrect things.

1.1 Geographic vs Projectes CRS

A geographic CRS is one where locations of points are described by longitude and latitude, i.e. the angle between the Prime meridian (an almost mythological creature) and the location and the angle between the Equator and the location, respectively. The most used geographic CRS is WGS84 (EPSG:4326).

A projected CRS is a system to represent the 3D Earth surface on a plane. Representing a 3D object into a 2D plane accurately is not possible. Therefore, projectons always distort a property of the Earth surface, in particular, at least one of: distance, angular conformity, and area. Projections can be grouped into types, depending on which property of the Earth surface they do not distort:

- Conformal projections: they correctly represent the angles between points and, thus, shapes. E.g., ESRI:54004 (Mercator).

- Equidistant: they correctly represent distances. E.g., ESRI:54002.
- Equal-area: they corretly represent areas. E.g. ESRI:54034.

ESRI stands for Environmental Systems Research Institute, Inc., which is the company that developed ArcGIS and created a code standard for projections. The other commonly used standard is maintained by the European Petroleum Survey Group (EPSG). For instance, World Mercator (conformal) is also EPSG:3395.

A common projected CRS is the Universal Transverse Mercator projection (conformal), e.g. EPSG:32632. We will talk more about this in a later chapter. You can find an overview of ESRI and EPSG projections at https://spatialreference.org/. Wikipedia also has a nice list with the property of each projection: https://en.wikipedia.org/wiki/List_of_map_projections

Chapter 2

Shapefiles

Shapefiles contain geospatial vector data, also called geometries. A shapefile has the extension *.shp* and it is usually accompanied by other files. Common supplementary files are:

- *.shx* = the index of the geometries in the data model.
- *.dbf* = the attributes of the geometries in a table format.
- *.prj* = the WKT representation of the coordinate reference system.

The core concept of shapefiles is that geographic features are stored using three fundamental geometry types:

1. Points, defined by coordinates (x, y).
2. Lines, defined by two points and a line interpolating them.
3. Polygons, defiend by several lines.

These three fundamental types can also be "stacked", e.g. one spatial feature can be represented by multiple points, lines, and polygons. In this case, we talk of *multipoints*, *multilines*, and *multipolygons*. The difference between multilines and polygons is that a polygon inscribe an area of space, that is it creates an "inside" area and an "outside" one, while multilines do not.

2.1 Geometries in R

Let's see an exampled in R using the package *terra*. Remember to load the package if you did not already: `library(terra)`. In *terra*, all the hustle that came with older GIS software has been removed, and geometries can be simply be created using the `vect()` function.

```
xy <- matrix(c(0, 0), ncol = 2)
poi <- vect(xy)
geomtype(poi)
```

```
## [1] "points"
```

This creates our first geometry, a point at the origin. To be meaningful, however, we must assign a coordinate reference system to our geometry. This can be done during the initialization itself.

```
poi <- vect(xy, crs = "EPSG:4326")
poi
```

```
##  class       : SpatVector
##  geometry    : points
##  dimensions  : 1, 0  (geometries, attributes)
##  extent      : 0, 0, 0, 0  (xmin, xmax, ymin, ymax)
##  coord. ref. : lon/lat WGS 84 (EPSG:4326)
```

Geometries can be converted into other types by *casting*. In *terra* this has been made extremely easy. Let's create some points.

```
xy <- matrix(seq_len(10), ncol = 2)
pois <- vect(xy, crs = "EPSG:4326")
geomtype(pois)
```

```
## [1] "points"
```

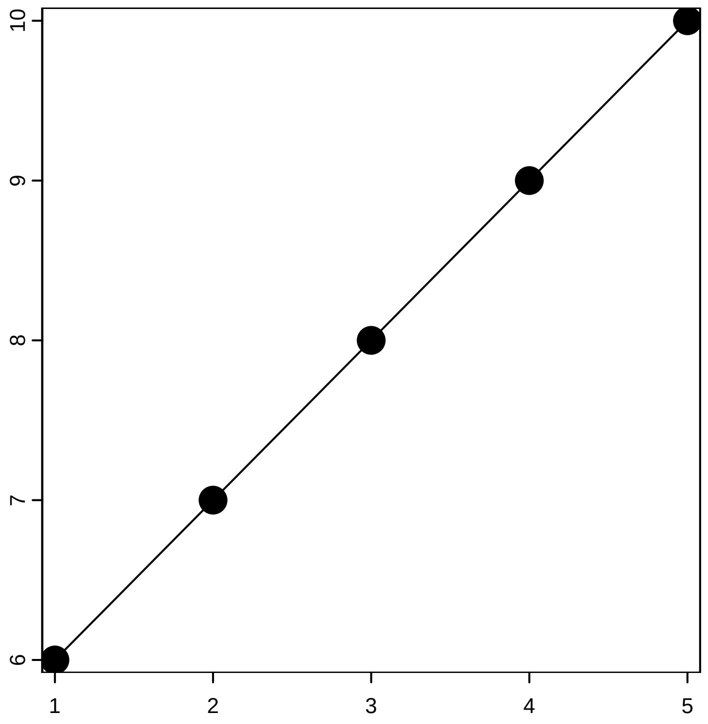
And cast them to lines using `as.lines()`.

```
ls <- as.lines(pois)
geomtype(ls)
```

```
## [1] "lines"
```

We may want to cast this to polygons, notice however that these lines do not inscribe an area of space, i.e. they are not "closed".

```
plot(ls)
points(pois, cex = 2)
```



Trying to cast these lines as polygons, will return an empty geometry.

```
pol = as.polygons(ls)
geomtype(pol)
```

```
## [1] "none"
```

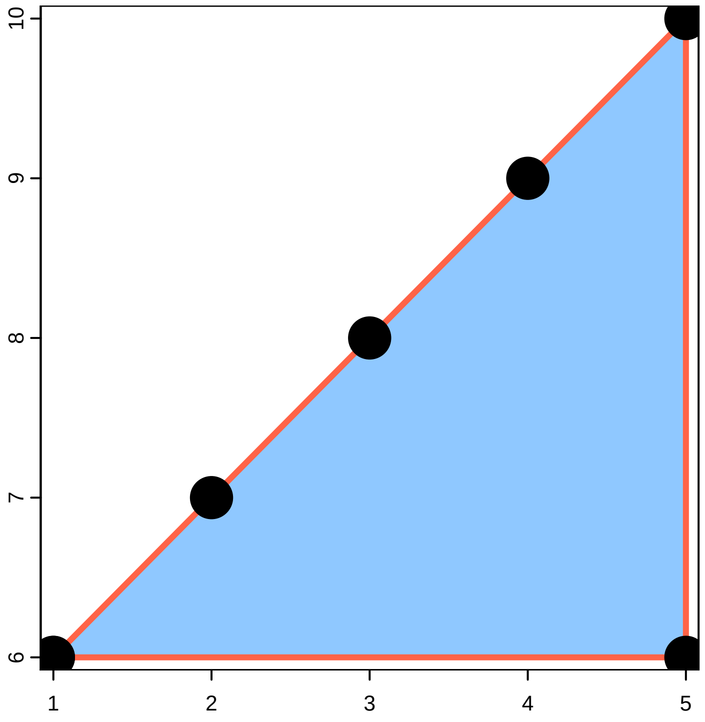
To cast lines to polygons, two points must be identical.

```
xy <- matrix(seq_len(10), ncol = 2)
xy <- rbind(xy, cbind(5, 6)) #add bottom-right point
xy <- rbind(xy, xy[1, ]) #add again first point
pois <- vect(xy, crs = "EPSG:4326")
ls <- as.lines(pois)
pol <- as.polygons(ls)
pol
```

```
##  class       : SpatVector
##  geometry    : polygons
##  dimensions  : 1, 0  (geometries, attributes)
##  extent      : 1, 5, 6, 10  (xmin, xmax, ymin, ymax)
##  coord. ref. : lon/lat WGS 84 (EPSG:4326)
```

```
# show it
plot(pol, col = "dodgerblue", alpha = .5)
```

```r
lines(ls, col = "tomato", lw = 3)
points(pois, cex = 3)
```



If you want to cast points to polygons, it is considered best practice to first cast them to lines and then cast these into polygons (as I just did). Trying to cast points to polygons directly can introduce errors, if it works at all.

Each geometry has its own set of properties. Points have coordinates:

```r
geom(pois)[, c("x", "y")] #coordinates of points
```

```
##      x  y
## [1,] 1  6
## [2,] 2  7
## [3,] 3  8
## [4,] 4  9
## [5,] 5 10
## [6,] 5  6
## [7,] 1  6
```

Lines have length:

```r
perim(ls) #length of line
```

```
## [1] 1509805
```

And polygons have perimeter and area:

```
perim(pol) #perimeter of polygon
```

```
## [1] 1509805
```

```
expanse(pol, unit = "km") #area of polygon
```

```
## [1] 97586.03
```

Note that lines and polygons also have coordinates, which are the breaking points of the straight lines, defined by the points:

```
geom(pol)
```

```
##      geom part x  y hole
## [1,]    1    1 1  6    0
## [2,]    1    1 2  7    0
## [3,]    1    1 3  8    0
## [4,]    1    1 4  9    0
## [5,]    1    1 5 10    0
## [6,]    1    1 5  6    0
## [7,]    1    1 1  6    0
```

2.2 Create geometry in R interactively

R *terra* has a nice feature that allows to draw on a plot and save the resulting geometry in the environment. This is achieved using the function `draw()`, which takes as argument the type of geometry you want to draw (*points*, *line*, *polygon*, or *extent*).

```
plot(pol, col = "dodgerblue3", alpha = .5)
_drawing = draw("lines")
```

And then just left-click on the map. When you are done, right click and the output will be saved in the variable *drawing*.

This is very useful when you want to zoom on a region of a map or draw coarse polygons, but don't want to open a more interactive GIS software, such as QGIS.

Chapter 3

Rasters

Because rasters are fundamentally 2D matrices with matadata, it is useful to understand what is a matrix before jumping straight into the topic. A 2D matrix A is a mathematical object that can be represented as a grid. The elements A_{ij} of the matrix A are called its entries. The subscript in A_{ij} indicate the row and column coordinates of the entries. For example, $A_{1,1}$ is the element in first row and first column, while $A_{3,4}$ is the element in the third row and fourth column.

3.1 Matrices in R

R provides a native support for matrices. In fact, R was designed to work with matrices, as most statistical models can be represented as operation on matrices.

```
vals <- seq_len(9)
A <- matrix(vals, byrow = TRUE, nrow = 3, ncol = 3)
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

When creating a matrix, there are two import things that are often neglected. First, by specifying `byrow = TRUE`, we make sure that the values contained in `vals` are inserted into the matrix row-wise, i.e. $A_{1,1} =$ `vals[1]`, $A_{1,2} =$ `vals[2]`, etc. This argument can be omitted if inserting elements column-wise is preferred; in this case, $A_{2,1} =$ `vals[2]`, etc..

Second, at least one of `ncol` or `nrow` should be specified, or the matrix will have only one column (column vectors). If you specify only one of the two, R will infer the other dimension. You can specify both of them and I invite you to do so for clarity, whenever possible.

3.2 Rasters

A raster is basically a 2D matrix with associated metadata. The most important metadata define:

- The spatial *extent*.
- The spatial *resolution*.
- The coordinate reference system, *crs*.

A raster without these three metadata is, in most cases, useless.

3.3 Rasters in R

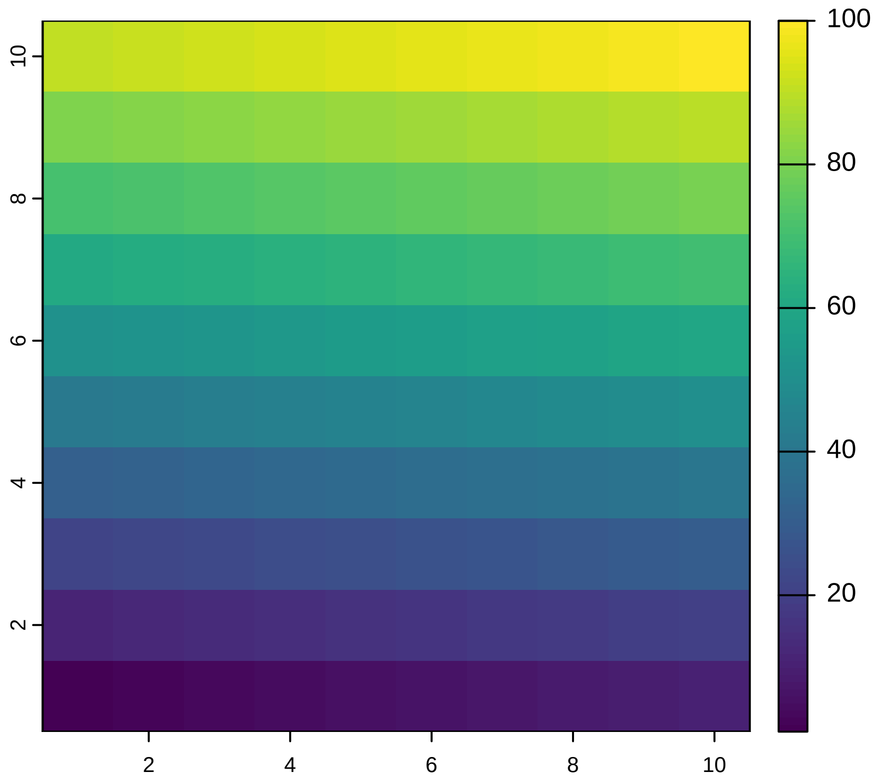
In R, *terra* provides all you need to work with rasters. To create a raster, you can use the `rast()` function. You can use as input a matrix.

```r
r <- rast(A, crs = "EPSG:4326")
r
```

```
## class       : SpatRaster
## dimensions  : 3, 3, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0, 3, 0, 3  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        : lyr.1
## min value   :     1
## max value   :     9
```

Or a dataframe.

```r
d <- data.frame(
  x = rep(seq_len(10), 10), #x coord
  y = rep(seq_len(10), each = 10), #y coord
  z = seq_len(100) #values
)
r <- rast(d, crs = "EPSG:4326")
plot(r)
```



`rast()` is also used to read rasters from files.

```r
r <- rast("some-file.tif")
```

To write rasters to files, the `writeRaster()` function is used.

```r
writeRaster(r, "some-file.tif", overwrite = TRUE, datatype = "INT4U")
```

The `overwrite` and `datatype` arguments are optional, but I strongly recommend you to always include them. `overwrite = TRUE` simply make sure that, if the raster already exists on disk, it is overwritten. *terra*, by defaults, throw an error if the file exists and `overwrite = FALSE`, which can be annoying when working on servers. `datatype` specifies the numeric type of the matrix elements. Available types are *signed* and *unsigned integer* and *float*. You also have to specify the number of bytes to use for each entry. Available dataype choices are:

- INT1U, unsigned integer of 1 byte: $\in [0, 2^8]$.
- INT2U, unsigned integer of 2 bytes: $\in [0, 2^{16}]$.
- INT2S, signed integer of 2 bytes: $\in [-2^8 + 1, 2^8]$.
- INT4U, unsigned integer of 4 bytes: $\in [0, 2^{32}]$.
- INT4S, signed integer of 4 bytes: $\in [-2^{16} + 1, 2^{16}]$.
- FLT4S, signed floating point (real) number of 4 bytes: $\in [1.2 \cdot 10^{-38}, 3.4 \cdot 10^{38}]$

- FLT8S, signed floating point (real) number of 8 bytes: $\in [2.3 \cdot 10^{-308}, 1.7 \cdot 10^{308}]$

You can actually store numbers also outside the range of the numeric types, but this will come to the cost of precision, so you should try to avoid going outside ranges. This does not mean that you should always save files as *FLT8S*, as this will require more disk space and slow down writing/reading operations. For example, if you want to store the elevation at global scale with a precision of one meter, *INT2U* is the optimal choice, as it is the smallest type that can store the information.

3.4 Raster manipulation

Rasters can be manipulated in the same way you would do with a matrix. For example:

```
r + 10 #add a constant
```

```
## class       : SpatRaster
## dimensions  : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        :   z
## min value   :  11
## max value   : 110
```

```
2 * r #multiply by a constant
```

```
## class       : SpatRaster
## dimensions  : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        :   z
## min value   :   2
## max value   : 200
```

```
r ^ 2 #power
```

```
## class       : SpatRaster
## dimensions  : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        :     z
## min value   :     1
## max value   : 10000
```

```
2 * r + 10 #combined addition and multiplication
```

```
## class       : SpatRaster
## dimensions  : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        :   z
## min value   :  12
## max value   : 210
```

You can also add and multiply rasters together:

```
r + r
```

```
## class       : SpatRaster
## dimensions  : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        :   z
## min value   :   2
## max value   : 200
```

```
r * r
```

```
## class       : SpatRaster
## dimensions  : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        :     z
## min value   :     1
## max value   : 10000
```

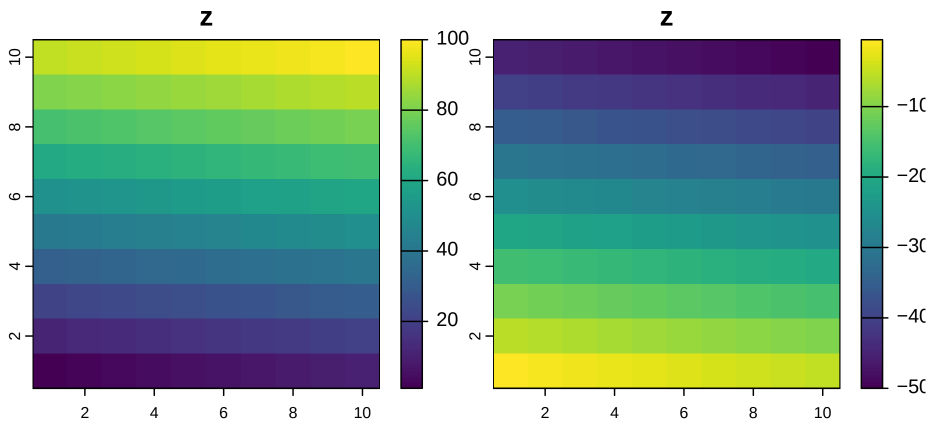
3.5 Raster stacks

You can also stack rasters to create 3D stacks. In *terra*, this is achieved simply by appending the rasters together:

```
s <- c(r, -r / 2)
s
```

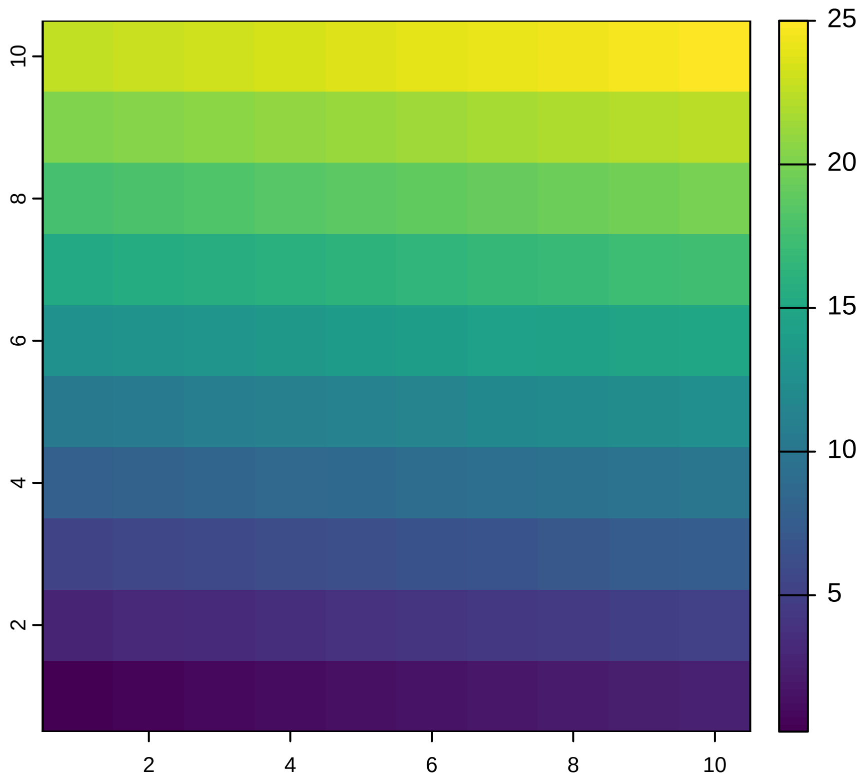
```
## class       : SpatRaster
## dimensions  : 10, 10, 2  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## names       :   z,     z
## min values  :   1, -50.0
## max values  : 100,  -0.5
```

```
plot(s)
```



Stacks are quite useful because you can call vectorized functions on them, e.g.:

```
plot(mean(s)) #mean
```



```
stdev(s)^2 #variance
```

```
## class       : SpatRaster
## dimensions  : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        :       std
## min value   :    0.5625
## max value   : 5625.0000
```

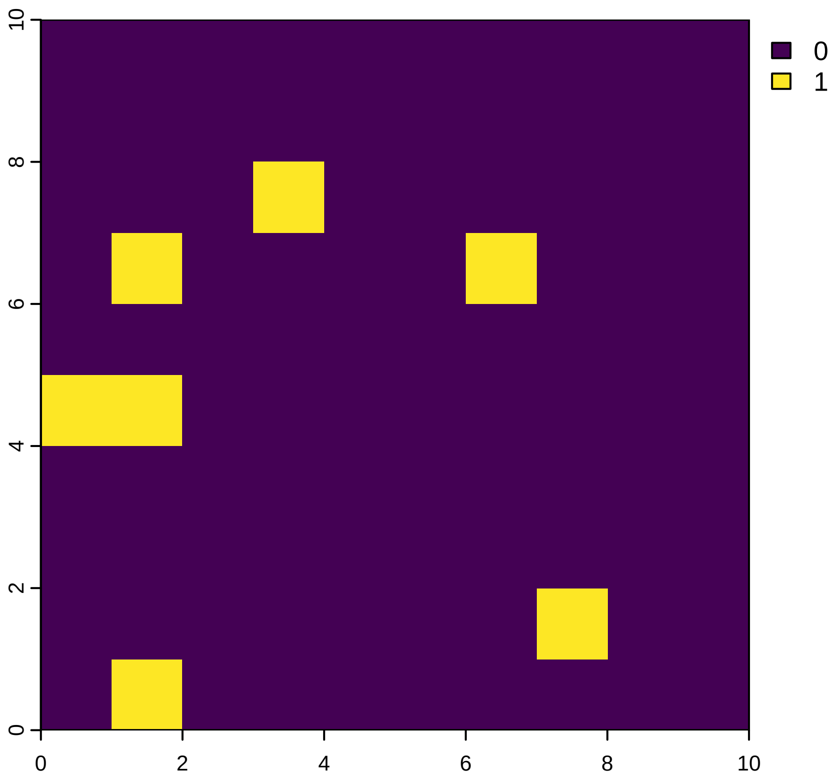
```
prod(s) #product
```

```
## class       : SpatRaster
## dimensions  : 10, 10, 1  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## name        :   prod
## min value   : -5e+03
## max value   : -5e-01
```

3.6 Example: stacks for ecology

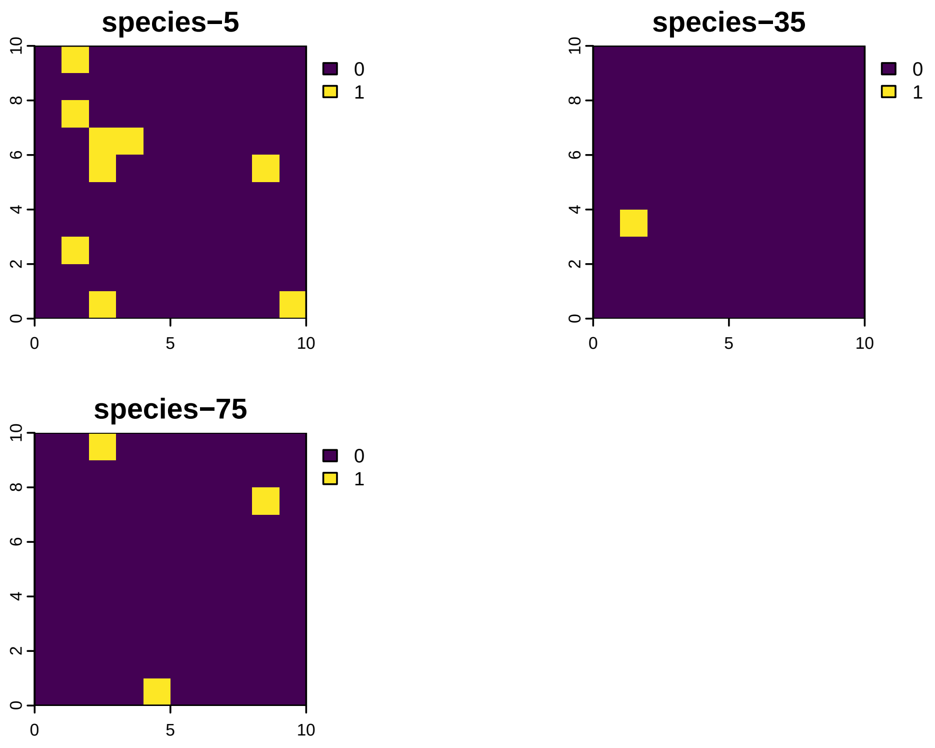
Imagine you sampled the occurrences for 100 species and want to know the number of species detected in each cell. The distribution of one species is saved in the file *species-1.tif* and may look like this:

```
r <- rast("species-1.tif")
plot(r)
```



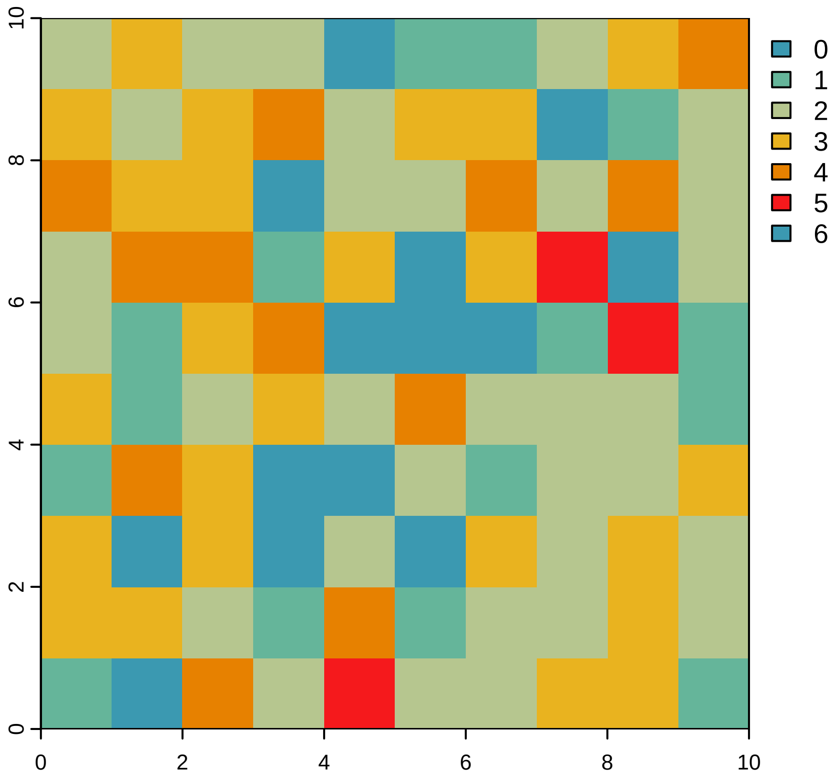
You can load all of them at once:

```
files <- list.files("data/", pattern = "species-")
s <- rast(files)
plot(s[[c(5, 35, 75)]]) #show species 5, 35, and 75
```



To get the number of species detected in each cell, you simply need to sum across the stack:

```r
S <- sum(s)
plot(S, col = hcl.colors(minmax(S)["max", ], "Zissou 1"))
```


0
2
4
6
8
10
0
1
2
3
4
5
6

Chapter 4

Basic Operations

There are some operations that are the staple of GIS. For example: What is the distance between two features? What is the area of a polygon? What is the value of a raster at specific locations?

In the GitHub repository, I put a *data* folder that contains some data we can use.

Chapter 5

Vector operations

5.1 Loading vectors

Load the vector shapefile of EU countries:

```
eu <- vect("data/EU/EU.shp")
eu
```

```
## class       : SpatVector
## geometry    : polygons
## dimensions  : 27, 11  (geometries, attributes)
## extent      : -10.38743, 34.39352, 34.57181, 70.0864  (xmin, xmax, ymin, ymax)
## source      : EU.shp
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## names       : CNTR_ID NAME_ENGL NAME_FREN ISO3_CODE         SVRG_UN     CAPT
## type        :   <chr>     <chr>     <chr>     <chr>           <chr>    <chr>
## values      :      AT   Austria  Autriche       AUT UN Member State   Vienna
##                    BE   Belgium  Belgique       BEL UN Member State Brussels
##                    BG  Bulgaria  Bulgarie       BGR UN Member State    Sofia
## EU_STAT EFTA_STAT CC_STAT  NAME_GERM   FID
##   <chr>     <chr>   <chr>      <chr> <chr>
##       T         F       F Österreich    AT
##       T         F       F    Belgien    BE
##       T         F       F  Bulgarien    BG
```

This vector contains the polygons of 27 countries in the European Union (EU). Additionally, an attribute table is attached to it, providing information about the country name, the ISO3 code, etc. You can see full list of attributes using `names(eu)`:

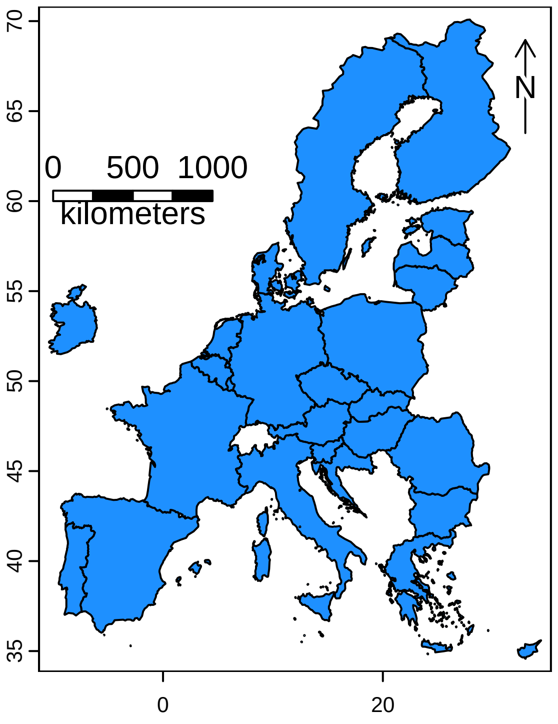
```
names(eu)
```

```
##  [1] "CNTR_ID"   "NAME_ENGL" "NAME_FREN" "ISO3_CODE" "SVRG_UN"   "CAPT"
##  [7] "EU_STAT"   "EFTA_STAT" "CC_STAT"   "NAME_GERM" "FID"
```

5.2 Plotting vectors

Let's put it on a map:

```
plot(eu, col = "dodgerblue")
north() #show the North
sbar(1e3, type = "bar", divs = 4, below = "kilometers", xy = c(-10, 60)) #show a scale
```



If you are familiar with maps, you may recognize that this is a longitude-latitude projection: the axes represent arc-degrees. To be sure, we check the CRS of the vector:

```
crs(eu, proj = TRUE)
```

```
## [1] "+proj=longlat +datum=WGS84 +no_defs"
```

As expected, this is a *longlat* CRS. The argument `proj = TRUE` specifies that we want a PROJ-string output. This is more readable than the standard output of `terra`:

```
crs(eu, proj = FALSE)
```

```
## [1] "GEOGCRS[\"WGS 84\",\n    DATUM[\"World Geodetic System 1984\",\n        ELLIPSOID[\"WGS 8
```

For some cases, for instance calculating areas of polygons, *longlat* CRS are recommended.

5.3 Area extent

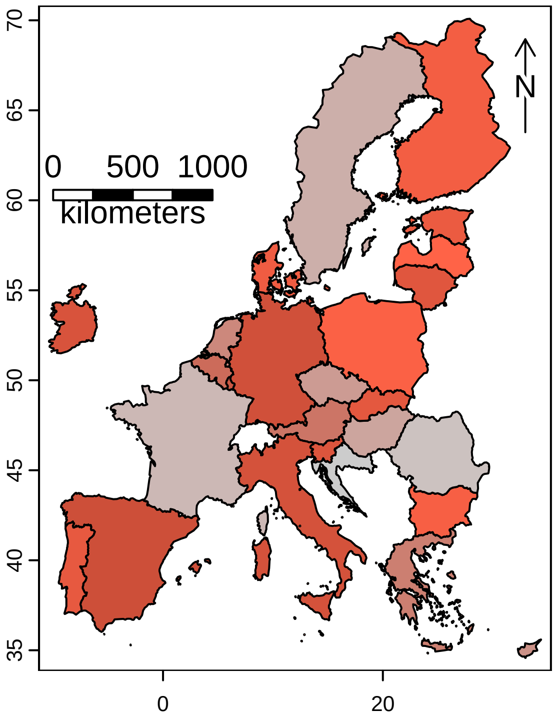
In `terra`, the function `expanse()` calculate areas. If you read the documentation, it also specified that longlat CRS is the most accurate to calculate areas. Let's calcualte the area of each country and add it to the attribute table of the vector:

```
A <- expanse(eu, unit = "km")
eu$area <- A
eu[, c("NAME_ENGL", "area")]
```

```
##  class       : SpatVector
##  geometry    : polygons
##  dimensions  : 27, 2  (geometries, attributes)
##  extent      : -10.38743, 34.39352, 34.57181, 70.0864  (xmin, xmax, ymin, ymax)
##  source      : EU.shp
##  coord. ref. : lon/lat WGS 84 (EPSG:4326)
##  names       : NAME_ENGL      area
##  type        :     <chr>     <num>
##  values      :   Austria 8.386e+04
##                  Belgium  3.07e+04
##                 Bulgaria 1.111e+05
```

We can now make a map with polygons filled with colors representing the county area:

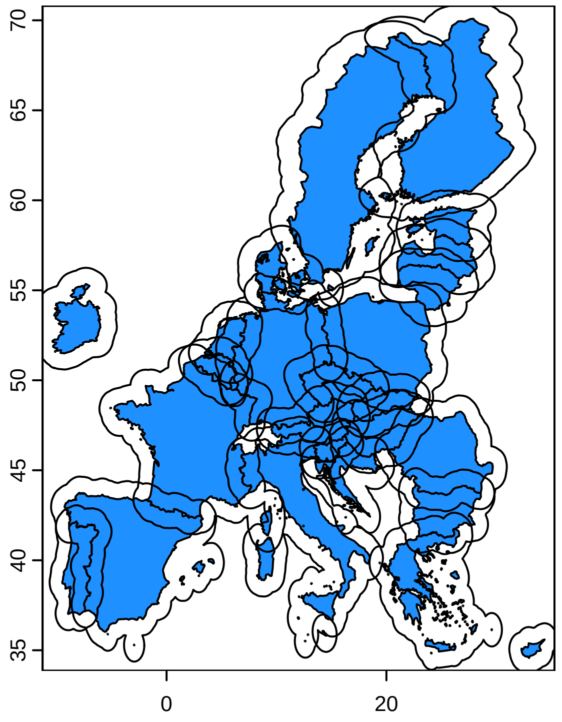
```
# a gradient palette with number of colors = countries
pal <- colorRampPalette(c("grey80", "tomato3", "tomato"))(nrow(eu))
pal <- rev(pal[order(eu$area)]) #arrange colors
plot(eu, col = pal) #adding a legend to vector maps is not easy and will leave it for another cha
north()
sbar(1e3, type = "bar", divs = 4, below = "kilometers", xy = c(-10, 60))
```



5.4 Buffers and neighbours of a polygon

Our shapefile is a good shapefile, meaning that the people who created it took care that the boundaries of countries did not overlap. Indeed, this would be a reason for tension between countries. However, we can still trick our way to obtain the neighbors of countries. Imagine to enlarge the countries, i.e. to expand their borders by a small amount, let's say 1km. This opearation is called buffering:

```r
b <- buffer(eu, 1e5) #100km for visualizing it
plot(eu, col = "dodgerblue")
lines(b)
```



If done on one country only, then this will overlap with its adjacent neighbors (and with non-adjacent ones if they are closer than 1km). We can use this together with `terra::intersect()` to get its neighbors:

```r
neigh_name <- as.list(rep(NA, nrow(eu)))
neigh_area <- as.list(rep(NA, nrow(eu)))
names(neigh_name) <- eu$NAME_ENGL
names(neigh_area) <- eu$NAME_ENGL

# iterate over countries
for (i in seq_len(nrow(eu))) {
  b <- buffer(eu[i], 1e3)
  ans <- intersect(eu[-i], b)
  neigh_name[[i]] <- ans$NAME_ENGL
  neigh_area[[i]] <- ans$area
}
```

```
## Warning: [intersect] no intersection
## Warning: [intersect] no intersection
## Warning: [intersect] no intersection
```

Some warning are thrown telling use that some countries do not have neighbors:

```r
names(neigh_name)[sapply(neigh_name, length) == 0]
```

```
## [1] "Cyprus"  "Ireland" "Malta"
```

These are all islands (Cyprus, Ireland, and Malta), so that is correct.

Importantly, `terra` has a special function to get adjanceny polygons, called (surprisingly) `adjacent()`. This is the funciton you should normally use; the approach used above is for teaching purposes. However, note that, in this case, the two approach give the same results:

```r
adj <- adjacent(eu)
neigh_name_correct <- as.list(rep(NA, nrow(eu)))
names(neigh_name_correct) <- eu$NAME_ENGL
for (i in seq_along(neigh_name_correct)) {
  neigh_name_correct[[i]] <- eu$NAME_ENGL[adj[adj[, 1] == i, 2]]
}

compare <- data.frame(
  buffering = unlist(neigh_name),
  correct = unlist(neigh_name_correct)
)
any(compare[, 1] != compare[, 2])
```

```
## [1] FALSE
```

```r
head(compare)
```

```
##          buffering  correct
## Austria1   Czechia  Czechia
## Austria2   Germany  Germany
## Austria3   Hungary  Hungary
## Austria4     Italy    Italy
## Austria5  Slovenia Slovenia
## Austria6  Slovakia Slovakia
```

The reason they are identical is because there is no country close 1km or less to another and that is also not an adjacent neighbor.

Chapter 6

Mapping

One of the most common goal of GIS is to produce a map of a phenomenon or process.

As example, we will use data from the Italian National Insitute of Statistics (ISTAT):

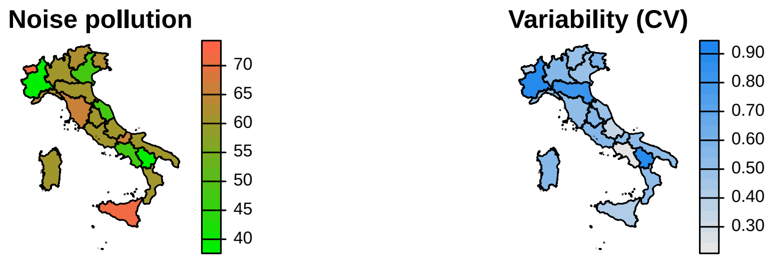
1. Administrative division shapefile (https://www.istat.it/it/archivio/222527).
2. GDP of each of the regions, the second administrative division after the State.

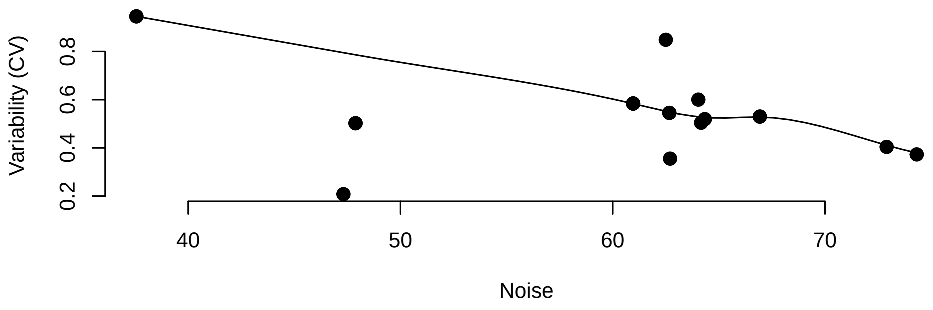
Both of these are in the *data/ISTAT/* folder, in the respective subdirectories.

Load the vector shapefile of EU countries:

```r
library(terra)

# shapefile of regions
regs <- vect("data/ISTAT/Limiti01012024_g/Reg01012024_g/Reg01012024_g_WGS84.shp")

# shapefile of cities
cities <- vect("data/ISTAT/Limiti01012024_g/Com01012024_g/Com01012024_g_WGS84.shp")

# noise pollution
noise <- read.csv("data/ISTAT/city-acoustic-noise.csv")
noise <- noise[noise$TIME_PERIOD == 2012, ] #only 2012
noise <- noise[noise$TIME_PERIOD != "IT", ] #only city level
noise <- noise[noise$TYPE_OF_MONITORING == 1, ] #sensor type
noise <- noise[!is.na(noise$OBS_VALUE), ] #remove empty
noise <- noise[noise$DATA_TYPE == "MON_LIM", ] #monthly limit

# not all cities studies: remove not studied
```

```
studied <- intersect(cities$PRO_COM_T, noise$REF_AREA)
cities <- cities[cities$PRO_COM_T %in% studied]
noise <- noise[noise$REF_AREA %in% studied, ]

# add attribute
cities$noise <- noise$OBS_VALUE[sapply(noise$REF_AREA, \(x) which(cities$PRO_COM_T ==
cities <- cities[order(cities$noise)]
cities <- cities[cities$noise > 0] #suspect data

# map -----------
noise_vals <- unique(cities$noise)
colors <- colorRampPalette(c("green3", "tomato"))(length(noise_vals))
pal <- c()
for ( x in noise_vals ) {
  pal <- c(pal, rep(colors[which(x == noise_vals)], sum(cities$noise == x)))
}

#plot(regs)
#plot(cities, col = as.numeric(cities$noise), add = TRUE)

# regional statistics ----------
regs$noise <- NA
regs$noise_cv <- NA
for ( i in seq_along(regs) ) {
  within <- relate(cities, regs[i], "within")
  ids <- which(within, arr.ind = TRUE)[, "row"]
  regs$noise[ids] <- mean(cities$noise[ids])
  regs$noise_cv[ids] <-  sd(cities$noise[ids]) / mean(cities$noise[ids])
}
regs$noise <- as.numeric(regs$noise)
regs$noise_cv <- as.numeric(regs$noise_cv)

layout(matrix(c(2, 3, 1, 1), byrow = TRUE, ncol = 2))
par(mar = c(4, 4, 2, 2))
scatter.smooth(
  regs$noise, regs$noise_cv,
  pch = 20, cex = 2, frame = FALSE,
  xlab = "Noise",
  ylab = "Variability (CV)"
)
plot(
  regs,
  "noise",
  col = colorRampPalette(c("green2", "tomato"))(20),
  type = "continuous",
  axes = FALSE,
  main = "Noise pollution"
)
plot(
  regs,
  "noise_cv",
  col = colorRampPalette(c("grey90", "dodgerblue2"))(20),
  type = "continuous",
  axes = FALSE,
  main = "Variability (CV)"
)
```


Noise pollution
70
65
60
55
50
45
40
Variability (CV)
0.90
0.80
0.70
0.60
0.50
0.40
0.30


Variability (CV)
Noise