GIS And Remote Sensing For Ecologists

Emilio Berti

2024-08-21

Contents

Preface

I started learning GIS in 2017 during my PhD. I first approached GIS using ArcGIS, which I hated with all my guts. As I prefer typing on terminal rather than clicking on icons, I quickly switched to R. I will use the R package `terra` thorough this course. You can replicate most of the content of this course using python, but I will leave that for another time. 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.

In the first part of this course, I will introduce the basic concepts of GIS using R. 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 in R. 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. It would be great to talk about how to interface with Google Earth Engine using python, but this topic is probably too advanced and will be the focus of another course.

Software requirements

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
```

The why 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.

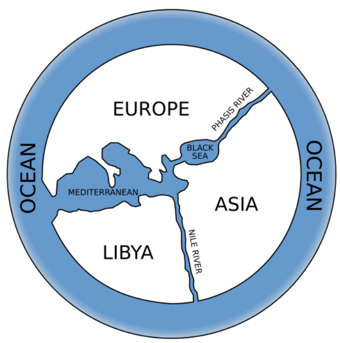


Figure 1: Reconstruction of the map of the known world from Anaximander.

Maps were not particularly realistic, as people did not really use them the way we do now. Looking at maps during battles, for examples, is a very modern thing; ancient people simply did not do it. It was not until the "Age of Exploration", where accurate maps could make the difference between riches and starving at sea, that maps became more realistic.

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

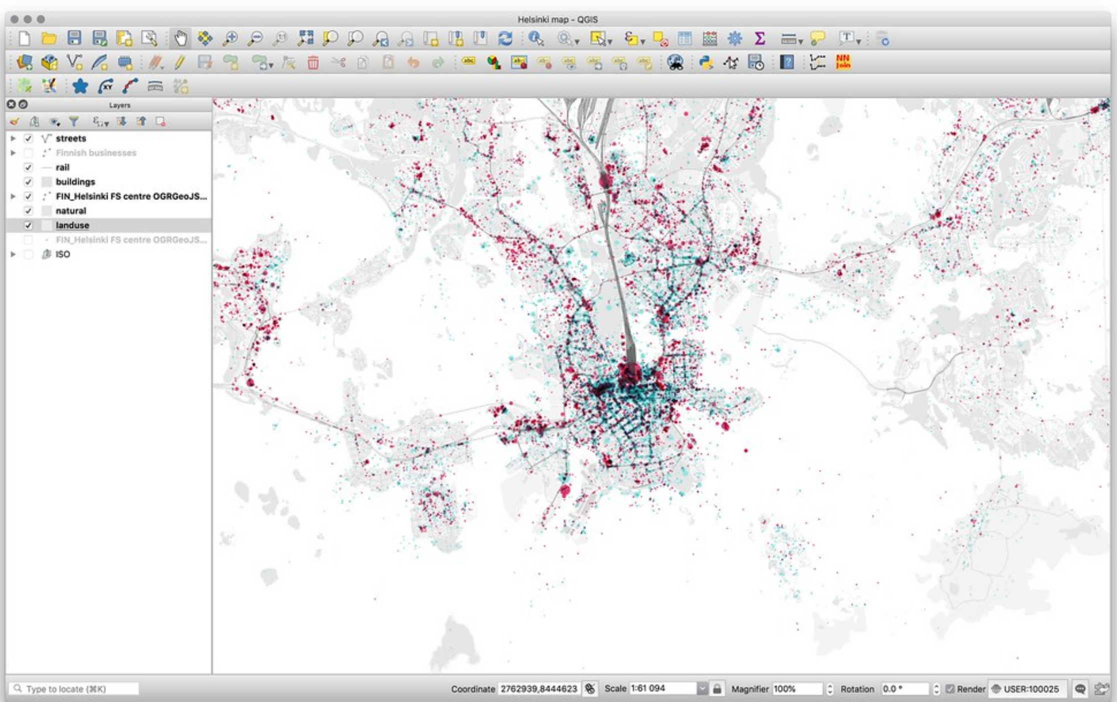


Figure 3: An example map produced using QGIS, a free GIS software.

Chapter 1

Projections

A map projection is a 2D representation of the Earth surface. Technically, a projection is a set of transformations to represent the surface of the Earth on a plane.

Figure 1.1: The Earth as a sphere. Black lines show the coastline and grey lines the meridians (longitude) and paralleles (latitude). If you walk along a parallel or a meridian, you will be going on straight line, but on a sphere they are curved.

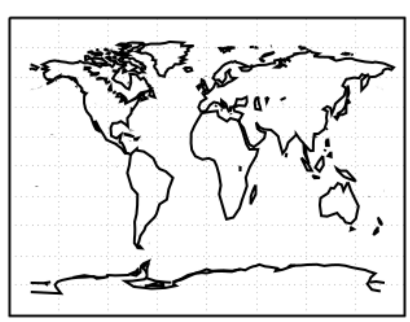


Figure 1.2: The Earth on a plane. Black lines show the coastline and grey lines the meridians (longitude) and paralleles (latitude). You can see something has been distorted.

There are several type of projections, each approximating the surface of the Earth in different ways. 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. However, choosing an inappropriate CRS for your analysis can introduce many errors.

1.1 Geographic CRS

A geographic CRS is one where locations of points are described by longitude and latitude, i.e. the angle between the Prime meridian (oh, the mythological creature) and the location and the angle between the Equator and the location, respectively. The most used geographic CRS is WGS84 (EPSG:4326). Of the many ways to encode the information of a CRS in computers, the easiest for humans to read is the PROJ.4 standard. The PROJ.4 of EPSG:4326 is: `+proj=longlat +datum=WGS84 +no_defs +type=crs`. In `terra`, the function `crs(x, proj = TRUE)` return the PROJ notation of `x`, which can be a spatial object or also a registered CRS code:

```
crs("EPSG:4326", proj = TRUE)
```

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

Note that `proj=longlat` tells us this is a geographic CRS.

Setting `proj = FALSE` (default), returns the WKT (well-known text) representation:

```
cat(crs("EPSG:4326"))  # cat to show it nicely
```

```
## GEOGCRS["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]],
##     CS[ellipsoidal,2],
##         AXIS["geodetic latitude (Lat)",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["geodetic longitude (Lon)",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     USAGE[
##         SCOPE["Horizontal component of 3D system."],
##         AREA["World."],
##         BBOX[-90,-180,90,180]],
##     ID["EPSG",4326]]
```

Note that this tells us also important information, for instance the area and scope of the CRS. For instance, look at this CRS for the Philippines:

```
cat(crs("EPSG:3124"))
```

```
## PROJCRS["PRS92 / Philippines zone 4",
##     BASEGEOGCRS["PRS92",
##         DATUM["Philippine Reference System 1992",
##             ELLIPSOID["Clarke 1866",6378206.4,294.978698213898,
##                 LENGTHUNIT["metre",1]]],
##         PRIMEM["Greenwich",0,
##             ANGLEUNIT["degree",0.0174532925199433]],
##         ID["EPSG",4683]],
##     CONVERSION["Philippines zone IV",
##         METHOD["Transverse Mercator",
##             ID["EPSG",9807]],
##         PARAMETER["Latitude of natural origin",0,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8801]],
##         PARAMETER["Longitude of natural origin",123,
##             ANGLEUNIT["degree",0.0174532925199433],
##             ID["EPSG",8802]],
##         PARAMETER["Scale factor at natural origin",0.99995,
##             SCALEUNIT["unity",1],
##             ID["EPSG",8805]],
##         PARAMETER["False easting",500000,
##             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["Engineering survey, topographic mapping."],
##         AREA["Philippines - approximately between 122°E and 124°E onshore and offshore - southeast
##         BBOX[3.44,121.74,22.18,124.29]],
##     ID["EPSG",3124]]
```

which specifies the area (`AREA`) and extent (`BBOX`) for which this CRS should be used. Check these attributes before using a CRS: you may find out that you are using a CRS developed to have minimal distortion (more about this in the next section) for a part of the world that is not of interest, while it may have a lot of distortion for your region of interest.

1.2 Projected CRS

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, projections 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 – 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 correctly represent areas. E.g. ESRI:54034.

```
crs("ESRI:54004", proj = TRUE)
```

```
## [1] "+proj=merc +lat_ts=0 +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs"
```

`proj=merc` is a Mercator projection.

```
crs("ESRI:54002", proj = TRUE)
```

```
## [1] "+proj=eqc +lat_ts=60 +lat_0=0 +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs"
```

`proj=eqc` is an equidistant cylindrical projection.

```
crs("ESRI:54034", proj = TRUE)
```

```
## [1] "+proj=cea +lat_ts=0 +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs"
```

`proj=cea` is a cylindrical equal-area projection.

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. At proj.org there is a good list of projections and their attributes: https://proj.org/en/9.4/operations/projections/index.html. You can find an overview of ESRI and EPSG projections also 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

Vectors or Geometries

Vectors (or 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 connecting 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

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"
```

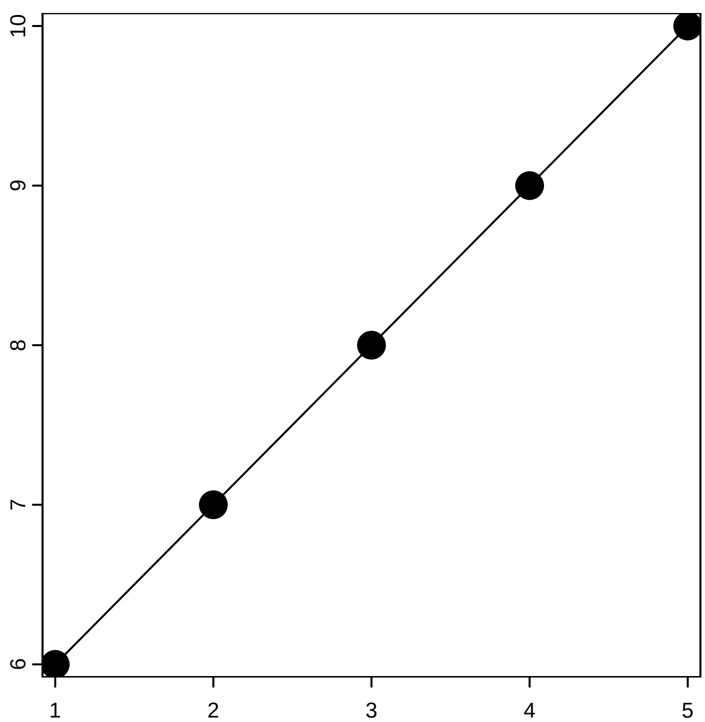
And cast them to lines using `as.lines()`.

```
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.

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

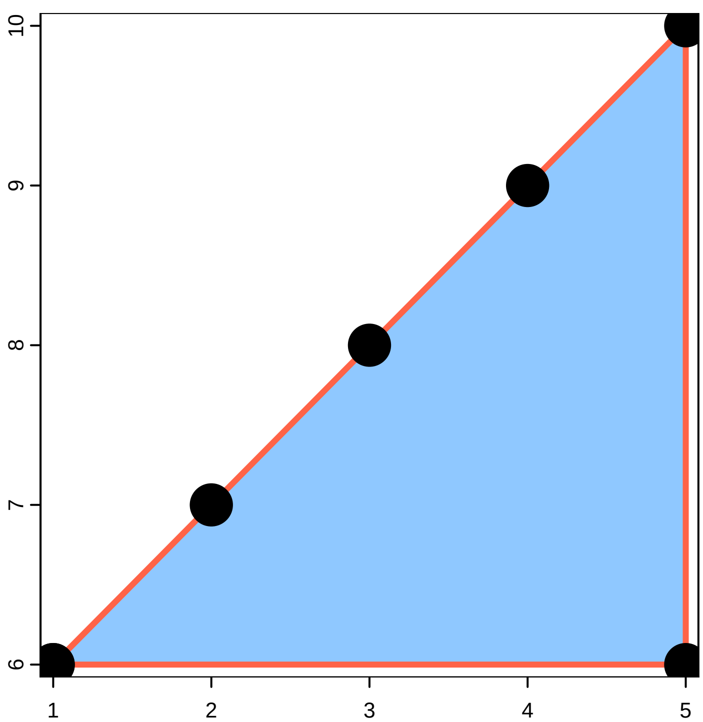
```
## [1] "none"
```

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

```r
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)
```

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

```
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:

```
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

A nice feature of `terra` is that it 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.

2.3 Reading and writing shapefiles

The function `vect()` is also used to read shapefiles:

```
vect(<filename>)
```

To write shapefiles, use `writeVector()`.

```
writeVector(<spatVect>, <filename>)
```

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 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 `byrow = TRUE`, 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. The arguments `ncol` and `nrow` specify the dimension of the matrix. 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 and to avoid guessing.

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*.

The extent tells us what is the area covered by the raster, i.e. the coordinates of its four angles. The resolution specifies how big a cell pixel is. The CRS defines the projection (if any) of the raster. A raster without these three metadata is, in most cases, useless.

3.3 Rasters in R

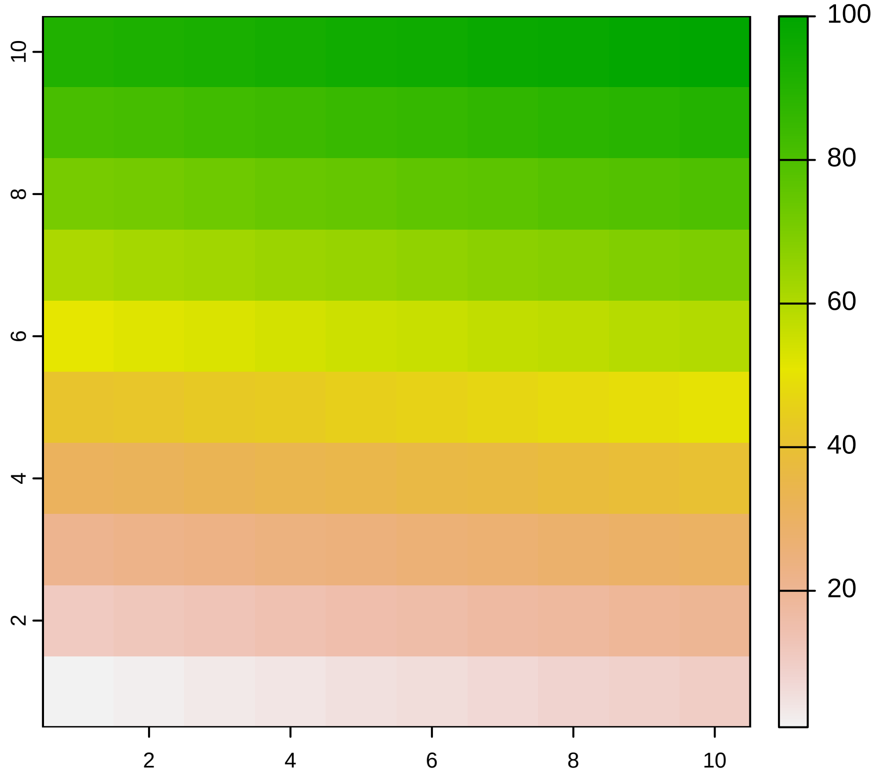
To create a raster using `terra`, you can use the `rast()` function. You can use as input a matrix.

```
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.

```
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)
```



Let's see also the associated metadata:

```r
ext(r)  # spatial extent
```

```
## SpatExtent : 0.5, 10.5, 0.5, 10.5 (xmin, xmax, ymin, ymax)
```

```r
res(r)  # resolution, i.e. cell pixel size
```

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

```r
crs(r, proj = TRUE)  # CRS
```

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

3.4 Reading and writing rasters

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

```r
r <- rast(<filename>)
```

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

```r
writeRaster(<spatRast>, <filename>)
```

3.5 Raster manipulation

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

```r
r + 10  # add a constant
2 * r  # multiply by a constant
r ^ 2  # power
2 * r + 10  # combined addition and multiplication
```

You can also add and multiply rasters together:

```r
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 * 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
```

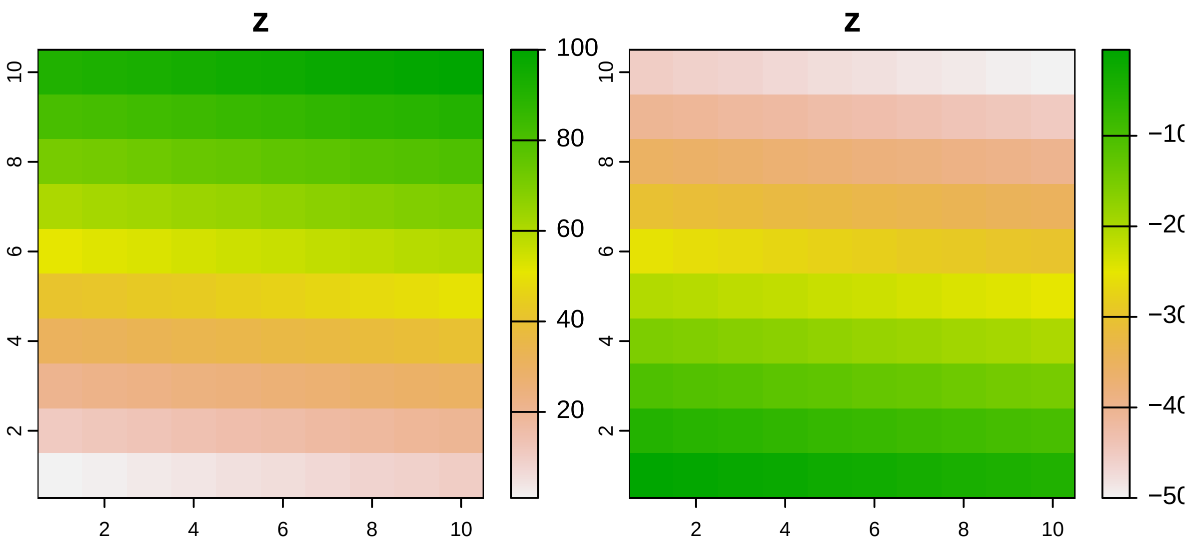
3.6 Raster stacks

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

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

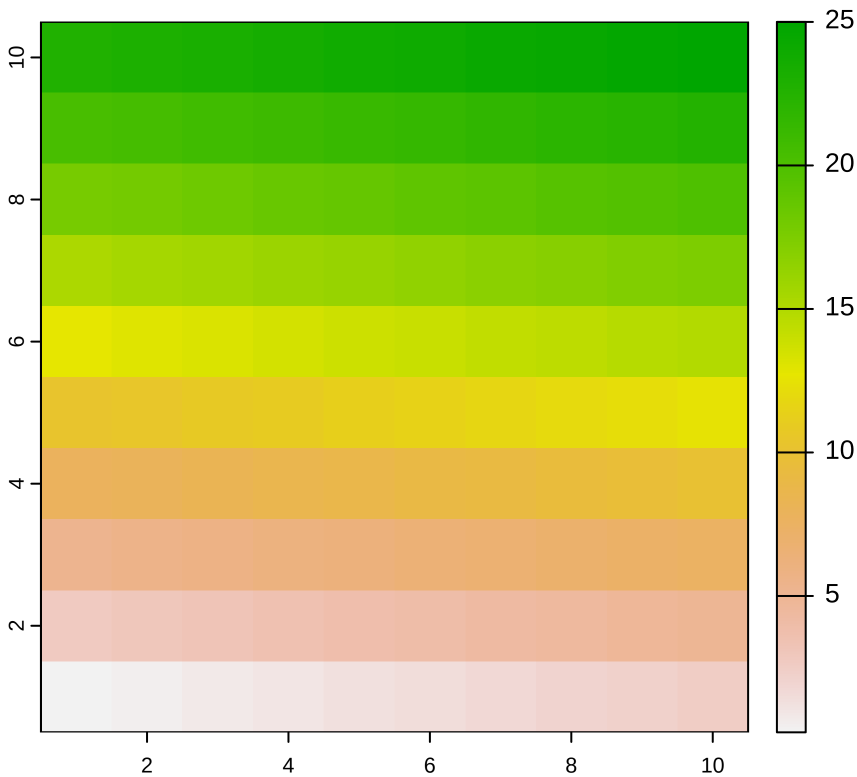
```
## class       : SpatRaster
## dimensions  : 10, 10, 2  (nrow, ncol, nlyr)
## resolution  : 1, 1  (x, y)
## extent      : 0.5, 10.5, 0.5, 10.5  (xmin, xmax, ymin, ymax)
## coord. ref. : lon/lat WGS 84 (EPSG:4326)
## source(s)   : memory
## names       :   z,     z
## min values  :   1, -50.0
## max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

```r
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
```

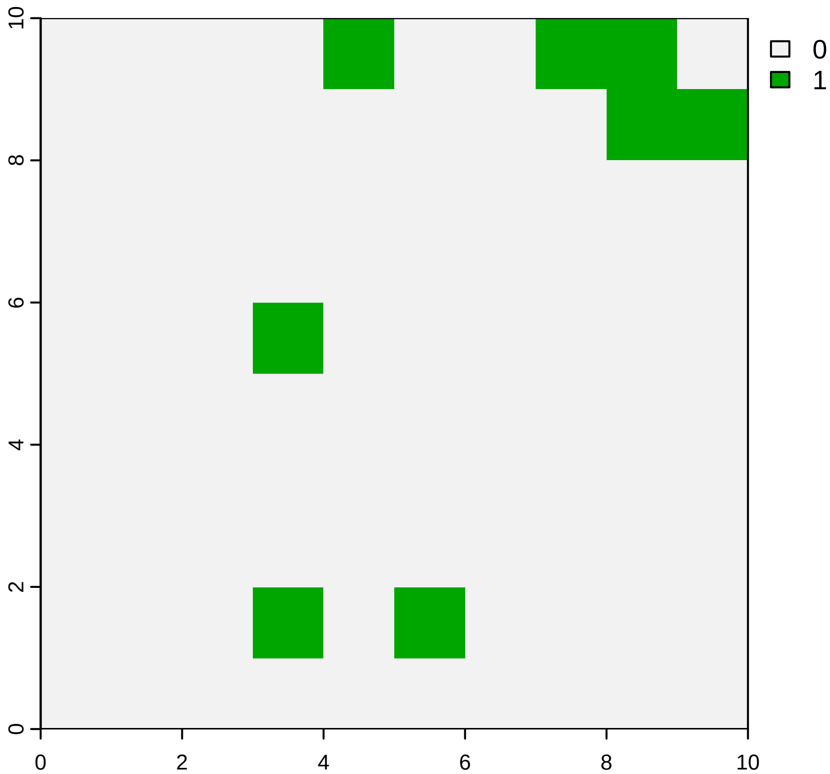
```
prod(s)  # product
```

```
## 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
```

3.7 Example: species richness per cell

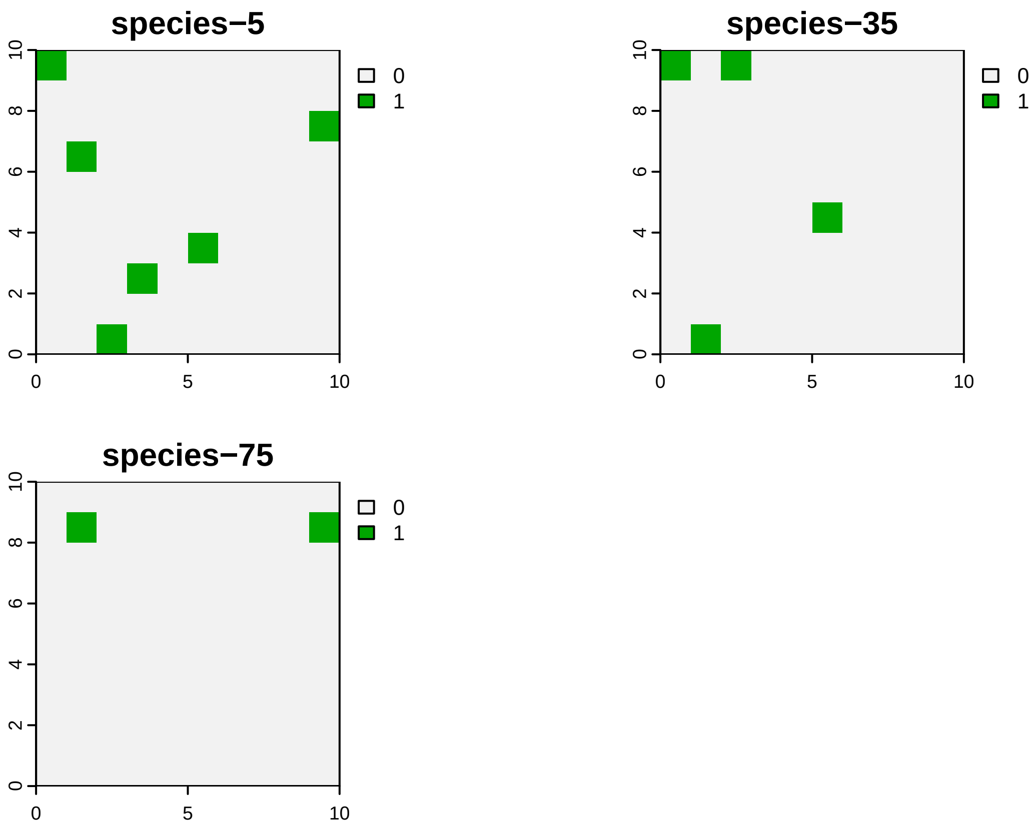
Imagine you sampled the occurrences for 100 species and want to know the number of species detected in each cell. The distribution of one species is saved in the file *species-1.tif* and may look like this:

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



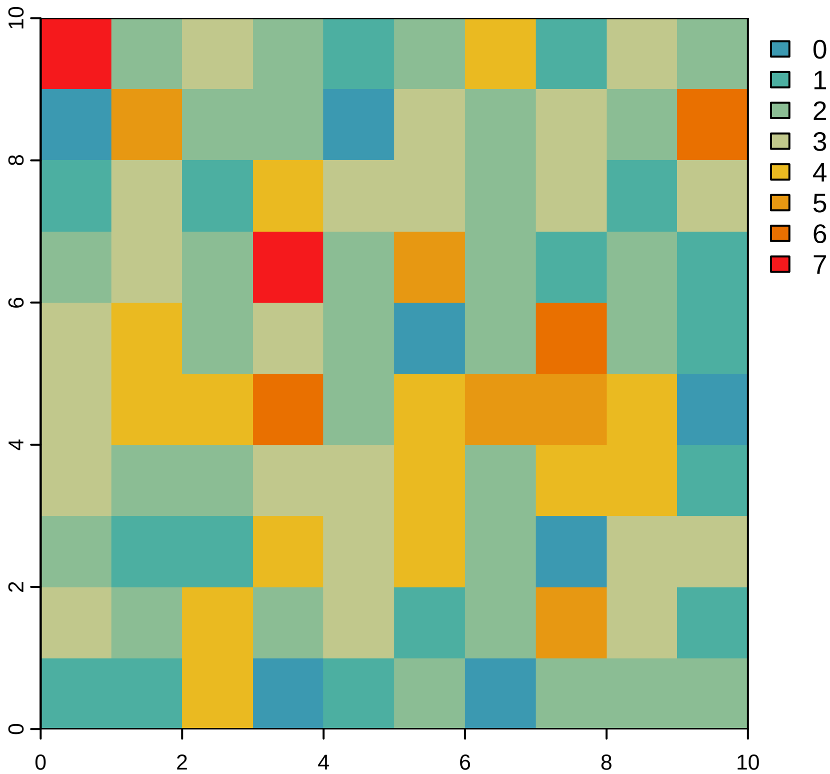
You can load all of them at once:

```
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:

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


0
1
2
3
4
5
6
7
0
2
4
6
8
10
0
2
4
6
8
10

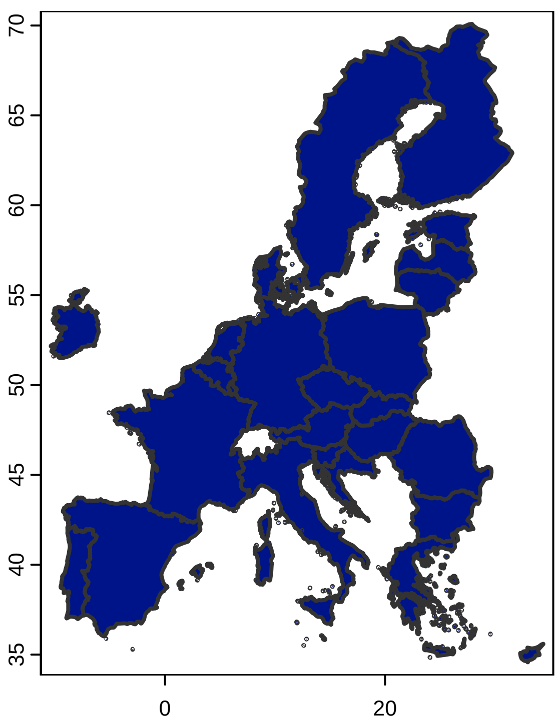
Chapter 4

Overview of the data we will use

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? To illustrate this with practical examples, we will use some data that you can find in the `data/` folder in the repo https://github.com/emilio-berti/gis-course.

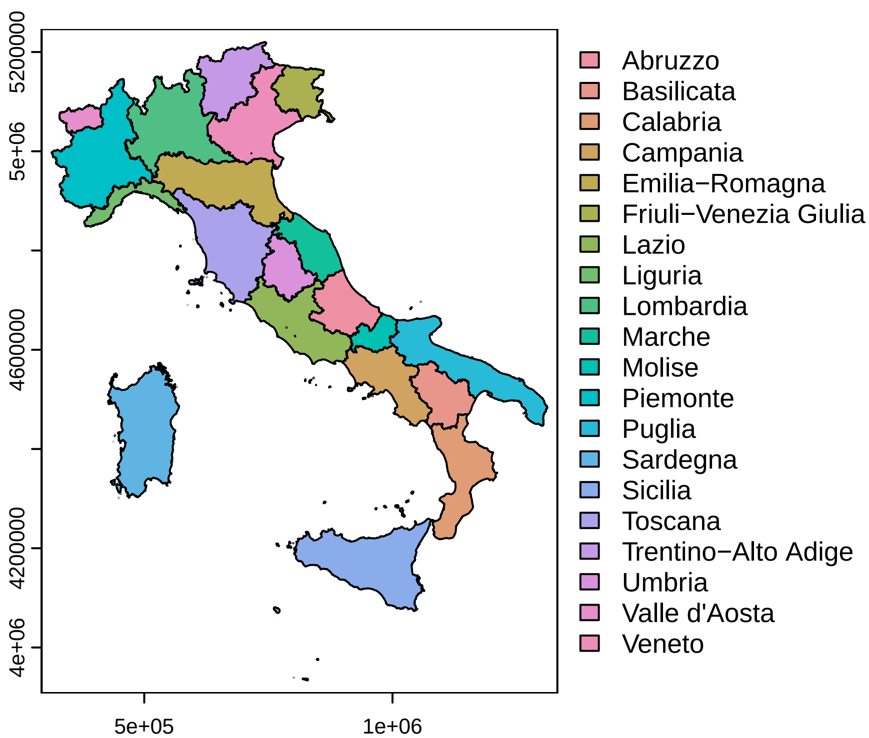
`data/EU/EU.shp` contains the polygons of the countries in the European Union:

```
eu <- vect("data/EU/EU.shp")
plot(eu, col = "#001489", border = "grey20", lw = 2)
```



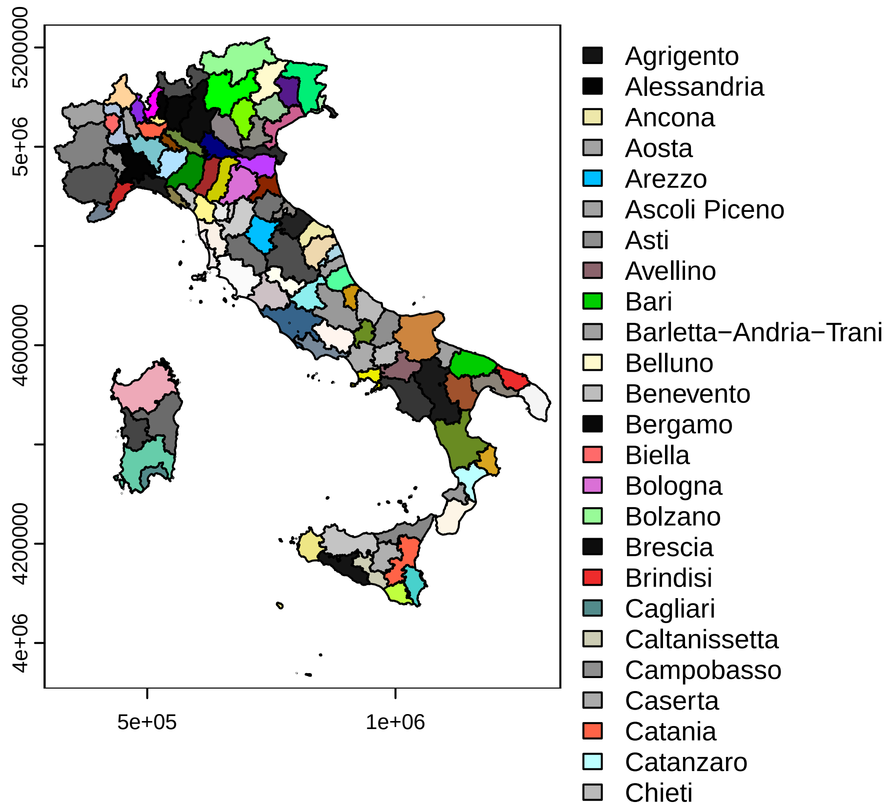
- `data/ISTAT/Limiti01012024_g/Reg01012024_g/Reg01012024_g_WGS84.shp` contains the polygons of the regions (administrative division below the state) of Italy:

```r
reg <- vect("data/ISTAT/Limiti01012024_g/Reg01012024_g/Reg01012024_g_WGS84.shp")
plot(
  reg, "DEN_REG",
  col = hcl.colors(length(reg), "Set 2")
)
```



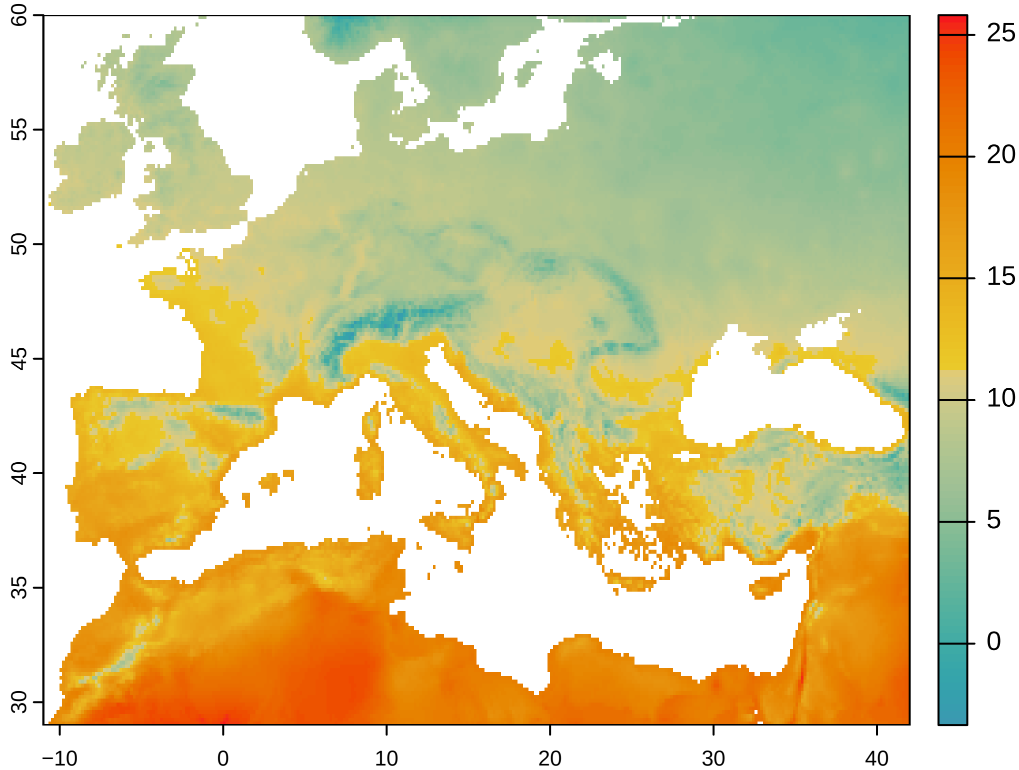
- `data/ISTAT/Limiti01012024_g/ProvCM01012024_g/ProvCM01012024_g_WGS84.shp` contains the polygons of the provinces (administrative division between communes and regions) of Italy:

```r
italy <- vect("data/ISTAT/Limiti01012024_g/ProvCM01012024_g/ProvCM01012024_g_WGS84.shp")
plot(
  italy, "DEN_UTS",
  col = colors()[sample(seq_along(colors()), length(italy))]
)
```



- `data/wc2.1_10m_bio_1.tif` is the raster of the mean annual temperature (averaged across several decades) at 10 minute resolution from WorldClim. I cropped this for Europe (excluding parts of Scandinavia to save disk space):

```
bio1 <- rast("data/wc2.1_10m_bio_1.tif")
plot(bio1, col = hcl.colors(100, "Zissou 1"))
```



60
55
50
45
40
35
30
−10
0
10
20
30
40
25
20
15
10
5
0

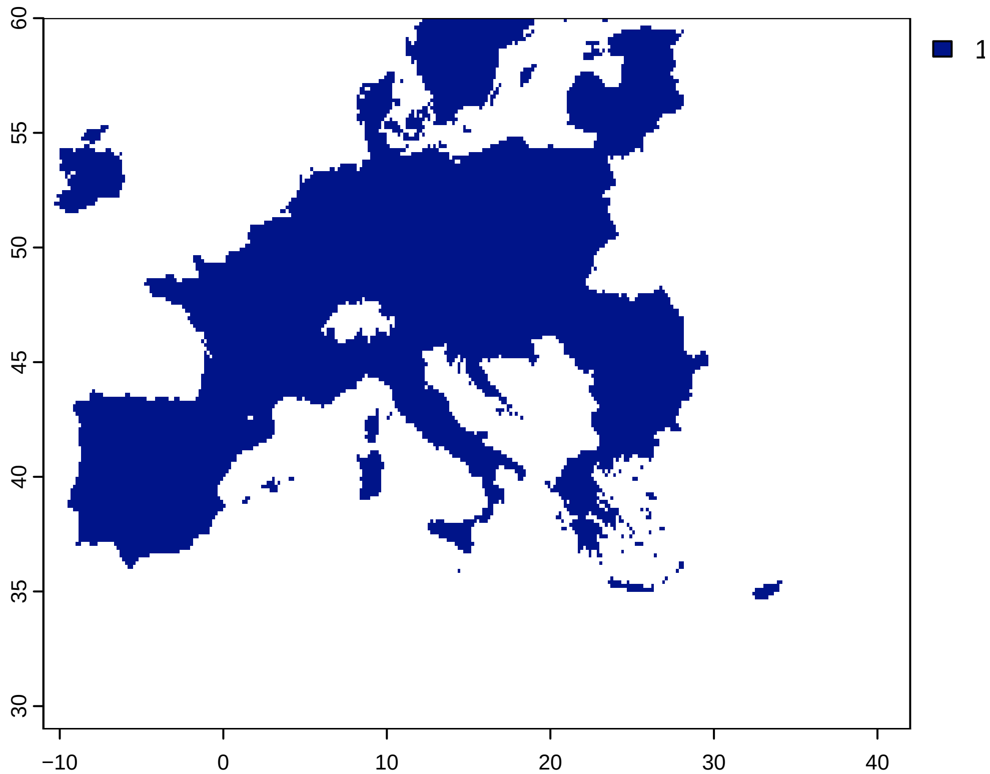
Chapter 5

Vectors to rasters and back

5.1 Convert a vector to a raster

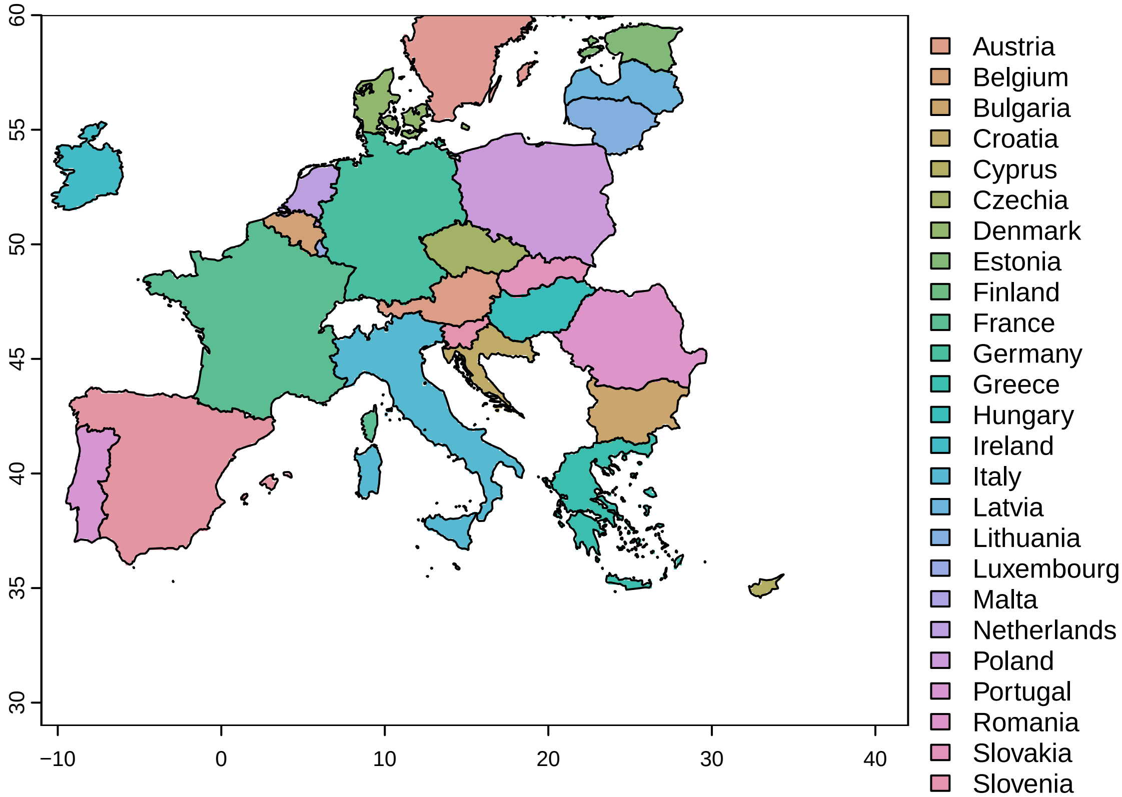
In `terra`, a vector can be converted to a raster by using `rasterize(x, y)`, where `x` is the vector to convert to a raster and `y` is a template raster from which the metadata is re-used for `x`:

```
bio1 <- rast("data/wc2.1_10m_bio_1.tif") #load raster template
eu <- vect("data/EU/EU.shp") #load zone layer
r_eu <- rasterize(eu, bio1)
plot(r_eu, col = "#001489")
```



`rasterize()` can also take the optional argument `field`, which is used to assign to the cells values:

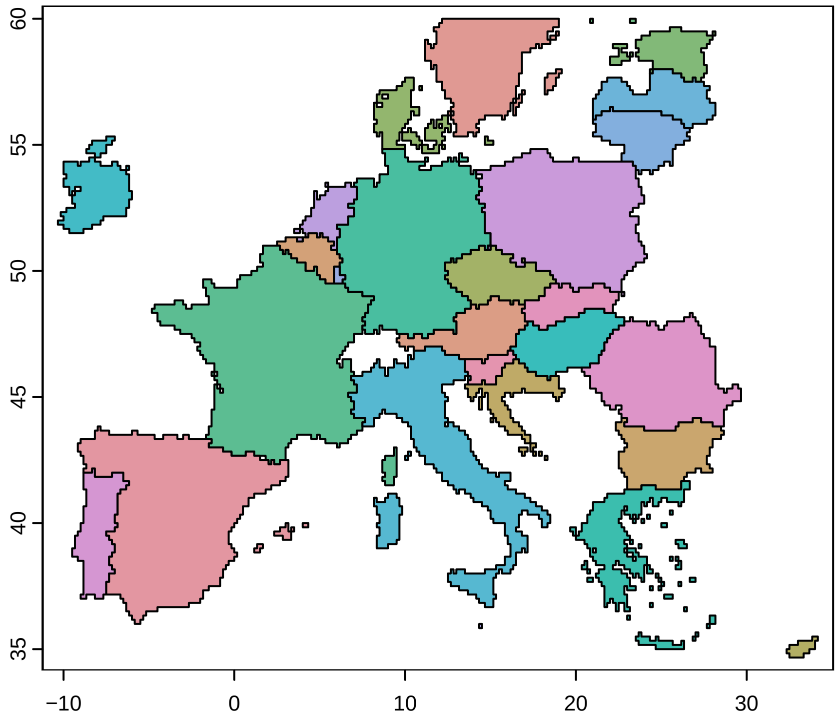
```
r_eu <- rasterize(eu, bio1, field = "NAME_ENGL")
plot(r_eu, col = hcl.colors(length(eu), "Dynamic"))
lines(eu)
```



5.2 Covert a raster to a vector

`as.polygons()` can be used to convert a raster into a vector:

```
plot(
  as.polygons(r_eu),  # covert it back to polygons
  col = hcl.colors(length(eu), "Dynamic")
)
```



By default, the new polygon will be dissolved, i.e. the vector will have only one geometry for each value of the raster; this can be turned off by specifying `dissolve = FALSE`, but usually you want them dissolved.

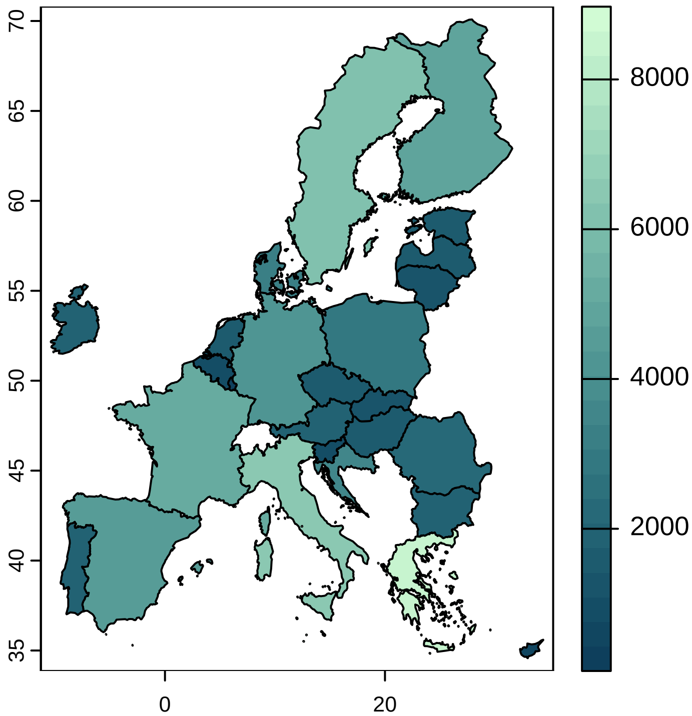
Chapter 6

Simple operations on vectors

6.1 Perimiter

`perim()` is used to get the perimiter of a vector:

```
eu <- vect("data/EU/EU.shp")
eu[["perimeter"]] <- perim(eu) / 1e3 #in km
plot(
  eu, "perimeter",
  type = "continuous",
  col = hcl.colors(length(eu), "Dark Mint")
)
```

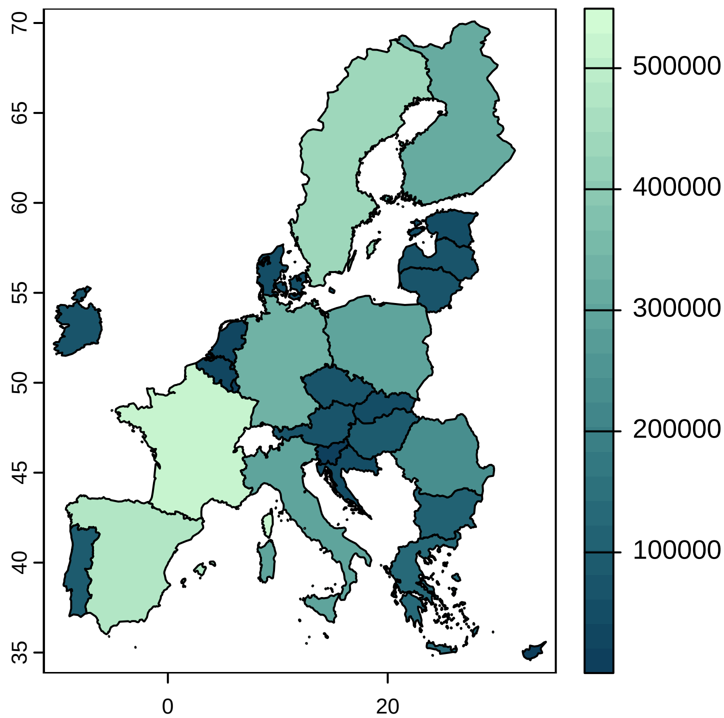


(Look at Greece with all that islands!)

6.2 Area

`expanse()` is used to calculate the area of vectors:

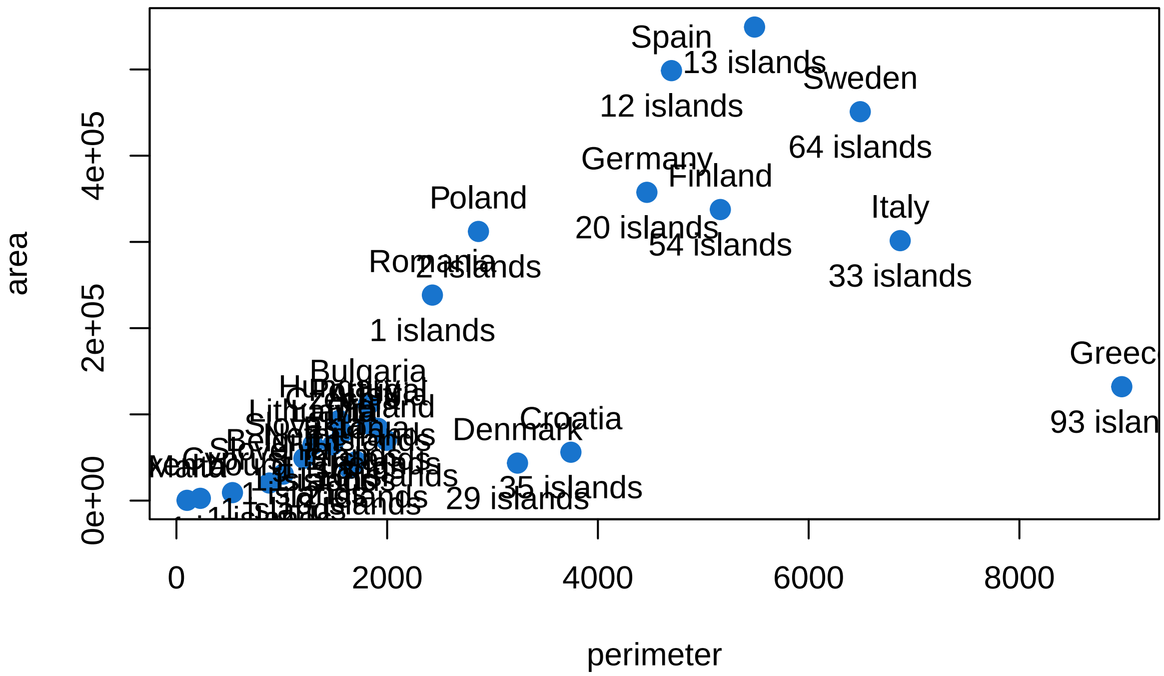
```
eu[["area"]] <- expanse(eu, unit = "km")
plot(
  eu, "area",
  type = "continuous",
  col = hcl.colors(length(eu), "Dark Mint")
)
```



(Sweden is smaller than what it looks)

Bonus:

```
mapped_islands <- table(disagg(eu)$NAME_ENGL)[eu$NAME_ENGL] #some magic
with(as.data.frame(eu), plot(
  perimeter, area,
  pch = 20, cex = 2,
  col = "dodgerblue3"
))
text(
  x = eu$perimeter, y = eu$area,
  labels = eu$NAME_ENGL,
  adj = c(0.5, -1),
)
text(
  x = eu$perimeter, y = eu$area,
  labels = paste(mapped_islands, "islands"),
  adj = c(0.5, 2),
)
```

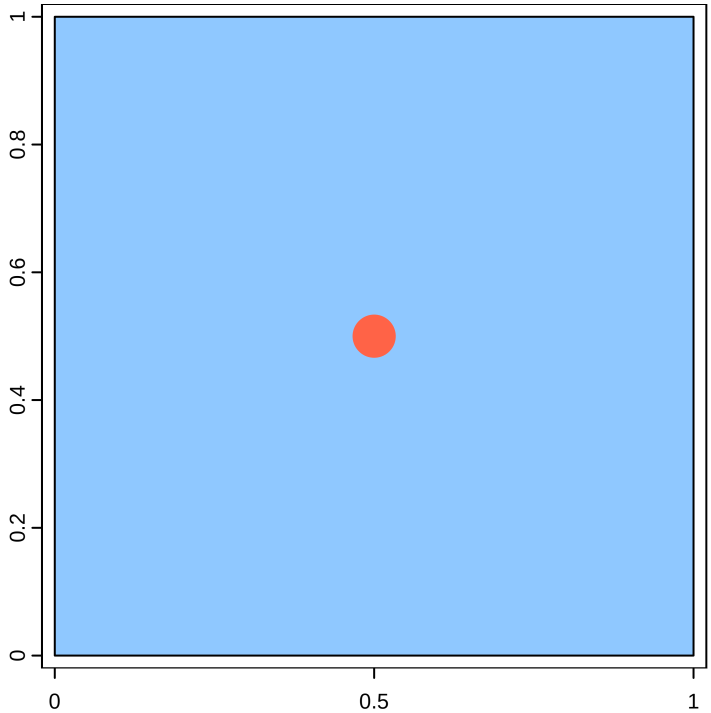


(I call this relationship the *island factor*)

6.3 Centroids

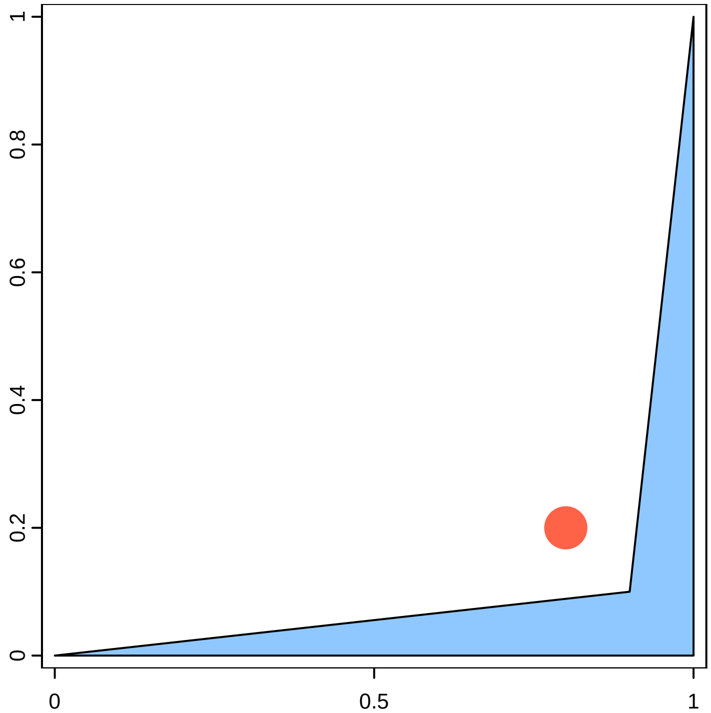
The centroid is the point defined as the arithmetic mean position of all the points in the surface of the polygon. For instance, the centroids of a square polygon it's its center:

```
p <- vect(
  matrix(
    c(0, 0, 1, 0, 1, 1, 0, 1, 0, 0), byrow = TRUE, ncol = 2
  )
)
p <- as.lines(p)
p <- as.polygons(p)
centr <- centroids(p)
plot(p, col = "dodgerblue", alpha = .5)
points(centr, col = "tomato", cex = 3)
```



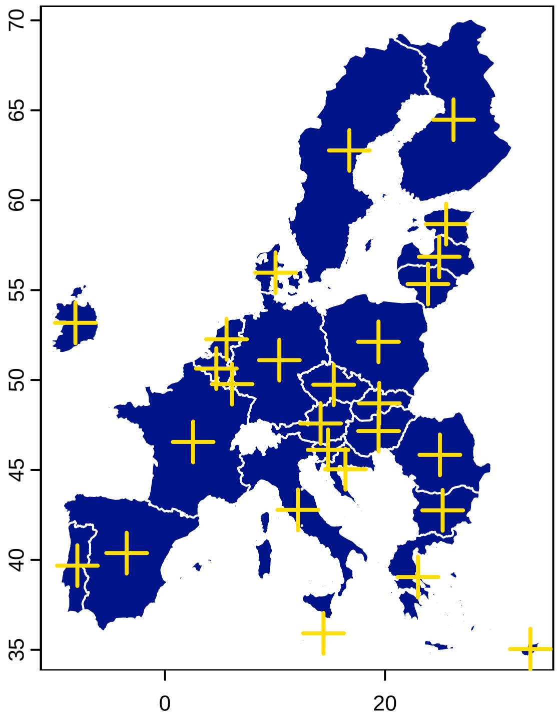
But the centroid of a complex polygon may not even be inside it:

```
p <- vect(matrix(c(0, 0, 1, 0, 1, 1, 0.9, 0.1, 0, 0), byrow = TRUE, ncol = 2))
p <- as.lines(p)
p <- as.polygons(p)
centr <- centroids(p)
plot(p, col = "dodgerblue", alpha = .5)
points(centr, col = "tomato", cex = 3)
```



Neverthelss, centroids can someitmes be useful to get an idea of a process at large spatial scales. For instance, the latitude of the centroids of the countries in the EU is somewhat their average latitude:

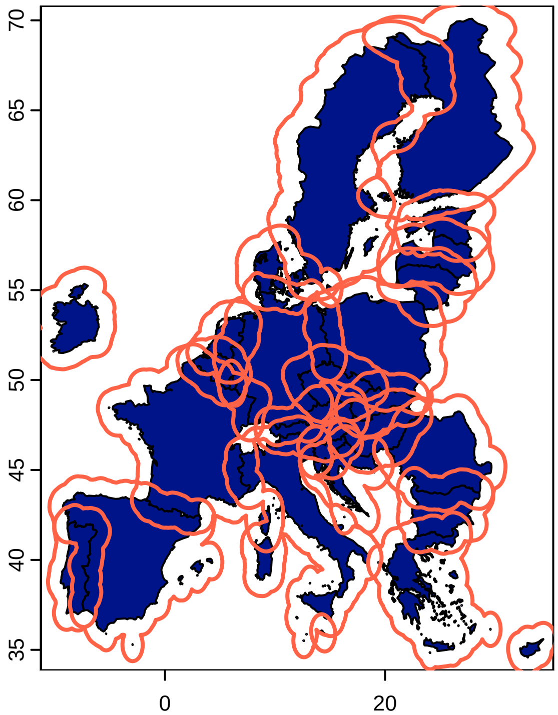
```
eu_centr <- centroids(eu)
eu <- vect("data/EU/EU.shp")
plot(eu, col = "#001489", border = "white", lw = 1)
points(eu_centr, col = "#FFDD00", cex = 2, pch = 3, lw = 2)
```



6.4 Buffer

To *buffer* a polygon is to extend its perimeter in all directions at the same time, i.e. orthogonally to the tangent line of each point. Buffering is achieved in `terra` using `buffer(x, width)`. It is easier to see it than to explain it.

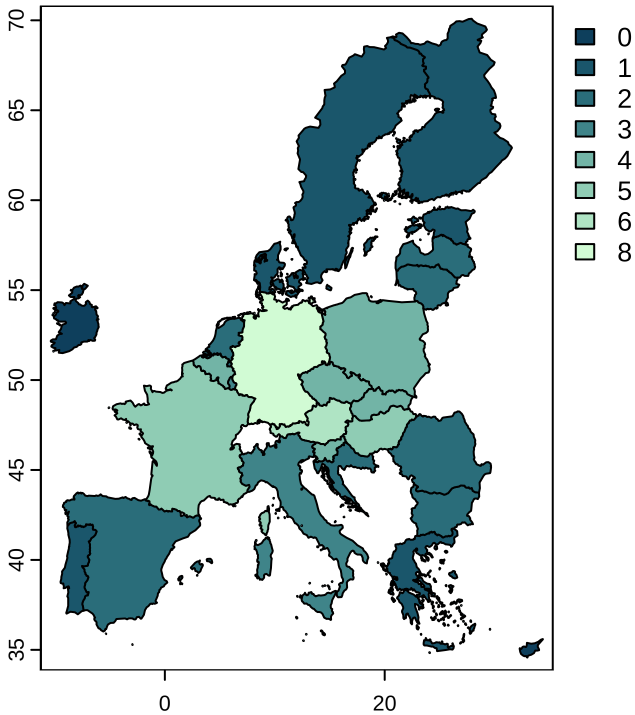
```
b <- buffer(eu, 1e5) #100km for visualizing it
plot(eu, col = "#001489")
lines(b, col = "tomato", lw = 2)
```



6.5 Neighbors of polygons

In terra, neighbors of a polygon are obtained using `adjacent()`. The output of `adjacent()` is a matrix, either of two columns with the first being the IDs of the n countries or of $n \times n$ representing the adjacency matrix of the countries. I like more the adjacency matrix, therefore:

```
neigh <- as.list(rep(NA, length(eu))) #initialize empty list
names(neigh) <- eu$NAME_ENGL
adj <- adjacent(eu, pairs = FALSE)

# then a convoluted code to show the number of neighbors
for (i in seq_along(eu)) {
  ctr <- eu[adj[i, ] == 1, ]
  if(length(ctr) > 0) {
    neigh[[i]] <- unique(ctr$NAME_ENGL)
  } else {
    neigh[[i]] <- "No neighbours"
  }
}
eu$neigh <- sapply(neigh, length)
eu$neigh[sapply(neigh, \(x) all(x == "No neighbours"))] <- 0
plot(
  eu, "neigh",
  col = hcl.colors(max(eu$neigh) + 1, "Dark Mint")
)
```



(Germany truly is the hearth of the EU)

Chapter 7

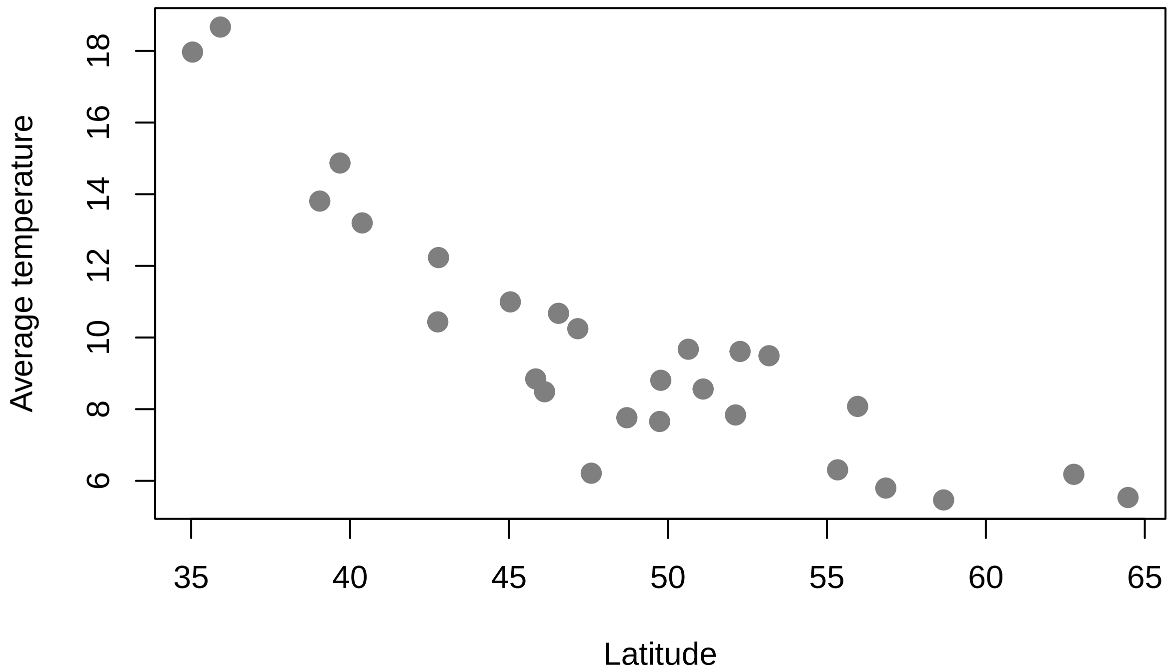
Zonal statistics

Zonal statstics are calcualted for *zones*, i.e. regions defined either by a cell value (for rasters) or by polygons (for shapefiles). Zonal calculation are implemented in *terra* using `zonal()`.

7.1 Zones as a shapefile

When zones are defined by a shapefile, e.g. the polygons of EU, zonal statistics are calculated. The syntax is `zonal(x, y, fun)`, where `x` is the layer with the values to calculate the statistics of, `y` is the the layer with the zones, and `fun` is the function of the statistic. For example, we can calculate the average value of the annual temperature for a coutry and see how it varies with latitude:

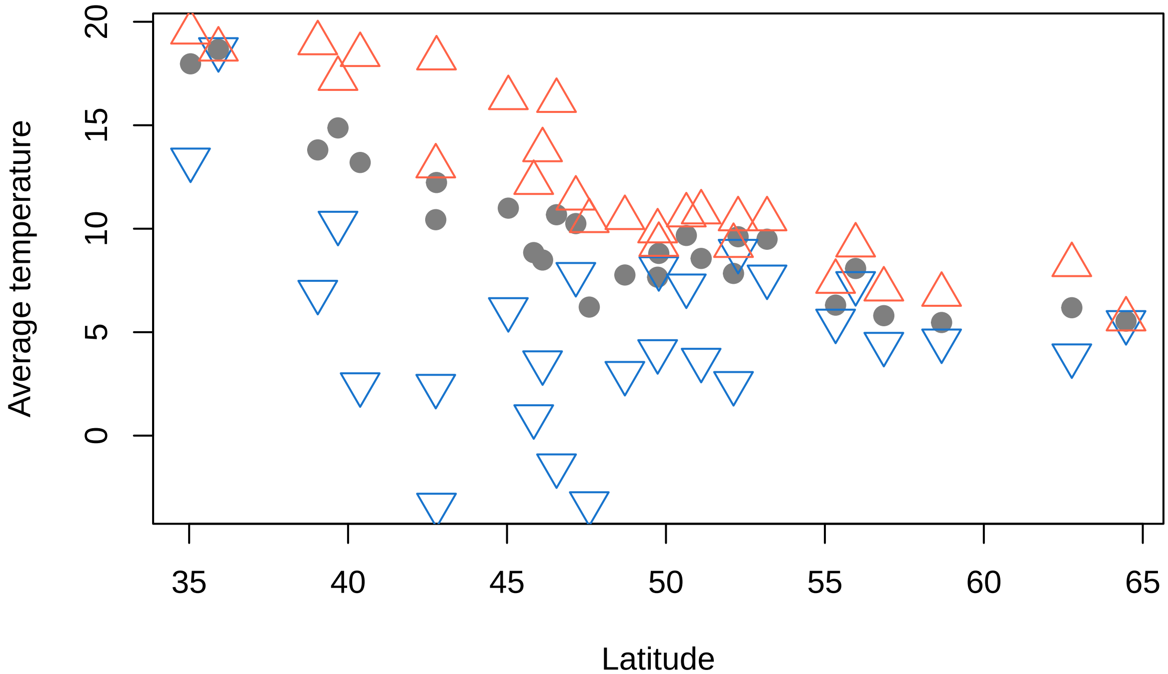
```
eu <- vect("data/EU/EU.shp") #load zone layer
eu_centr <- centroids(eu) #get centroids
lat <- geom(eu_centr)[, "y"] #get latitude of centroids

bio1 <- rast("data/wc2.1_10m_bio_1.tif") #load temperature layer

bio1_mean <- zonal(bio1, eu, "mean") #zonal calculation - mean value
bio1_mean <- bio1_mean[[1]] #as vector
plot(
  lat, bio1_mean,
  pch = 20, cex = 2,
  col = "grey50",
  xlab = "Latitude",
  ylab = "Average temperature"
)
```



In the case above `x` is a raster (it usually is), but this can also be a geometry.

By using another `fun` we can obtain other statistics. For example, we can get the minimum and maximum values of each country:

```
bio1_min <- zonal(bio1, eu, "min") #zonal calculation
bio1_min <- bio1_min[[1]] #as vector
bio1_max <- zonal(bio1, eu, "max") #zonal calculation
bio1_max <- bio1_max[[1]] #as vector
plot(
  lat, bio1_mean,
  pch = 20, cex = 2,
  col = "grey50",
  xlab = "Latitude",
  ylab = "Average temperature",
  ylim = c(min(bio1_min), max(bio1_max))
)
points(
  lat, bio1_min,
  pch = 6, cex = 2,
  col = "dodgerblue3"
)
points(
  lat, bio1_max,
  pch = 2, cex = 2,
  col = "tomato"
)
```


Average temperature
20
15
10
5
0
35
40
45
50
55
60
65
Latitude

Chapter 8

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):

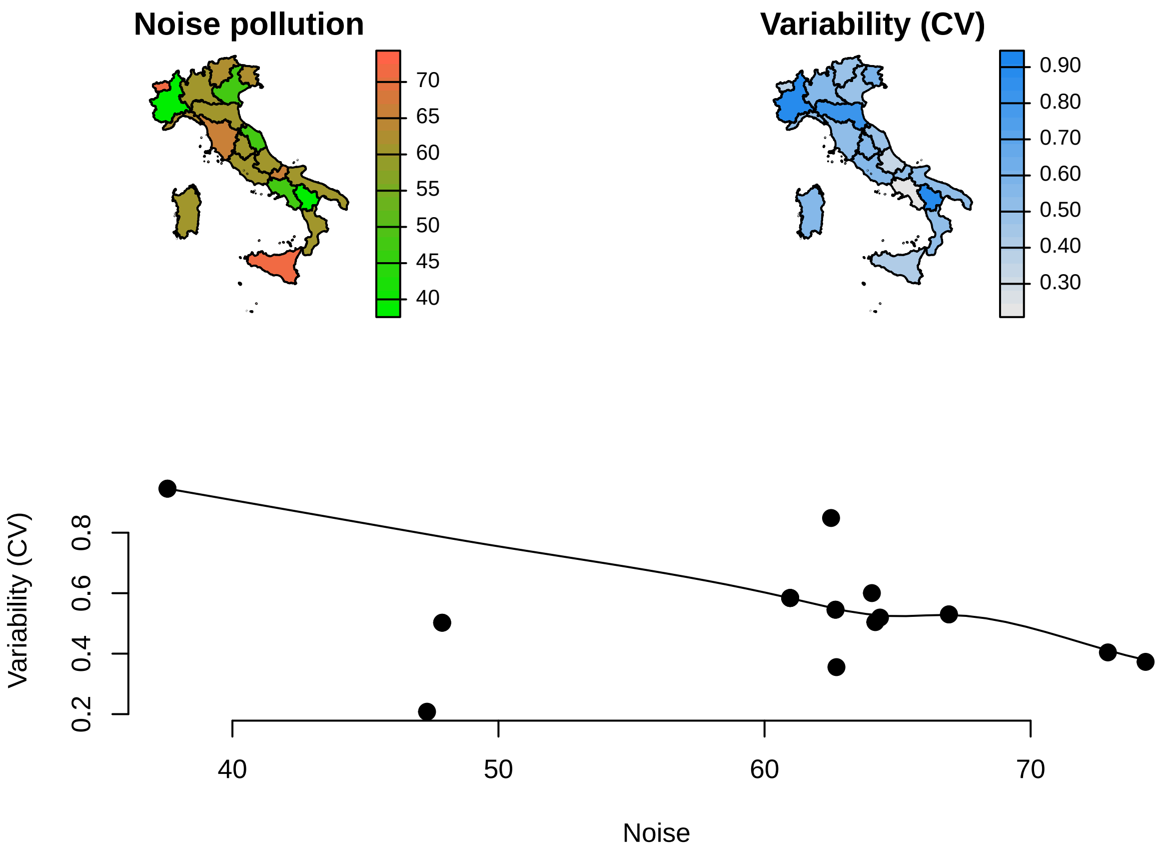
1. Administrative division shapefile (https://www.istat.it/it/archivio/222527).
2. GDP of each of the regions, the second administrative division after the State.

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

Load the vector shapefile of EU countries:

```
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$PRO_COM_T == x))]
cities <- cities[order(cities$noise)]
```

```r
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)
0.2
0.4
0.6
0.8
40
50
60
70
Noise

Chapter 9

Exercises

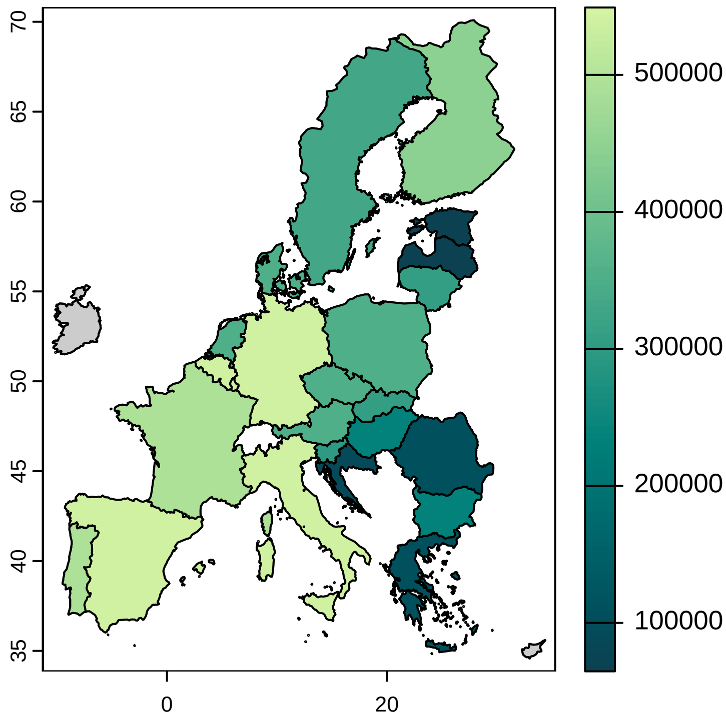
9.1 6.1 Size of neighbors of EU states

Using `data/EU/EU.shp`, find the size (km^2) of the neighbors of each state and make a map with the states colored by the size of the largest neighbor.

Solution

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

# neighbors and country size
neigh <- adjacent(eu, type = "touches", pairs = FALSE)
dimnames(neigh) <- list(eu$NAME_ENGL, eu$NAME_ENGL)
size_km <- expanse(eu, unit = "km")
names(size_km) <- eu$NAME_ENGL

# calculate maximum neighbor size
max_neigh_size <- as.list(rep(NA, length(eu)))
names(max_neigh_size) <- eu$NAME_ENGL
for (x in names(max_neigh_size)) {
  if (any(neigh[, x] == 1)) {  # some states have no neighbors
    max_neigh_size[[x]] <- max(size_km[which(neigh[, x] == 1)])
  }
}

# add attribute to the shapefile
eu$max_size_neighbor <- max_neigh_size

# plot it
plot(eu, col = "grey80")  # base map to show isolated countries
plot(
  eu, "max_size_neighbor",
  type = "continuous",
  col = hcl.colors(100, "Emrld"),
  add = TRUE
```

```
)
```


70
65
60
55
50
45
40
35
0
20
500000
400000
300000
200000
100000