GIS course @ yDiv

Emilio Berti

22 January 2025

1 Introduction

1.1 Set up

- R installed.
- `terra` installed.
- `codetools` installed (optional, used by `terra` to perform code checks).
- Download the data archive and decompress it.

1.2 Data Sets

In this tutorial, we will use data sets for:

- NDVI for the year 2024 in Leipzig, obtained from Sentinel2 (`ndvi-2024.tif`).
- Landcover of Leipzig in 2015, obtained from the German Aerospace Center (DLR; `landcover-2015.tif`).

1.3 Program

1.3.1 What we will cover

- Two days course
- First day: get to know geometries and rasters
- Second day: GIS operations + individual/group project

1.3.2 What to expect

- Familiarize yourself with basic GIS data and operations
- Start a project of your interest
- Getting some simple figures/analyses started

1.3.3 What not to expect

- To become a world-leading GIS expert

1.4 Lecturers

1.4.1 Emilio Berti (TiBS)

- Theory in Biodiversity Science
- Macroecologist / biogeographer
- Theoretical ecologists

1.4.2 Guilherme Pinto (BioEcon)

```
library(terra)
```

2 Data Types

There are two main data types used in GIS:

1. Geometries, also called vectors or shapes
2. Rasters

They fill two different needs, namely to represent structures that can be well approximated using geometric objects, such as lines, circles, etc., and to represent grid data over an area.

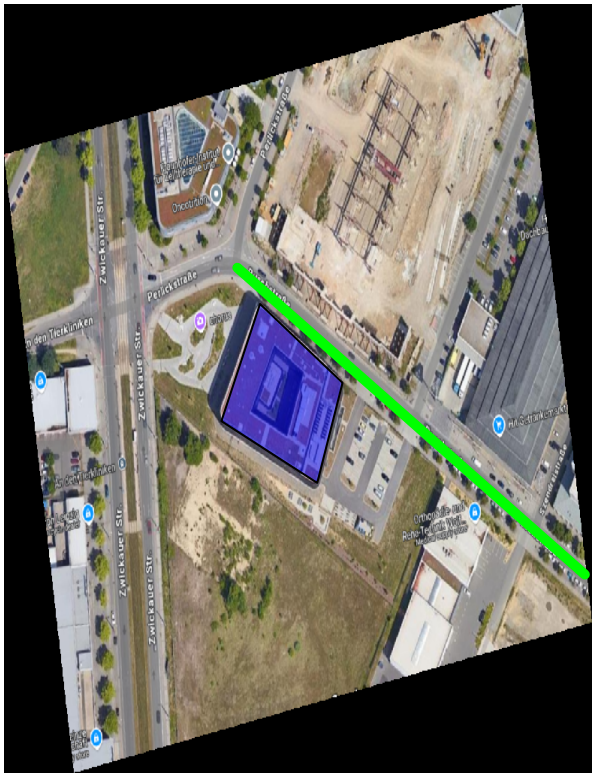
2.1 Geometries

2.1.1 Overview

Structures such as a street or a building can be represented by a geometric objects. For instance, a street can be represented by a straight line, whereas a building by a polygon. It makes sense to store the data of these type of structures as their geometric representation. For example, the building we are in and the adjacent street:

```
# This code in incomprehensible to you FOR NOW
# Just an example of geometries
r <- rast("data/idiv-building.tif")
plotRGB(r)
```

```r
l <- vect(
  rbind(
    c(12.39585, 51.31866),
    c(12.39846, 51.31722)
  ),
  crs = "EPSG:4326"
)
lines(l, col = "green", lw = 5)
idiv <- vect(
  rbind(
    c(12.39573, 51.31794),
    c(12.39645, 51.31765),
    c(12.39664, 51.31812),
    c(12.39602, 51.31849),
    c(12.39573, 51.31794)
  ),
  crs = "EPSG:4326"
) |>
  as.lines() |>
  as.polygons()
idiv$name <- "iDiv building"
idiv$perimeter <- perim(idiv)
idiv$area <- expanse(idiv)
plot(idiv, add = TRUE, col = adjustcolor("blue", alpha.f = .5))
```



Geometries are saved in *vector* or *shape* files. In addition to the geometric representation of the structure, shapefiles also contain metadata for the geometric representation, such as the its extent and coordinate reference system (we will talk about this later), and for the structure, such as their name, length, etc. For instance, the geometry of the iDiv building above is:

```
idiv
```

```
##  class       : SpatVector
##  geometry    : polygons
##  dimensions  : 1, 3  (geometries, attributes)
##  extent      : 12.39573, 12.39664, 51.31765, 51.31849  (xmin, xmax, ymin, ymax)
##  coord. ref. : lon/lat WGS 84 (EPSG:4326)
##  names       :          name perimeter  area
##  type        :         <chr>     <num> <num>
##  values      : iDiv building     237.7  3265
```

Note that this structure has one geometry and three attributes/variables (`dimensions : 1, 3 (geometries, attributes)`): `name`, `perimeter` (m), and `area` (m^2).

2.1.2 Geometry Types

In `terra`, there are three main geometry types:

1. Points: they are defined by a vector with two values for coordinates (e.g., longitude and latitude)
2. Lines: they are a series of points connected pairwise by straight lines
3. Polygons: they are a series of lines inscribing a closed area

In `terra`, geometries are created with the function `vect()`. To create a point geometry, you need to pass a matrix with the coordinates:

```
xy <- cbind(12.39585, 51.31866)  # cbind force it to be a matrix
vect(xy)
```

```
##  class       : SpatVector
##  geometry    : points
##  dimensions  : 1, 0  (geometries, attributes)
##  extent      : 12.39585, 12.39585, 51.31866, 51.31866  (xmin, xmax, ymin, ymax)
##  coord. ref. :
```

To create a geometry of multiple points, simply pass a matrix that has multiple rows:

```
xy <- rbind(
  cbind(12.39585, 51.31866),
  cbind(12.39584, 51.31865)
)
poi <- vect(xy)
```

In order to create lines, first create a multi-point geometry, then convert it using `as.lines()`:

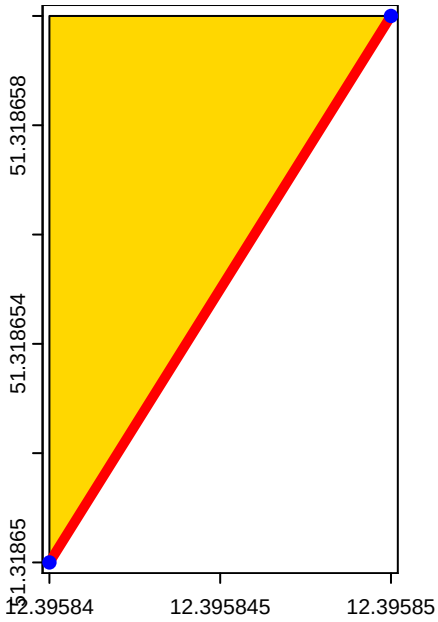
```
xy <- rbind(
  cbind(12.39585, 51.31866),
  cbind(12.39584, 51.31865)
)
lin <- vect(xy) |> as.lines()
```

In order to create polygons, first create a mutli-point geometry, then covert it to lines, and finally to polygons using `as.polygons()`. The last point should be the same as the first to close the geometry.

```
xy <- rbind(
  cbind(12.39585, 51.31866),
  cbind(12.39584, 51.31865),
  cbind(12.39584, 51.31866),
  cbind(12.39585, 51.31866)
```

```
)
pol <- vect(xy) |> as.lines() |> as.polygons()
```

```
plot(pol, col = "gold")
lines(lin, col = "red", lw = 5)
points(poi, col = "blue", cex = 1)
```



2.1.3 Reading & Writing Geometries

`vect()` is also used to read geometries from a file. For instance, to read the shapefile with the country boundary of Germany:

```
ger <- vect("data/germany.shp")
ger
```

```
##  class       : SpatVector
##  geometry    : polygons
##  dimensions  : 1, 5  (geometries, attributes)
##  extent      : 5.866755, 15.04179, 47.27012, 55.05866  (xmin, xmax, ymin, ymax)
##  source      : germany.shp
##  coord. ref. : lon/lat WGS 84 (EPSG:4326)
##  names       :            shapeName shapeISO              shapeID shapeGroup
##  type        :                <chr>    <chr>                <chr>      <chr>
##  values      : the Federal Republi~      DEU 19620994B6459175825~        DEU
##  shapeType
##      <chr>
##       ADM0
```

To write geometries to disk, use `writeVector()`.

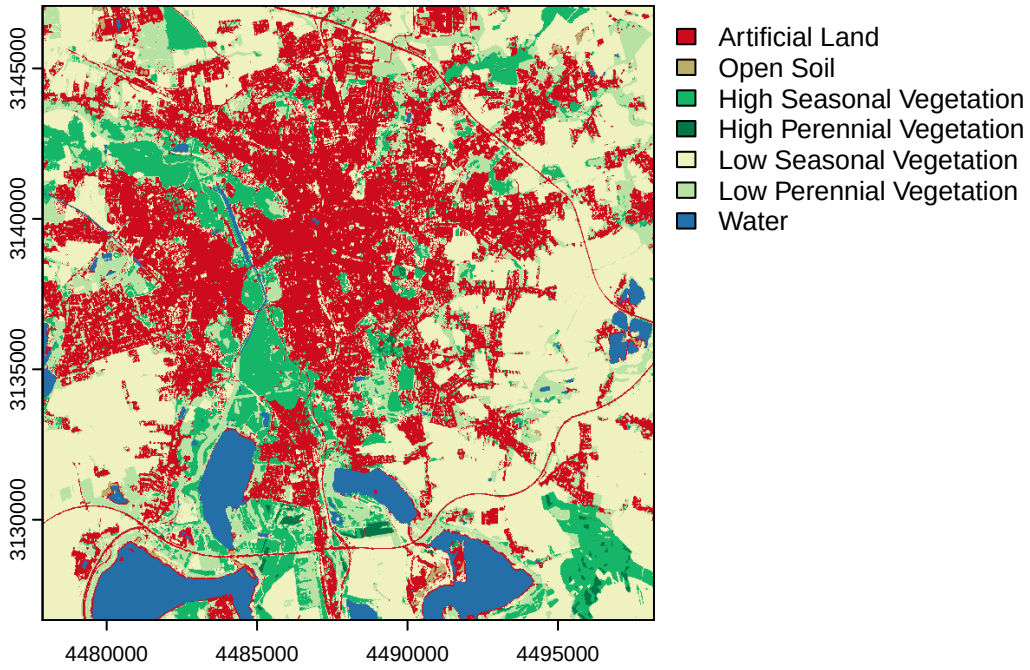
```
writeVector(ger, "data/germany.shp", overwrite = TRUE)
```

2.2 Rasters

2.2.1 Overview

Rasters are gridded areas where each grid pixel assumes a value. When the data we are interested in are values on a gridded area, rasters are a better options than to using geometries. For example, if we want to represent landcover type over a large area(forest, agriculture, build-up, etc.), it is easier to grid the area into pixels and save the landcover type of each pixel rather than to create a geometry for each different structure.

```
r <- rast("data/landcover-2015.tif")
plot(r)  # from EOC of DLR
```



Rasters are saved in raster files. In addition to the values of each grid, rasterfiles also contain metadata for the grid, such as the its extent and coordinate reference system (we will talk about this later). For instance, some of the metadata of the landcover raster:

```
r
```

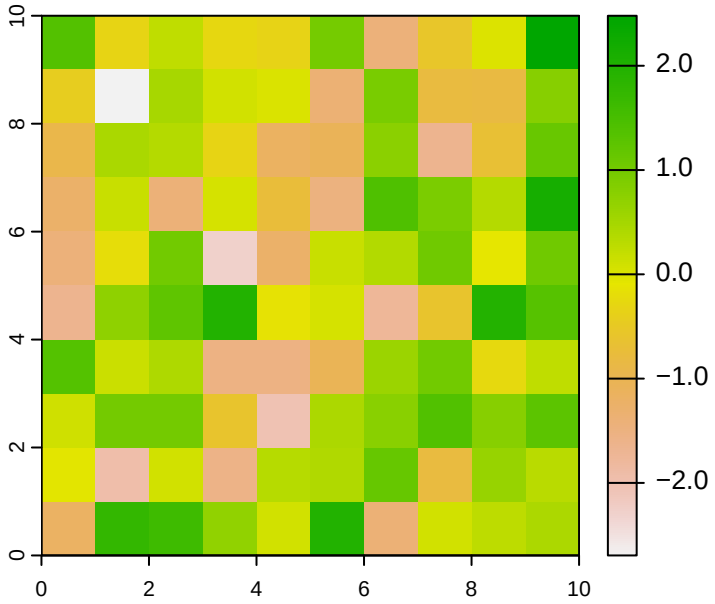
```
## class       : SpatRaster
## dimensions  : 2042, 2032, 1  (nrow, ncol, nlyr)
## resolution  : 10, 10  (x, y)
## extent      : 4477870, 4498190, 3126660, 3147080  (xmin, xmax, ymin, ymax)
## coord. ref. : ETRS89-extended / LAEA Europe (EPSG:3035)
## source      : landcover-2015.tif
## color table : 1
## categories  : category
## name        :        category
## min value   : Artificial Land
## max value   :           Water
```

Note the metadata `dimensions` (grid size), `resolution` (spatial resolution, in m), and `extent` (spatial extent).

2.2.2 Creating Rasters from Scratch

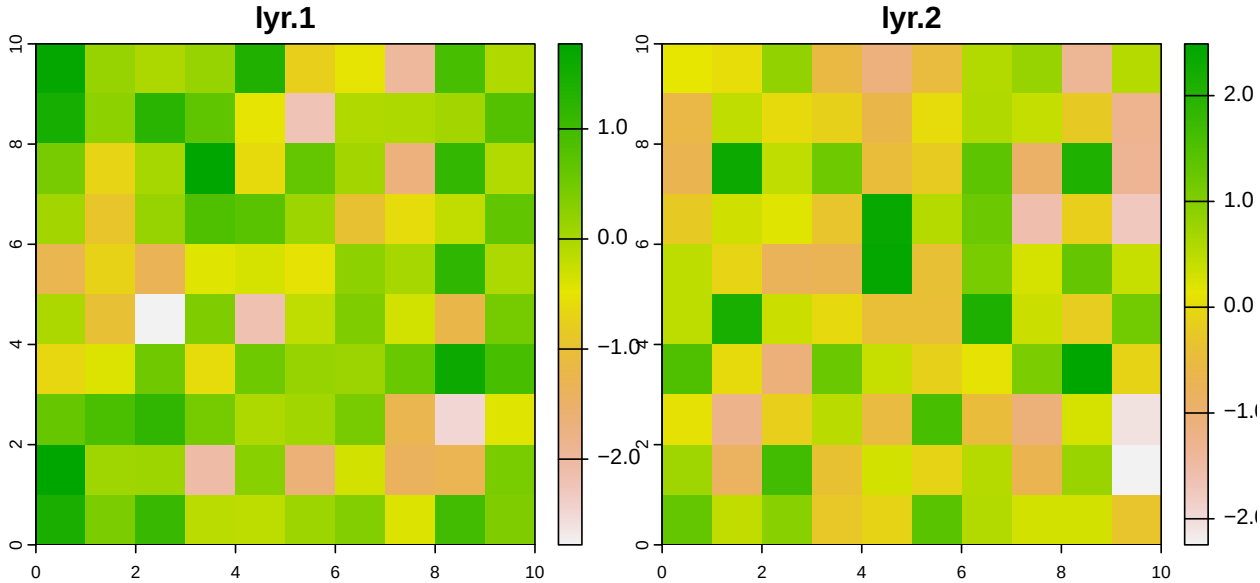
Simply put, rasters are array associated spatial metadata. In `terra`, use `rast()` to create a raster from a matrix.

```
m <- matrix(rnorm(100), nrow = 10, ncol = 10)  # 10x10 matrix
r <- rast(m)
plot(r)
```



Usually, you will not create rasters from scratch. However, if you are familiar with R arrays, you will notice a nice properties that is inherited by rasters: rasters can have multiple layers, with each layers being a matrix.

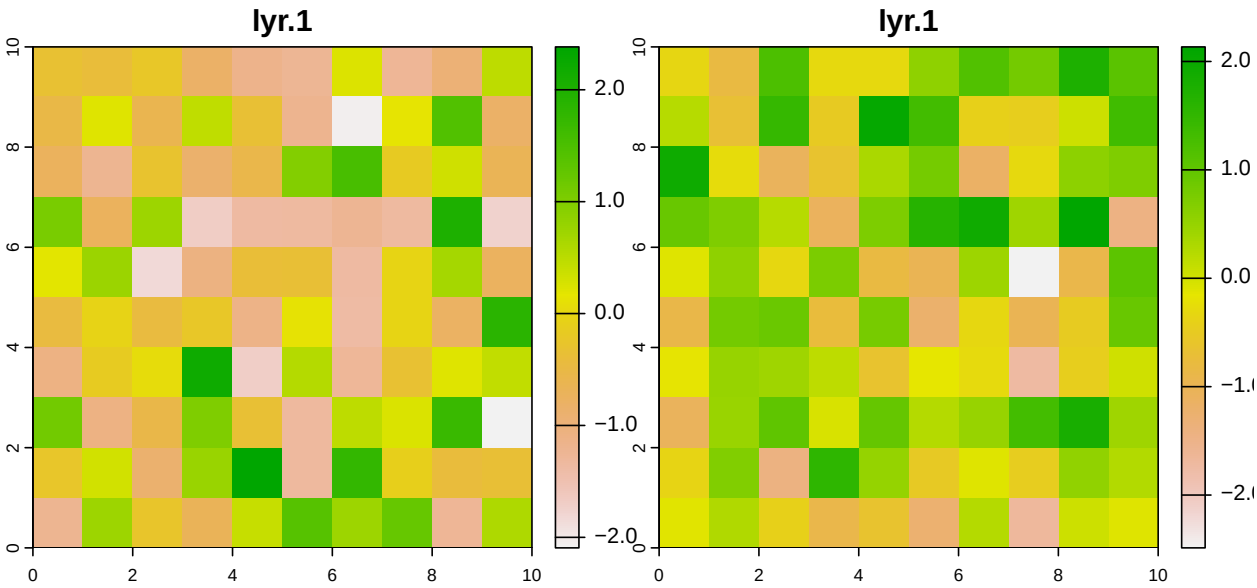
```
r <- rast(array(rnorm(200), dim = c(10, 10, 2)))
plot(r)
```



When an area has multiple rasters, it is possible to stack them to create a single raster object. The rasters in the stack need to have all the same extent and coordinate reference system. Use `c()` to stack rasters using

terra.

```
r1 <- rast(matrix(rnorm(100), nrow = 10, ncol = 10))
r2 <- rast(matrix(rnorm(100), nrow = 10, ncol = 10))
r <- c(r1, r2)  # a stack
plot(r)
```



2.2.3 Reading & Writing Rasters

To read rasters into memory, use `rast()`.

```
r <- rast("data/landcover-2015.tif")
r
```

```
## class       : SpatRaster
## dimensions  : 2042, 2032, 1  (nrow, ncol, nlyr)
## resolution  : 10, 10  (x, y)
## extent      : 4477870, 4498190, 3126660, 3147080  (xmin, xmax, ymin, ymax)
## coord. ref. : ETRS89-extended / LAEA Europe (EPSG:3035)
## source      : landcover-2015.tif
## color table : 1
## categories  : category
## name        :        category
## min value   : Artificial Land
## max value   :           Water
```

To write rasters to disk, use `writeRaster()`.

```
writeRaster(r, "data/landcover-2015.tif", overwrite = TRUE, datatype = "INT1U")
```

Here, the argument `datatype = "INTU1"` (specifying the values in the grid are 1-byte unsigned integers) is needed because the rasters contains classes (levels). In most cases, this argument is best left unspecified, as `terra` picks the optimal value by default.

2.3 Spatial metadata

Spatial objects, being geometries or rasters, always need certain metadata to be useful. For example, a raster is of little help if there is not information of the area it covers. The most important metadata are:

1. Spatial extent: the four corners of the quadrilateral polygon that inscribe the object represented
2. Coordinate reference system: how is the earth surface represented
3. Resolution (for rasters only)

We will cover each one of these three metadata in detail.

2.3.1 Spatial extent

The spatial extent is the quadrilateral that inscribe the area of the spatial data. The extent is usually represented by the coordinates of the four vertices of the quadrilateral, i.e. `xmin`, `xmax`, `ymin`, and `ymax`. To get the spatial extent of an object, use `ext()`.

```
r <- rast("data/landcover-2015.tif")
ext(r)
```

```
## SpatExtent : 4477870, 4498190, 3126660, 3147080 (xmin, xmax, ymin, ymax)
```

2.3.2 Coordinate reference system

GIS try to represent the surface of the Earth, a 3D spheroid, onto a plane. The coordinate reference system (CRS), also known as spatial reference system (SRS), defines how this *projection* of a 3D object to a 2D one is achieved. It is not possible to achieve this projection accurately and some distortions will always be present. In particular, at least one of distance, angular conformity, and area will be distorted. 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[1]).
- Equidistant: they correctly represent distances (e.g. ESRI:54002).
- Equal-area: they correctly represent areas (e.g. ESRI:54034).

An overview of ESRI and EPSG projections can be found 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.

To know the CRS of an object, use `crs()`. By default, `crs()` displays the CRS in Well Known Text (WKT) format.

```
crs(idiv, parse = TRUE)
```

```
##  [1] "GEOGCRS[\"WGS 84\","
##  [2] "    DATUM[\"World Geodetic System 1984\","
##  [3] "        ELLIPSOID[\"WGS 84\",6378137,298.257223563,"
##  [4] "            LENGTHUNIT[\"metre\",1]],"
##  [5] "    PRIMEM[\"Greenwich\",0,"
##  [6] "        ANGLEUNIT[\"degree\",0.0174532925199433]],"
##  [7] "    CS[ellipsoidal,2],"
##  [8] "        AXIS[\"geodetic latitude (Lat)\",north,"
##  [9] "            ORDER[1],"
## [10] "            ANGLEUNIT[\"degree\",0.0174532925199433]],"
## [11] "        AXIS[\"geodetic longitude (Lon)\",east,"
## [12] "            ORDER[2],"
## [13] "            ANGLEUNIT[\"degree\",0.0174532925199433]],"
```

[1] 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).

```
## [14] "    USAGE["
## [15] "        SCOPE[\"unknown\"],"
## [16] "        AREA[\"World\"],"
## [17] "        BBOX[-90,-180,90,180]],"
## [18] "    ID[\"EPSG\",4326]]"
```

Notice, among the others, the attributes `AREA` (the area of usage for the CRS) and `ID` (in this case, the EPSG code).

WKT is not very nice for humans; use the extra argument `proj = TRUE` to see the PROJ4 format of the CRS.

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

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

2.3.3 Resolution

Resolution applies only to rasters, as geometries are geometric representation of structure and can be scaled at any level. To get the resolution of a raster, use `res()`.

```
r <- rast("data/landcover-2015.tif")
res(r)
```

```
## [1] 10 10
```

The unit of the output of `res()` is the same as the unit of the CRS, in this case meters:

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

```
## [1] "+proj=laea +lat_0=52 +lon_0=10 +x_0=4321000 +y_0=3210000 +ellps=GRS80 +units=m +no_defs"
```

2.4 Data Conversion

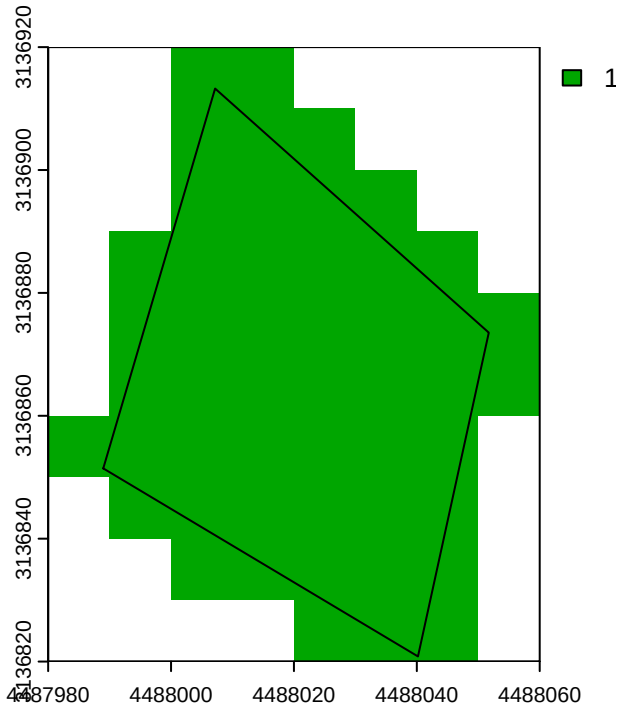
2.4.1 Geometries to Rasters

To convert geometries to raster, use `rasterize()`. In addition to the geometry to rasterize, you need to pass a raster to function as template, i.e. from which extent, CRS, and resolution are extracted from.

```
r <- rast("data/landcover-2015.tif")
idiv <- project(idiv, crs(r))
idiv_r <- rasterize(idiv, r, touches = TRUE)
```

I specified `touches = TRUE` assign a value of 1 to all cells that are touched by the polygon. The extent of this new raster is the same as the landcover one, which is too big to actually see the raterized polygon. Use `trim()` to trim the raster to the smallest raster containing all values that are not `NA`.

```
plot(trim(idiv_r))
lines(idiv)
```

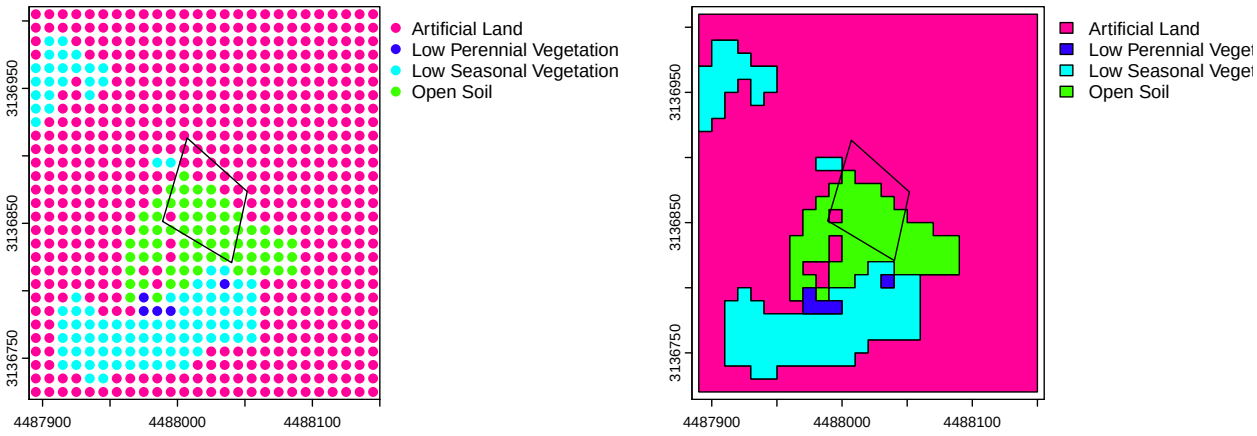


2.4.2 Rasters to Geometries

To convert rasters to geometries, use `as.points()` or `as.polygons()`.

```
r <- rast("data/landcover-2015.tif")
r <- crop(r, buffer(idiv, 1e2))  # restrict the area to something that can be plotted
poi <- as.points(r)
pol <- as.polygons(r)
```

```
op <- par(no.readonly = TRUE)
par(mfrow = c(1, 2))
plot(poi, "category")
lines(idiv)
plot(pol, "category")
lines(idiv)
```




```
par(op)
```

3 Reprojecting

It is quite common that data are obtained from a source that uses a CRS that is not ideal for analyses. It is even more common that data are gathered from multiple sources that do not use the same CRS. In such cases, the spatial data should be reprojected to one common CRS that is ideal for analyses.

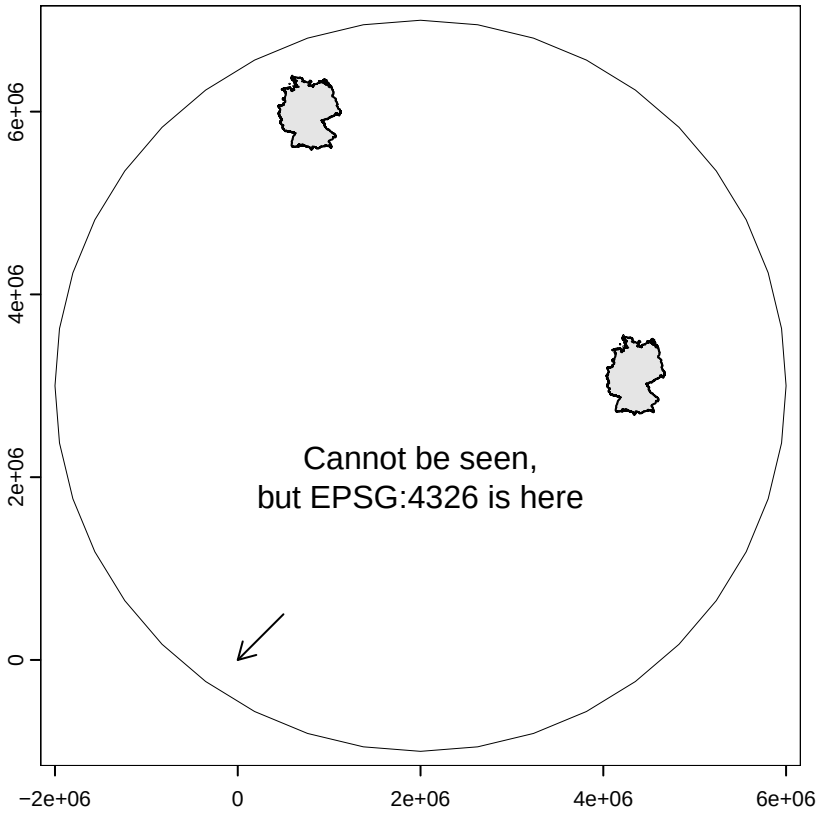
In `terra`, use `project()`, which can take three types of arguments:

1. A spatial object with known CRS.
2. A string with the WKT or PROJ4 CRS.
3. A code of a known CRS, e.g. from the EPSG standard.

```
r <- rast("data/landcover-2015.tif")
germany <- vect("data/germany.shp")

germany <- project(germany, crs(r))  # a spatial object
germany_mollweide <- project(germany, "+proj=moll")  # a proj4
germany_4326 <- project(germany, "EPSG:4326")  # a CRS code
```

CRS have a big influence on analyses and visualization.

```
plot(buffer(vect(cbind(2e6, 3e6)), 4e6), col = "white", lw = 1e-6)
polys(germany, col = "grey90")
polys(germany_mollweide, col = "grey90")
polys(germany_4326, col = "grey90")
l <- vect(
  rbind(
    cbind(5e5, 5e5),
    cbind(0, 0)
  )
)
lines(l[1], l[2], arrows = TRUE, length = .1)
l <- vect(
  rbind(
    cbind(2e6, 2e6),
    cbind(0, 0)
  )
)
text(l[1], labels = "Cannot be seen,\nbut EPSG:4326 is here")
```



4 Geometry Operations

Commonly operations on geometries are to calculate their length and area, find their centroids, calculate distance, and buffering them.

4.1 Length

Use `perim()` to obtain the length of lines or the perimeter of polygons.

```
perim(idiv)
```

```
## [1] 237.7479
```

```
idiv |> disagg(segments = TRUE) |> perim() |> sum()  # disagg(segments = TRUE) split the polygon into l
```

```
## [1] 237.7479
```

The units are defined by the CRS.

It is best to project the geometry to a longitude/latitude CRS to get more accurate results.

```
idiv |> project("EPSG:4326") |> perim()  # difference of 3 mm
```

```
## [1] 237.7449
```

4.2 Area

Use `expanse()` to obtain the area of polygons.

```
expanse(idiv)
```

```
## [1] 3264.621
```

The units are defined by the CRS.

It is best to project the geometry to a longitude/latitude CRS to get more accurate results.

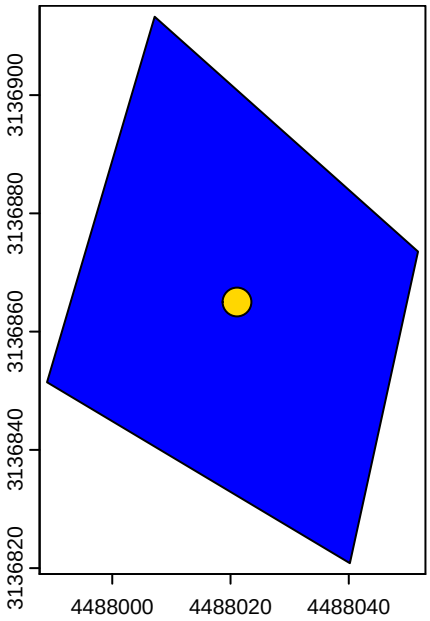
```
idiv |> project("EPSG:4326") |> expanse()  # same result, as CRS was equal-area
```

```
## [1] 3264.621
```

4.3 Centroid

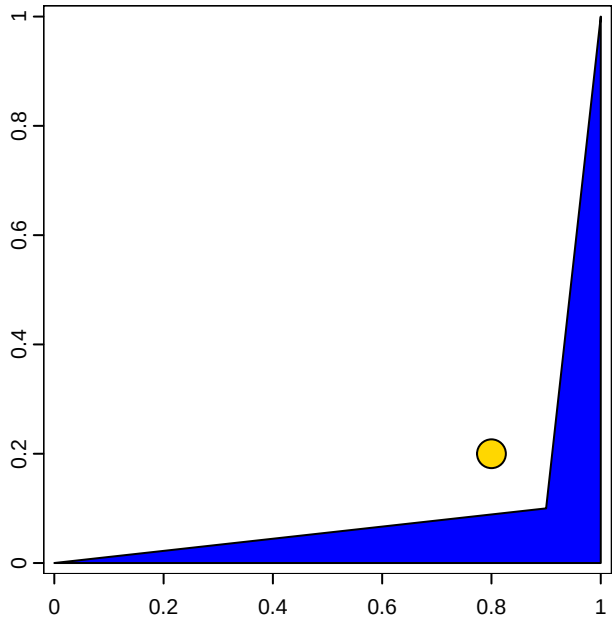
The centroid of a geometry is the point defined as the arithmetic mean position of all the points on the surface of the geometry. Use `centroids()` to get the centroids of the geometries.

```
centr <- centroids(idiv)
plot(idiv, col = "blue")
points(centr, bg = "gold", cex = 2, pch = 21)
```



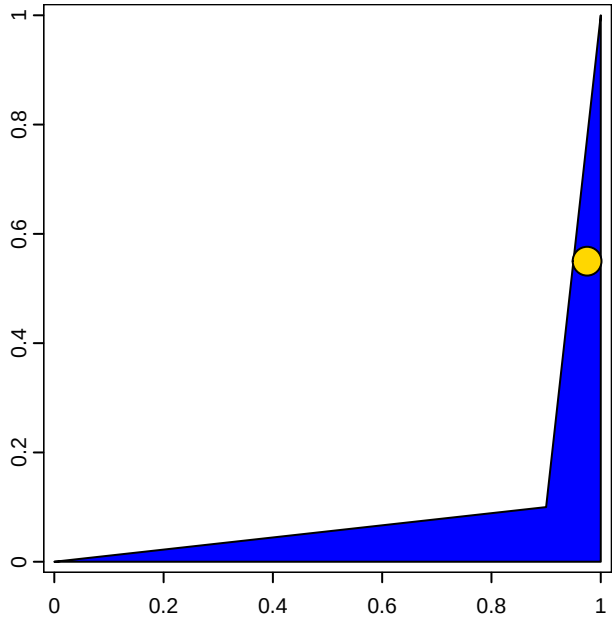
For concave polygons, the centroid may lay outside of the polygon itself

```
p <- vect(matrix(c(0, 0, 1, 0, 1, 1, 0.9, 0.1, 0, 0), byrow = TRUE, ncol = 2)) |>
  as.lines() |>
  as.polygons()
centr <- centroids(p)
plot(p, col = "blue")
points(centr, bg = "gold", cex = 2, pch = 21)
```



Specify `centroids(inside = TRUE)` to force the "centroid" to be inside the polygon.

```
p <- vect(matrix(c(0, 0, 1, 0, 1, 1, 0.9, 0.1, 0, 0), byrow = TRUE, ncol = 2)) |>
  as.lines() |>
  as.polygons()
centr <- centroids(p, inside = TRUE)
plot(p, col = "blue")
points(centr, bg = "gold", cex = 2, pch = 21)
```



4.4 Distance

Use `distance()` to get the distances between geometries When passing only an object will multiple geometries, distances will be calculated among each geometry and a (symmetric) matrix is returned.

```
poi <- vect(matrix(rnorm(10), ncol = 2), crs = "EPSG:4326")  # 5 random points
poi |> distance() |> as.matrix()
```

```
##           1        2         3        4        5
## 1      0.00 162509.0  78513.15 111138.5 244810.4
## 2 162508.97      0.0 174344.52 228341.3 255957.0
## 3  78513.15 174344.5      0.00 187665.6 168139.6
## 4 111138.54 228341.3 187665.60      0.0 355445.3
## 5 244810.36 255957.0 168139.62 355445.3      0.0
```

When passing also a second geometry, distances will be calculated among each geometry of the first object and each geometry of the second object.

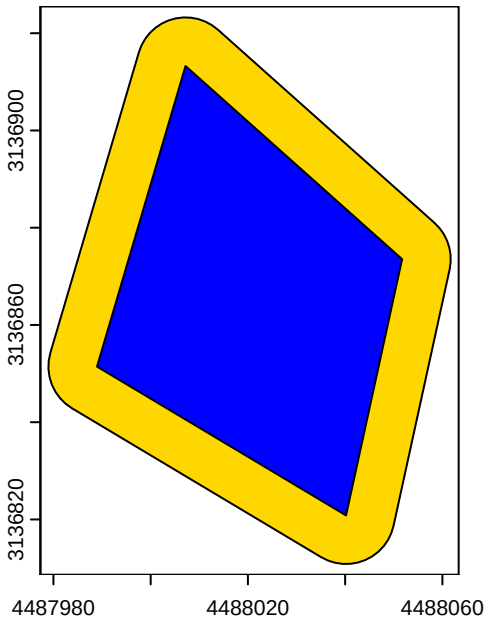
```
distance(
  poi,
  idiv |> project("EPSG:4326") |> centroids()
)
```

```
##         [,1]
## [1,] 5781295
## [2,] 5786858
## [3,] 5703259
## [4,] 5883611
## [5,] 5546948
```

4.5 Buffer

The buffer of a geometry is obtained by extending the geometry perpendicular to the tangent line of its side. It is easier to see it than to explain it. Use `buffer()` to buffer geometries.

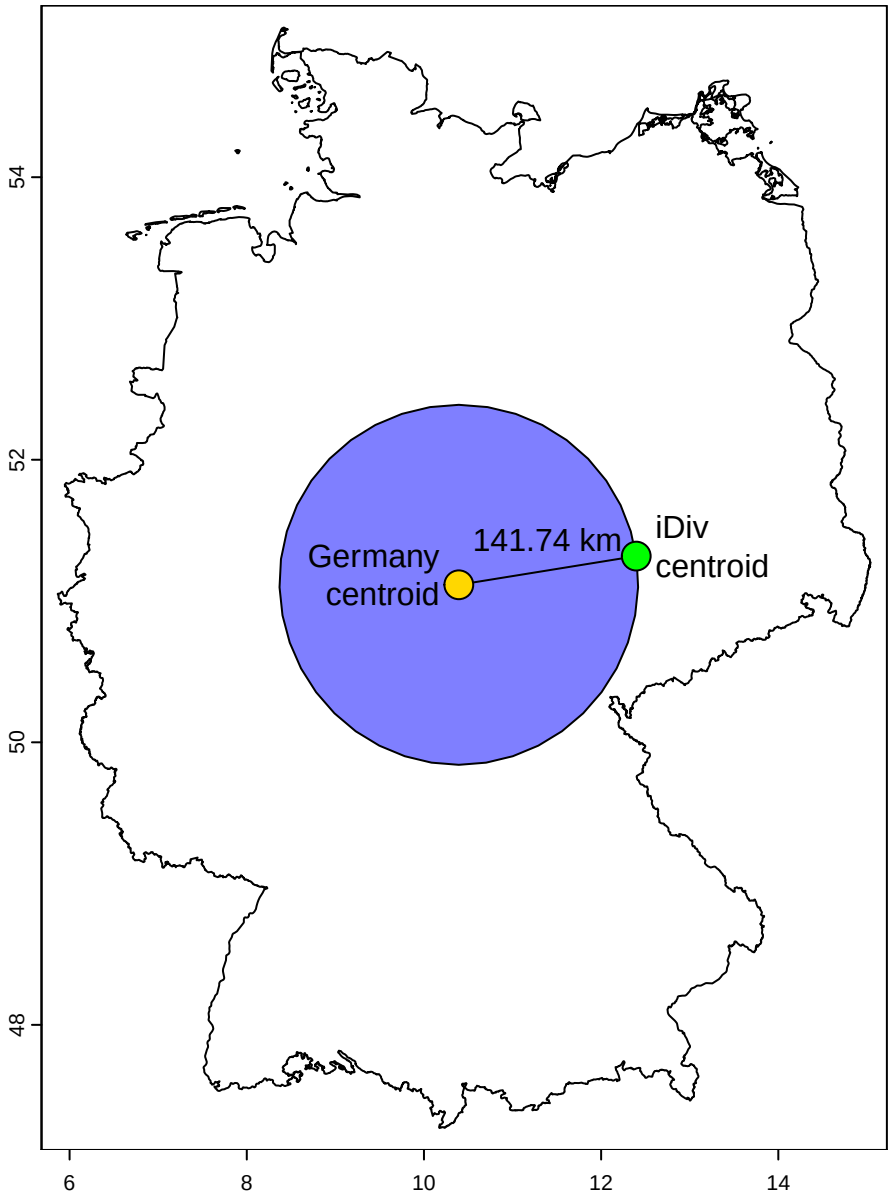
```
b <- buffer(idiv, 10)  # 10 meters
plot(b, col = "gold")
polys(idiv, col = "blue")
```



4.6 Example

Let us see an example where all operation are combined. I want to find the centroid of Germany, how far is from the centroid of the main iDiv bulding, and plot the circle centered at the centroid with the smallest area that is touching the centroid of the iDiv building.

```
ger <- vect("data/germany.shp")
centr <- centroids(ger)  # centroid of Germany
idiv_centr <- idiv |> project(crs(centr)) |> centroids()
d <- distance(centr, idiv_centr)  # distance between centroids
b <- buffer(centr, d)  # circle with the smallest area touching iDiv
l <- as.lines(rbind(centr, idiv_centr))  # line connecting centroids

plot(ger)
polys(b, col = adjustcolor("blue", alpha.f = .5))
lines(l)
points(centr, pch = 21, cex = 2, bg = "gold")
points(idiv_centr, pch = 21, cex = 2, bg = "green")
text(centr, "Germany\ncentroid", pos = 2)
text(idiv_centr, "iDiv\ncentroid", pos = 4, adj = 0.5)
text(l, paste(round(d) / 1e3, "km"), pos = 3)
```




```
message("Distance between Germany centroid and iDiv: ", round(d / 1e3), " km")
```

```
## Distance between Germany centroid and iDiv: 142 km
```

```
message("Area of the smallest circle centered at the centroid of Germany and touching iDiv: ", round(ex
```

```
## Area of the smallest circle centered at the centroid of Germany and touching iDiv: 62854 km^2
```

5 Raster Operations

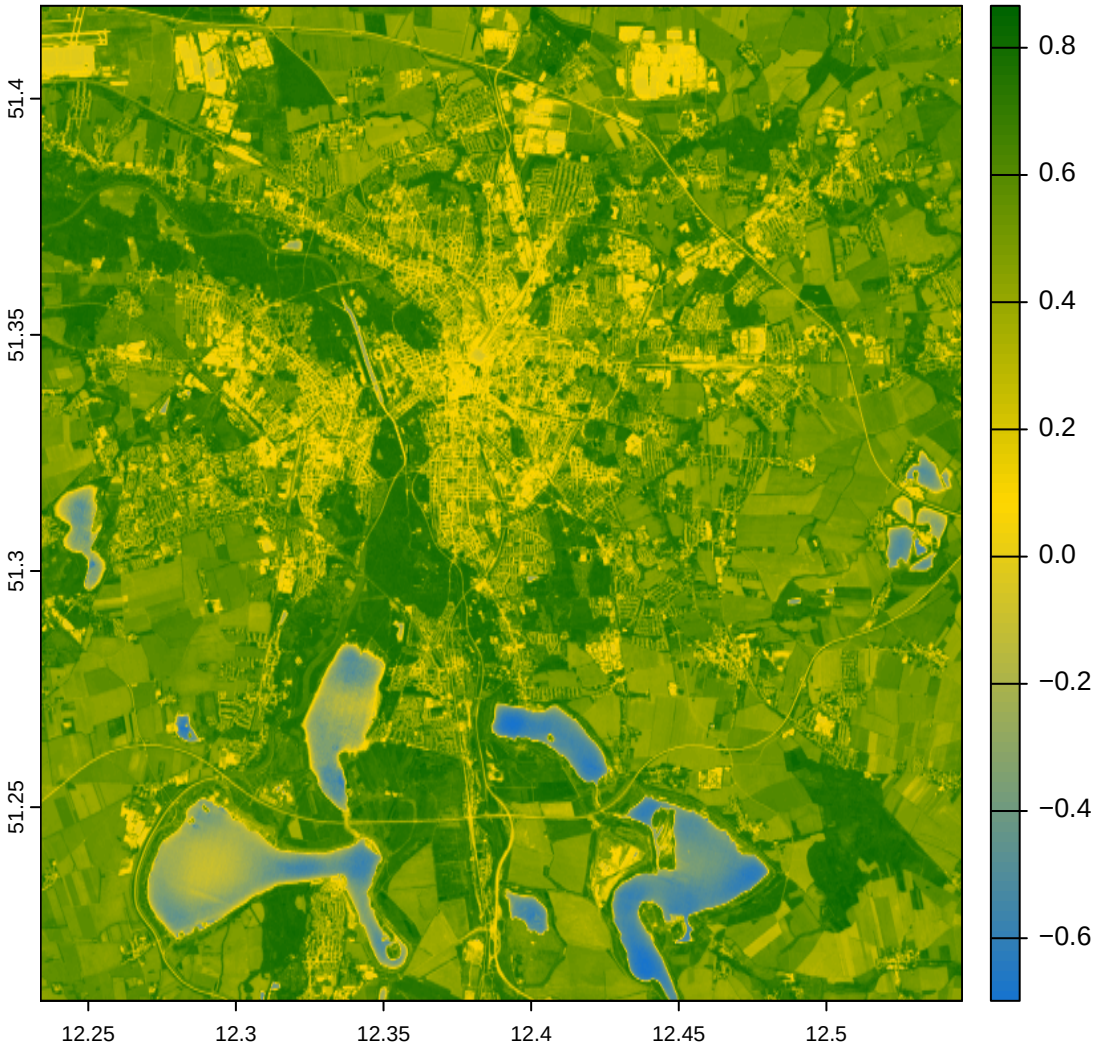
Commonly operations on rasters are to calculate summary statistics, aggregate of disaggregate to different resolutions, resampling, and interpolation.

5.1 Summary Statistics

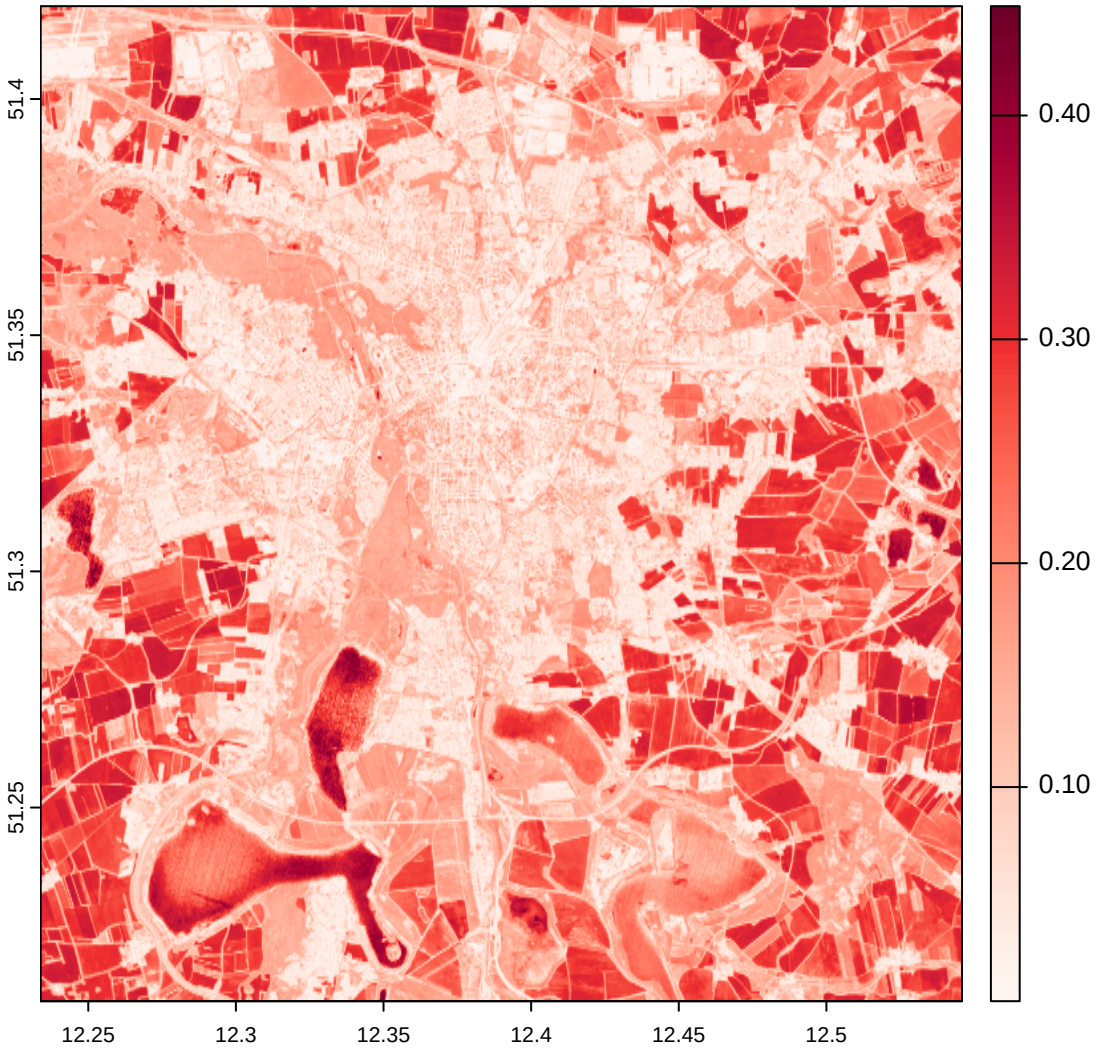
5.1.1 Local Summary

Raster stacks can be summarized using common function, such as `mean`, `sd`, `sum`, etc. The output is a single raster layer with the summary. Mean, sum, minimum, and maximum, can be calculated simply using `mean()`, `sum()`, etc. For other function, use `app()`.

```
ndvi <- rast("data/ndvi-2024.tif")
avg_ndvi <- mean(ndvi, na.rm = TRUE)  # equivalent to app(ndvi, "mean")
std_ndvi <- app(ndvi, "sd", na.rm = TRUE)
month_max_ndvi <- app(ndvi, "which.max", na.rm = TRUE)  # month with the maximum NDVI

plot(avg_ndvi, col = colorRampPalette(c("dodgerblue3", "gold", "darkgreen"))(100))
```




```
plot(std_ndvi, col = hcl.colors(100, "Reds", rev = TRUE))
```



This is called a local summary, because it is calculated across pixels with the same coordinates.

5.1.2 Global Summary

To get global summaries, i.e. calculated across all pixels of a layer, use `global()`

```
global(ndvi, "mean", na.rm = TRUE)
```

5.1.3 Zonal Summary

Zonal statistics summarize the values of a raster using categories from another raster of geometry. Use `zonal()` to obtain zonal statistics.

```
landcover <- rast("data/landcover-2015.tif")
avg_ndvi <- project(avg_ndvi, landcover)  # this will also resample avg_ndvi (see the chapter Raster Op
zonal(avg_ndvi, r, fun = "mean", na.rm = TRUE)
```

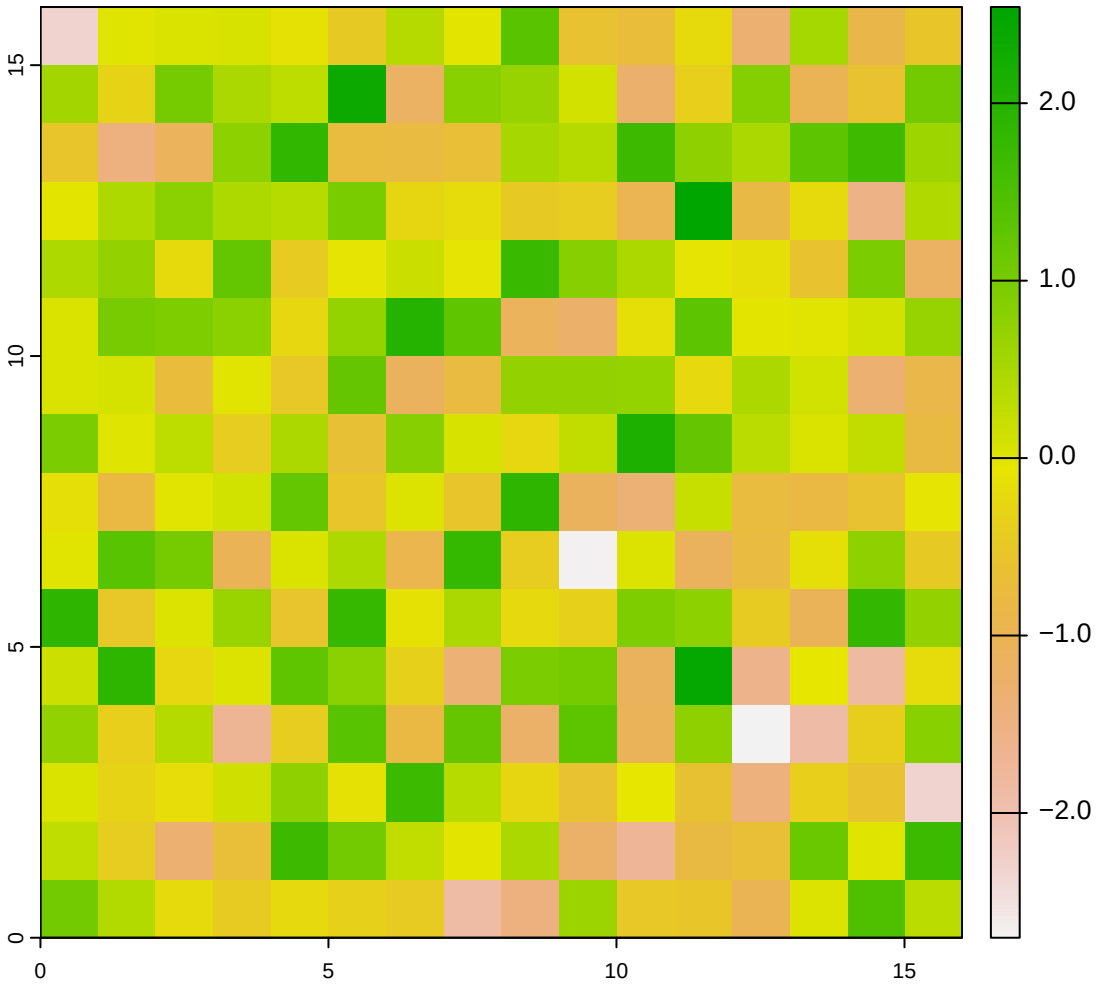
```
zonal(avg_ndvi, r, fun = "sd", na.rm = TRUE)  # spatial sd
```

5.2 Aggregate and Disaggregate

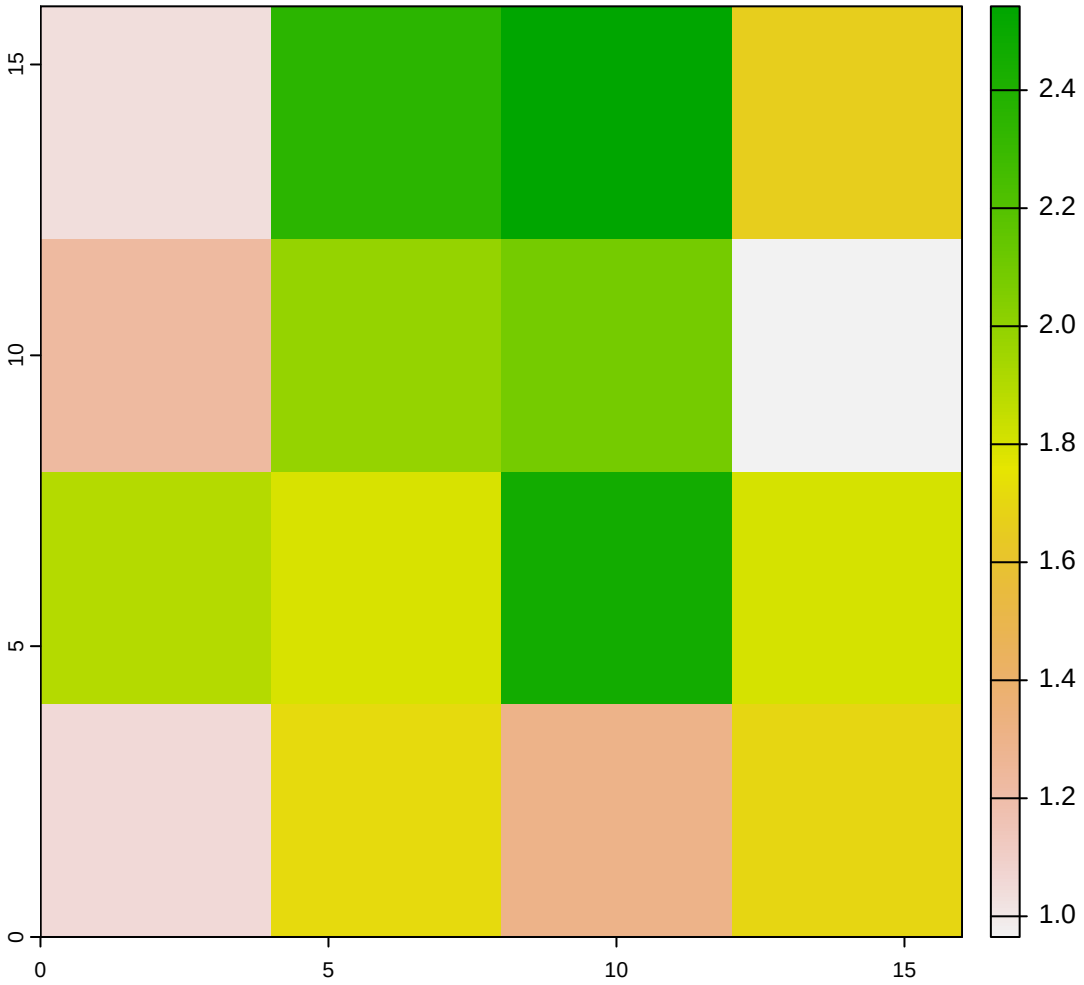
5.2.1 Aggregate

Aggregating creates a raster that is at coarser resolution compared to the original raster. Use `aggregate()` to aggregate rasters. The arguments `fact`, number of cells in each direction to be aggregated, and `fun`, the function used for aggregatation, need to be specified.

```
r <- rast(matrix(rnorm(16^2), 16, 16))
plot(r)
```



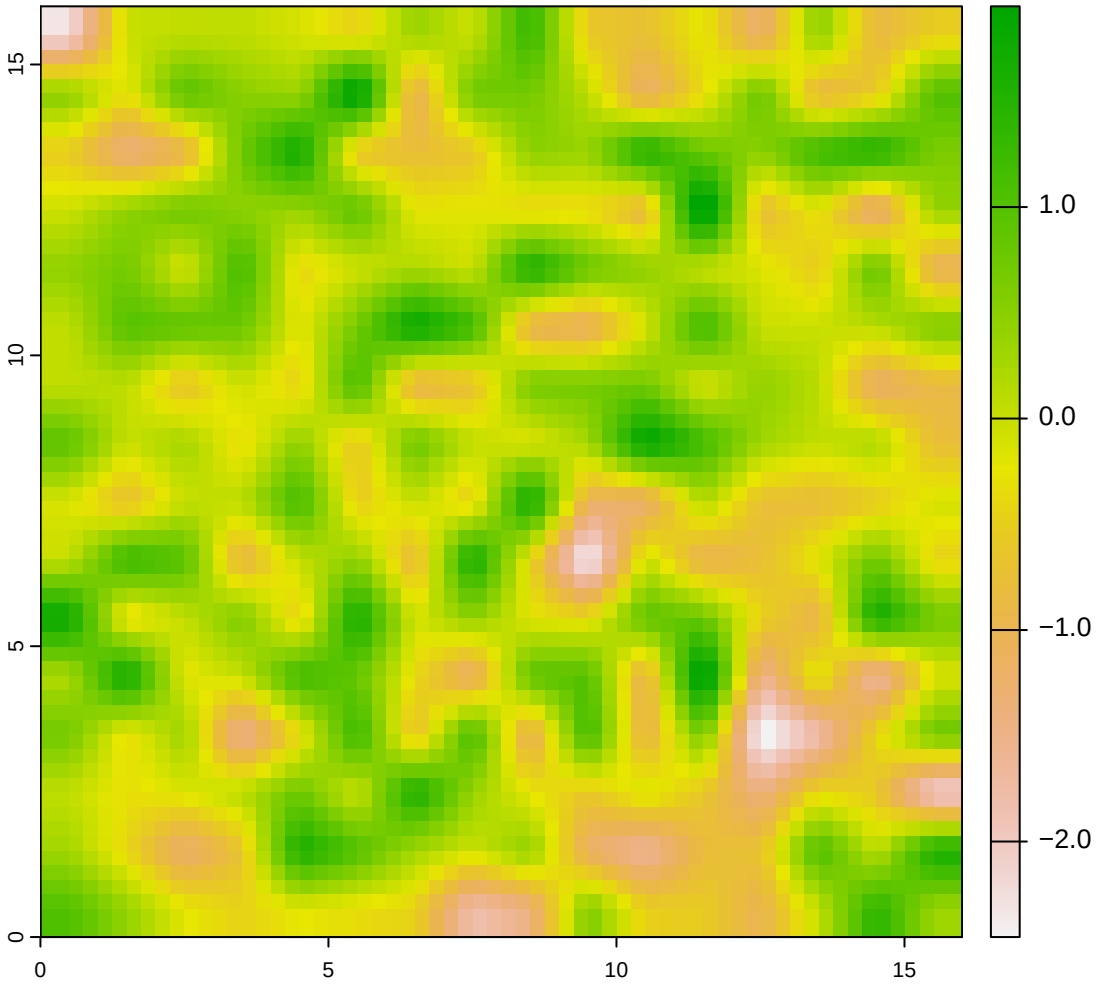

```
r_aggr <- aggregate(r, 4, "max")  # maximum values in the 4x4 grid
plot(r_aggr)
```



5.2.2 Disaggregate

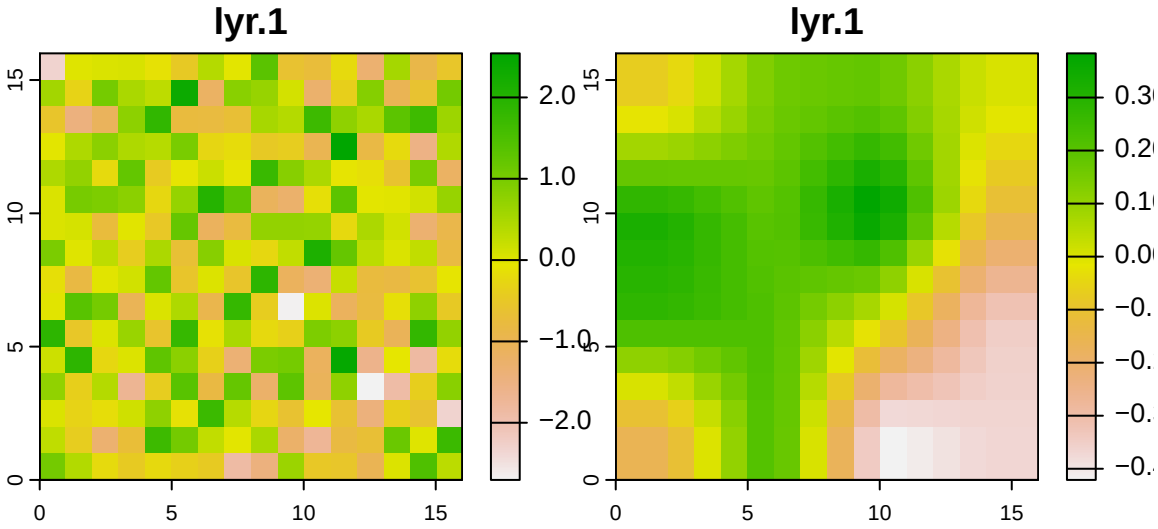
Disaggregating creates a raster that is at finer resolution compared to the original raster. Use `disagg()` to disaggregate rasters. The arguments `fact`, number of cells in each direction to create for each cell of the original raster, and `method`, the method used for disaggregation, need to be specified. Available methods are `near`, for nearest neighbor, or `bilinear`, for bilinear interpolation.

```
r_disagg <- disagg(r, 4, method = "bilinear")
plot(r_disagg)
```



Aggregating and disaggregating a raster does not give, in general, the same raster back. Some of the original information is lost during aggregation.

```
plot(c(r, r |> aggregate(4, "mean") |> disagg(4, "bilinear")))
```

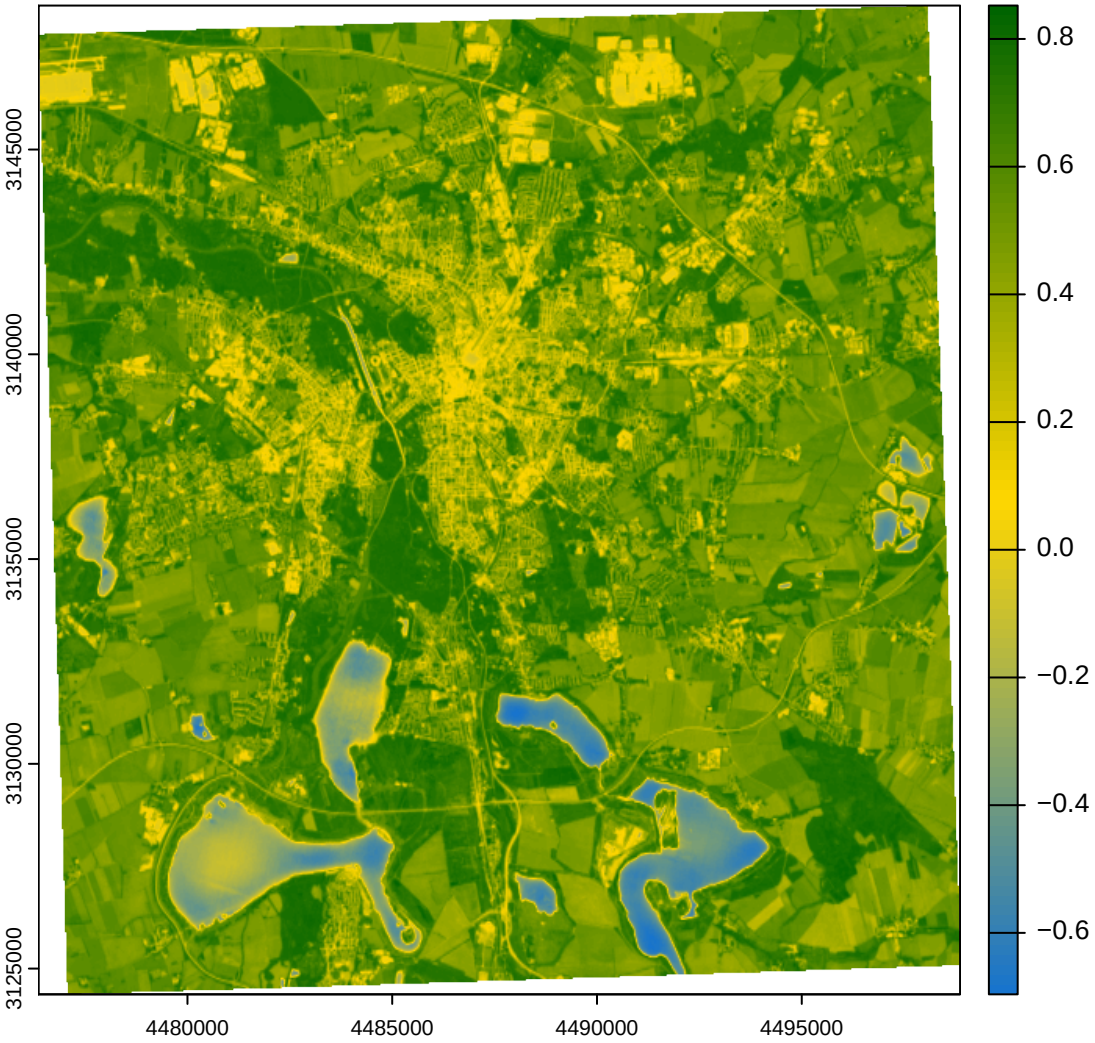


5.3 Resampling

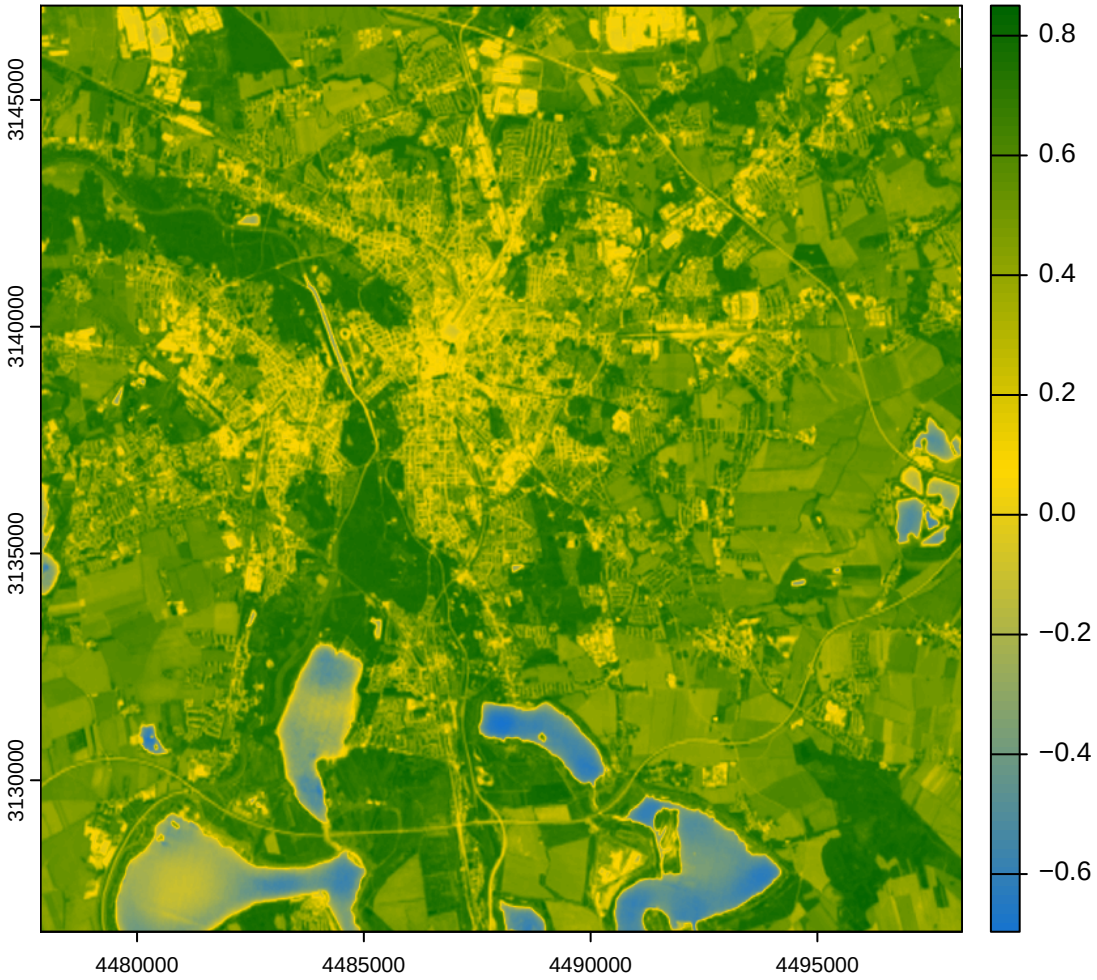
When two rasters do not align, i.e. they have different origin or resolution, to make them comparable one of them must be resampled. Resampling transform one of the two rasters to a raster that align with the other. Use `resample()` to resample rasters. The argument `method` is used to specify the algorithm used for resampling. Among the available algorithms are:

- `near`, for nearest neighbor.
- `bilinear`, for bilinear interpolation.
- `cubic`, for cubic interpolation.
- `cubicspline`, for cubic-spline interpolation.
- `lanczos`, for Lanczos resampling.

```
landcover <- rast("data/landcover-2015.tif")
ndvi <- mean(rast("data/ndvi-2024.tif"), na.rm = TRUE)
ndvi <- project(ndvi, crs(landcover))
ndvi_resampled <- resample(ndvi, landcover, method = "lanczos")
plot(ndvi, col = colorRampPalette(c("dodgerblue3", "gold", "darkgreen"))(100))
```




```
plot(ndvi_resampled, col = colorRampPalette(c("dodgerblue3", "gold", "darkgreen"))(100))
```



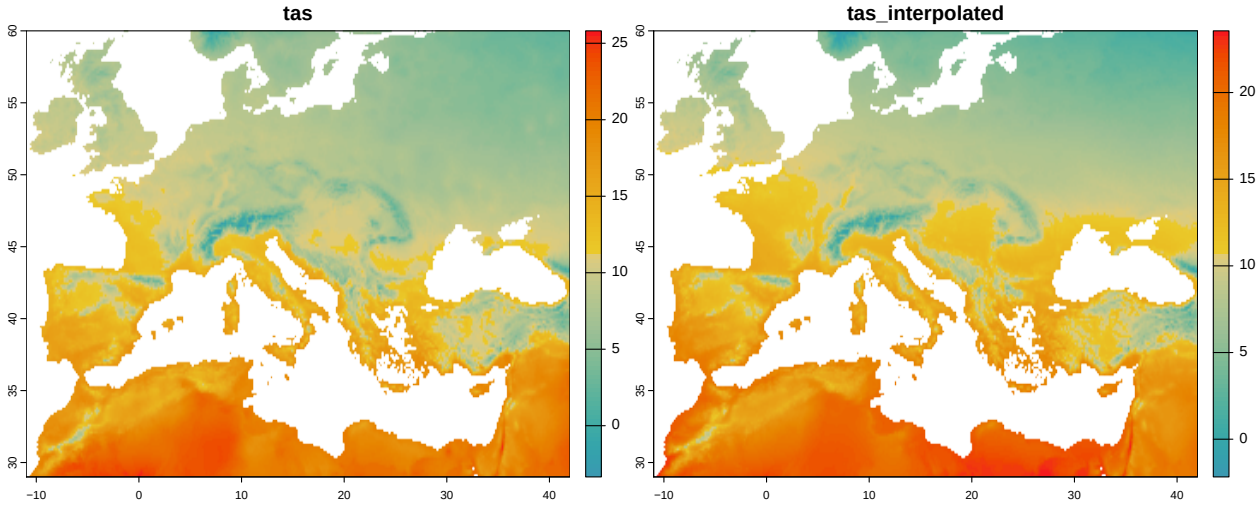
The difference between `project(x, crs(y))` and `project(x, y)` is that the first only project the CRS, whereas the second actually resample `x` to `y`. In many cases `resample(x, y, method = <method>)` can be avoided by using `project(x, y, method = <method>)`

5.4 Interpolation

Interpolation uses a statistical model to make geographic predictions. A simple statistical model can include, for example, the coordinates of the grid. This works relatively well with processes that change gradually with longitude or latitude, such as temperature.

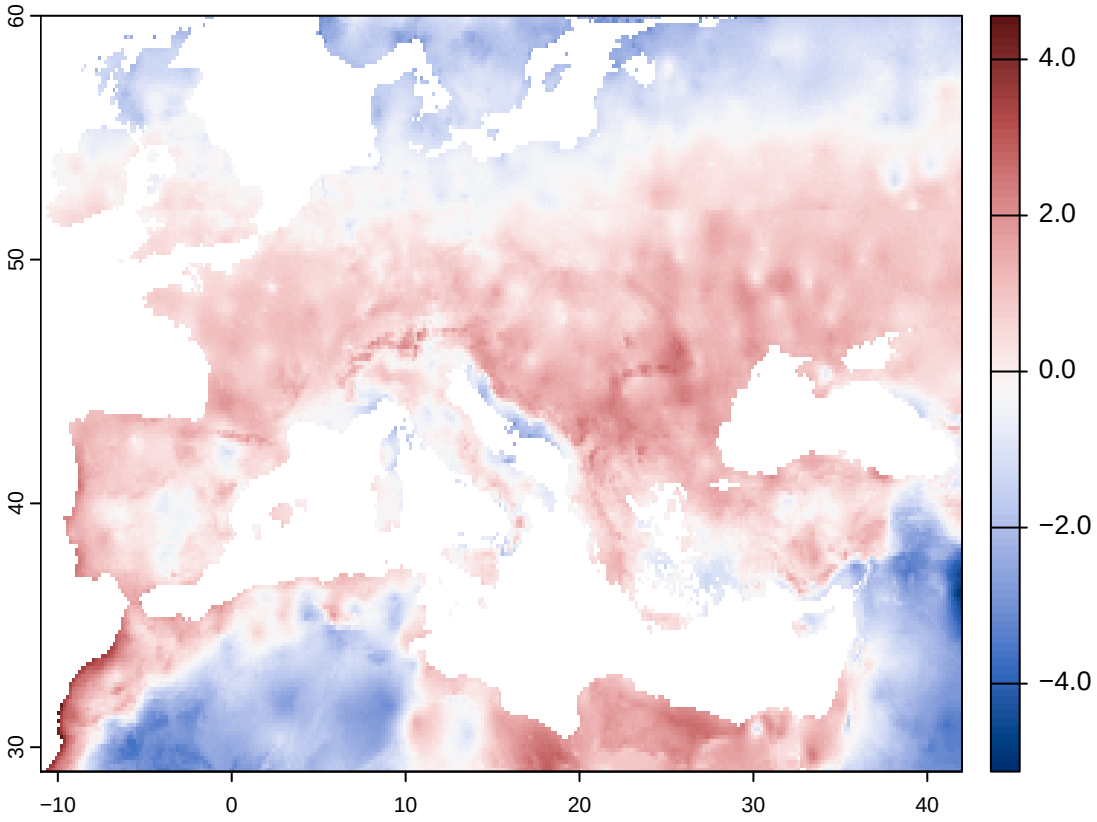
```
# average annual temperature
tas <- rast("data/wc2-bio1.tif")
elev <- rast("data/wc2-elevation.tif")
r <- c(tas, elev)

# construct table for model fitting
xy <- xyFromCell(r, 1:ncell(r))
xy <- cbind(xy, extract(r, xy, cells = TRUE))

# fit a simple linear model with coordinates
model <- lm(tas ~ x * y + elevation, data = xy)

# interpolate
```

```
interpolated <- interpolate(r, model)
interpolated <- mask(interpolated, tas)  # remove sea and large water bodies
names(interpolated) <- "tas_interpolated"
plot(c(tas, interpolated), col = hcl.colors(100, "Zissou 1"))
```



Considering the simplicity of the underlying model, the interpolation works quite well.

```
# Prediction errors
# In red, over-predictions
# In blue, under-predictions
plot(interpolated - tas, col = hcl.colors(100, "Blue-Red 3"))
```



6 Examples

6.1 Species Distribution Model (SDM)

I will show how to perform a simple species distribution model (SDM) for the species *Erica arborea*.

I load the libraries.

```
library(tibble)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:terra':
##
##     intersect, union
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(readr)
library(terra)
```

I load the file downloaded from GBIF and convert it to a geometry (points).

```
gbif <- read_tsv("data/erica-arborea-gbif.tsv") |>
  select("decimalLongitude", "decimalLatitude") |>
  vect(
    geom = c("decimalLongitude", "decimalLatitude"),
    crs = "+proj=longlat +datum=WGS84"  # I know this is GBIF CRS
  )
```

```
## Warning: One or more parsing issues, call `problems()` on your data frame for details, e.g.:
##   dat <- vroom(...)
##   problems(dat)
```

```
## Rows: 13228 Columns: 50
## -- Column specification --------------------------------------------------------
## Delimiter: "\t"
## chr  (31): datasetKey, occurrenceID, kingdom, phylum, class, order, family, ...
## dbl  (13): gbifID, individualCount, decimalLatitude, decimalLongitude, coord...
## lgl   (4): infraspecificEpithet, depth, depthAccuracy, typeStatus
## dttm  (2): dateIdentified, lastInterpreted
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

I load the bioclimatic variables from WorldClim.

```
bios <- rast("data/bios.tif")

# I also create a template of the land
land <- bios[[1]]
```

```
land[!is.na(land)] <- 1
names(land) <- "land"
```

I thin the GBIF data retaining only one point per grid cell.

```
gbif <- extract(land, gbif, cells = TRUE, xy = TRUE) |>
  as_tibble() |>
  filter(!is.na(land)) |>
  group_by(cell) |>
  slice_sample(n = 1) |>
  ungroup() |>
  select("x", "y") |>
  vect(geom = c("x", "y"), crs = "+proj=longlat +datum=WGS84")
```

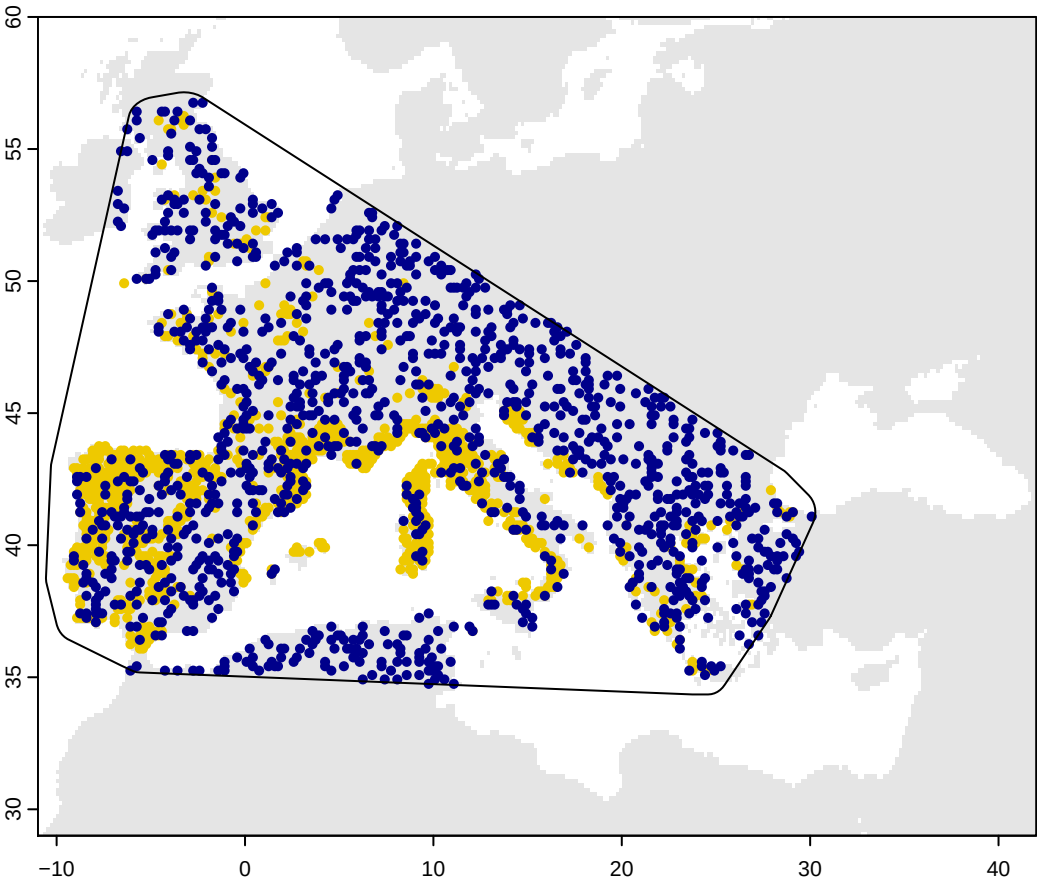
I sample pseudo-absences randomly in the polygon that contains all GBIF observation, buffered to 100 km.

```
range <- gbif |> convHull()
abs <- spatSample(buffer(range, 1e5), length(gbif) * 2)
abs <- extract(land, abs, cells = TRUE, xy = TRUE) |>
  as_tibble() |>
  filter(!is.na(land)) |>
  group_by(cell) |>
  slice_sample(n = 1) |>
  ungroup() |>
  select("x", "y") |>
  slice_sample(n = length(gbif)) |>
  vect(geom = c("x", "y"), crs = "+proj=longlat +datum=WGS84")
```

I prepare the data frame to be used for fitting the SDM.

```
gbif$occ <- 1
abs$occ <- 0
p <- rbind(gbif, abs)
d <- extract(bios, p, ID = FALSE) |> as_tibble()
d
```

```
plot(land, col = "grey90", legend = FALSE)
points(p, col = ifelse(p$occ == 0, "darkblue", "gold2"))
lines(buffer(range, 1e5))
```



I run the SDM, in this case a simple GLM. I use only some of the bioclimatic variables[2].

```
d <- d[, c("BIO01", "BIO04", "BIO12", "BIO15")]
d$occ <- p$occ
sdm <- glm(occ ~ ., data = d, family = "binomial")
summary(sdm)
```

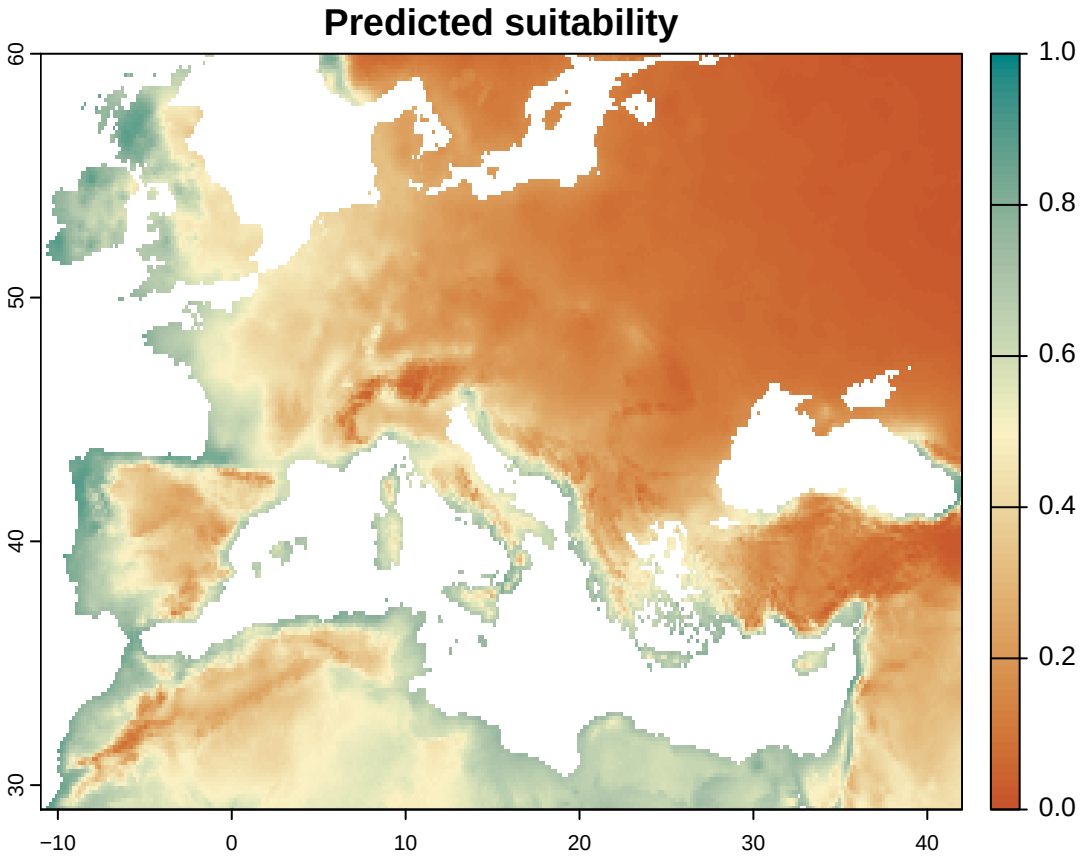
```
##
## Call:
## glm(formula = occ ~ ., family = "binomial", data = d)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1210  -1.0562   0.0479   1.0801   2.2245
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.0168285  0.4980212  -2.042   0.0412 *
## BIO01        0.2135554  0.0233451   9.148  < 2e-16 ***
## BIO04       -0.0041811  0.0004876  -8.574  < 2e-16 ***
## BIO12        0.0012585  0.0001937   6.497  8.2e-11 ***
## BIO15       -0.0048010  0.0035160  -1.365   0.1721
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

[2]This is just an example, and you should do better than this.

```
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 3410.3  on 2459  degrees of freedom
## Residual deviance: 3104.3  on 2455  degrees of freedom
## AIC: 3114.3
##
## Number of Fisher Scoring iterations: 4
```

I use the model to predict the suitability of the species in geographic space.

```
suitability <- predict(bios, sdm, type = "response")
plot(
  suitability,
  col = hcl.colors(100, "Geyser", rev = TRUE),
  breaks = seq(0, 1, by = .02),
  type = "continuous",
  main = "Predicted suitability"
)
```



Zooming on the Mediterranean regions.

```
plot(
  crop(suitability, ext(-9, 24, 35, 50)),
  col = hcl.colors(100, "Geyser", rev = TRUE),
  breaks = seq(0, 1, by = .02),
  type = "continuous",
  main = "Predicted suitability"
)
```

```
points(p[p$occ == 1, ], cex = .3)
```