

2 Getting started

In this session I aim to give you a general introduction to the course and how we developed it, some technical aspects about how we organize the course using git and gitHub. Then we will have a closer look at how to deal with spatial data in R.

Learning objectives after this session you should know

- how to use RStudio project.
- how to clone a git repository and update it.
- some familiarity with the packages **sf** and **terra** in R.

i Preparation before the session

Please make sure you do the following the before the session:

1. Make sure R, RStudio and git is installed on your laptop.
2. *Optionally create an account at GitHub*
3. Read: Chapters 2.2 - 2.4 and 4 from Geocomputation with R <https://r.geocomp.org/>.
4. Read the basic idea of git and gitHub <https://swcarpentry.github.io/git-novice/01-basics.html>.

2.1 RStudio Projects

Wrong paths often lead to unnecessary problems. An easy way to avoid problems with a wrong path is to use **RStudio projects**.

To create a project:

1. Press the button shown in 1) or go to **File > New Project**
2. Select **New directory** and then choose **New Project**.
3. Enter the name of the project (as shown in 4)) and the choose where on your computer you want to place the project. A new folder with the project name will be created. If you want to create a project for this course (which I would encourage you to do), then you want to place this probably inside a folder where you have all your university courses organized.

4. Finally, press on **Create Project**.

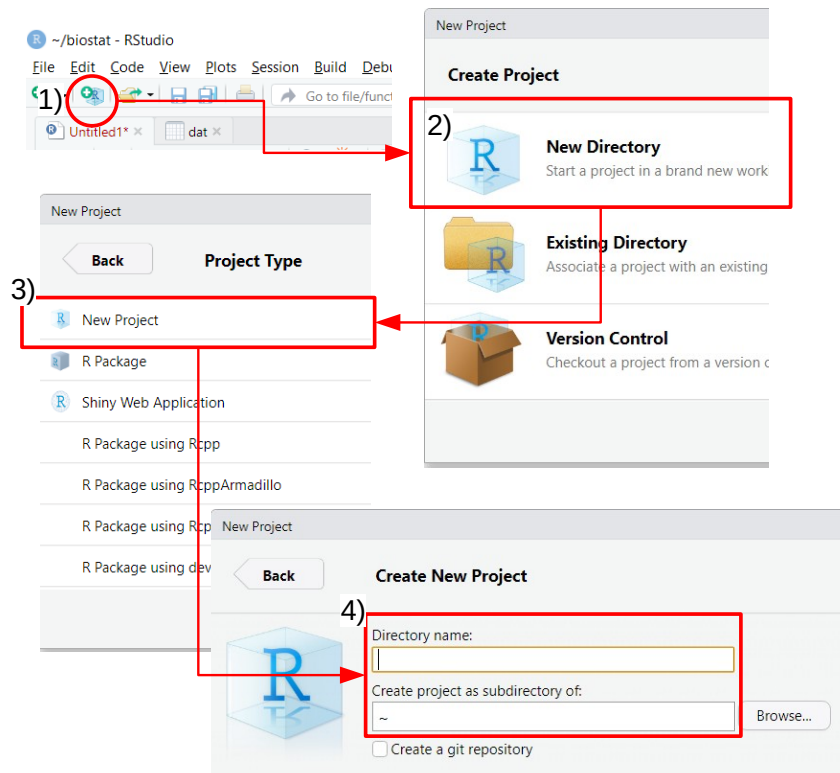


Figure 2.1: Workflow to create a new RStudio project.

- A nice read: <https://www.tidyverse.org/articles/2017/12/workflow-vs-script/>
- A project has the advantage, that all paths are set with reference to the project location.
- It is a good idea, to have a directory for data in each project.

Once you created a project, there are three ways to open it:

- Double click on the project icon in the directory:

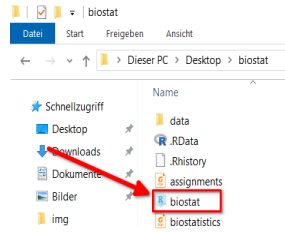


Figure 2.2: How to open an RStudio project.

Alternatively, it is possible to choose **File > Open Project** in RStudio or to select the name of a recent project on the top right in RStudio.

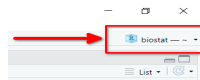


Figure 2.3: Selecting the name of a recent project.

🔥 Exercise 1: RStudio

1. Create an RStudio Project for this course or clone everything from GitHub.
2. Make sure project clones into this directory.
3. Read the data set `data/dbh_1.csv`, using only **relative** paths.

3 Spatial Data

3.1 Working with vector data

- Spatial data are omnipresent.
- Basically spatial data are regular data with coordinates (so we can reference data points in space).

3.1.1 What is a GIS

A geographic information system (GIS) is a system designed to capture, store, manipulate, analyze, manage, and present all types of spatial or geographical data ([wikipedia](#)).

3.1.2 Components of spatial data

1. Geometries
2. Attributes data

The OGC (open geospatial consortium) simple feature (SF) specification defines the following building blocks:

- Points (e.g., location of a tree),
- Lines (e.g., a road),
- Polygons (e.g., a lake).

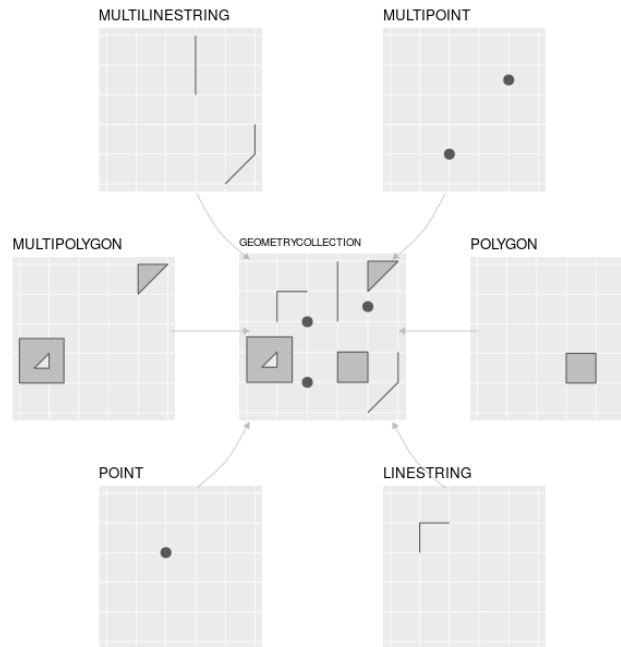


Figure 3.1: From Lovelace et al. 2021

3.1.3 Attribute data

Table with measured quantities of each subject (we often use the term *feature* in GIS).

E.g., if we were to describe the largest cities of the EU, we would obtain an attribute table like [this](#):

Rank	City	State	Official population	Date of assessment	Reference	Image
1	London	United Kingdom	8,615,246	1 June 2014	[1]	
2	Berlin	Germany	3,517,424	30 December 2010	[2]	
3	Madrid	Spain	3,168,235	1 January 2014	[3]	

Figure 3.2: Wikipedia

This data do not tell us anything of the geometries.

3.1.4 What is so special about spatial data

- Normal data with coordinates with reference to some known fixed location:

- The earth
- A country (e.g., with reference to the capital)
- A building (e.g., room map)

We often work with spatial data, that can be attributed to a unique location on the earth surface.

3.1.5 The earth is complicated



Figure 3.3: Source www.icsm.gov.au

We can start with a **Geoid**, that is the hypothetical surface of the oceans if there was no land and only gravitational forces.



Figure 3.4: Source www.icsm.gov.au

3.1.6 Ellipsoid (= spheroid) and Datum

A geometrical approximation of the geoid, that is used to describe the sphere of the earth (we often use WGS84).

This is generally referred to as a *Datum*.

3.1.7 What is a projection

A datum is used to reference a location on an ellipsoid with:

- longitude
- latitude
- altitude

To *project* a sphere from 3D to 2D we need a set of mathematical rules, these are known as *projections*.

- The coordinate reference system (CRS) of spatial data can be referred to with a EPSG code.
- See here for a list of EPSG codes: epsg.io

3.2 Vector data in R

- Traditionally the **sp** package provided classes to work with spatial data.
- The **sp** package is replaced by the **sf** package.
- We will be using the **sf** package most of the time.

We can create simple features with `st_point()`, `st_linestring()` and `st_polygon()`. This creates a simple feature geometry (`sfg-Object`)

```
library(sf)
p1 <- st_point(c(9.23, 52.3))
p1
```

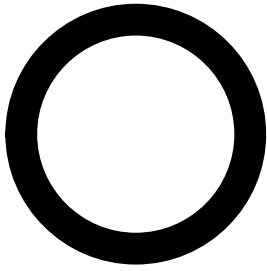
```
POINT (9.23 52.3)
```

```
l1 <- st_linestring(cbind(1:10, c(1:3, 3:1, 1:4)))
l1
```

```
LINESTRING (1 1, 2 2, 3 3, 4 3, 5 2, 6 1, 7 1, 8 2, 9 3, 10 4)
```

```
po1 <- st_polygon(list(cbind(c(1, 2, 3, 4, 1),
                             c(0, 10, 10, 0, 0))))
```

```
plot(p1)
```



Lets define the position of three trees

```
t1 <- st_point(c(103, 83))
t2 <- st_point(c(33, 73))
t3 <- st_point(c(103, 130))

t1
```

POINT (103 83)

```
t2
```

POINT (33 73)

```
t3
```

POINT (103 130)

We can now take these three trees and create a simple feature column (**sfc**-object in R) with the function `st_sfc`.

```
c1 <- st_sfc(t1, t2, t3)
c1
```

Geometry set for 3 features

Geometry type: POINT

Dimension: XY

Bounding box: xmin: 33 ymin: 73 xmax: 103 ymax: 130

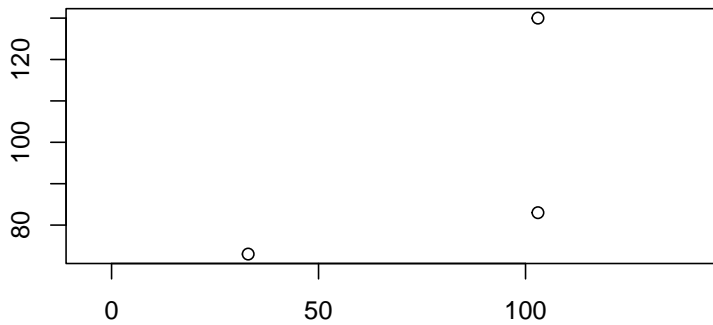
CRS: NA

POINT (103 83)

POINT (33 73)

POINT (103 130)

```
plot(c1, axes = TRUE)
```



Finally, we can combine the geometry with attribute data. So far we only focused on the geometries.

```
a1 <- data.frame(  
  z = c(150, 110, 167),  
  sp = c("oak", "beech", "oak")  
)  
a1
```

	z	sp
1	150	oak
2	110	beech
3	167	oak

To do this, we call the function `st_sf()`.

```
sf1 <- st_sf(a1, geometry = c1)  
sf1
```

Simple feature collection with 3 features and 2 fields

Geometry type: POINT

Dimension: XY

Bounding box: xmin: 33 ymin: 73 xmax: 103 ymax: 130

CRS: NA

z	sp	geometry
---	----	----------

```

1 150   oak POINT (103 83)
2 110 beech POINT (33 73)
3 167   oak POINT (103 130)

```

The data.frame `sf1` behaves like any other data.frame.

```
sf1[1, ]
```

```

Simple feature collection with 1 feature and 2 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 103 ymin: 83 xmax: 103 ymax: 83
CRS:          NA
   z  sp      geometry
1 150 oak POINT (103 83)

```

```
sf1 |> filter(sp == "oak")
```

```

Simple feature collection with 2 features and 2 fields
Geometry type: POINT
Dimension:     XY
Bounding box:  xmin: 103 ymin: 83 xmax: 103 ymax: 130
CRS:          NA
   z  sp      geometry
1 150 oak POINT (103 83)
2 167 oak POINT (103 130)

```

Now we can work with the spatial data, and for example add a buffer

```
st_buffer(sf1, dist = 10)
```

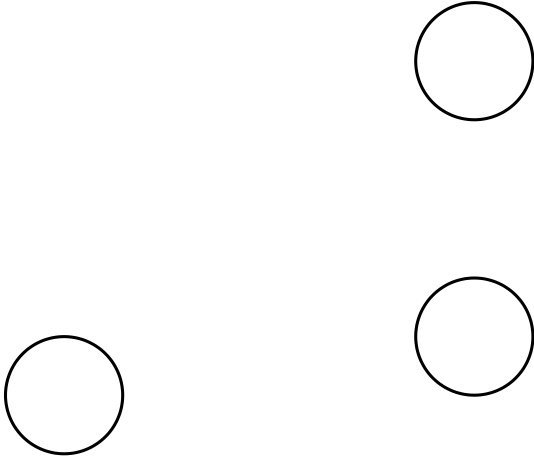
```

Simple feature collection with 3 features and 2 fields
Geometry type: POLYGON
Dimension:     XY
Bounding box:  xmin: 23 ymin: 63 xmax: 113 ymax: 140
CRS:          NA
   z  sp      geometry
1 150 oak POLYGON ((113 83, 112.9863 ...
2 110 beech POLYGON ((43 73, 42.9863 72...
3 167 oak POLYGON ((113 130, 112.9863...

```

sf is compliant with the tidyverse philosophy

```
sf1 <- sf1 |> mutate(buffer = st_buffer(geometry, dist = 10))
plot(sf1$buffer)
```



3.2.1 Creating spatial points from a data frame

Of course there is an easier way to create `st_points` from a `data_frame`

```
sta <- read.csv(here::here("data/weather/feb2015.csv"))
head(sta, 2)
```

	X	id	Feb1	Feb2	Feb3	Feb4	Feb5	Feb6	Feb7	Feb8	Feb9	Feb10	Feb11	Feb12	Feb13
1	1	44	0.4	0.6	0.2	-1.5	-1.4	-1.7	0.6	3.4	5.5	5.4	3.8	1.6	2.4
2	2	71	-2.0	-2.7	-6.8	-6.1	-4.9	-4.9	-6.2	-4.3	-3.0	-0.4	-1.9	-2.2	-0.3
			Feb14	Feb15	Feb16	Feb17	Feb18	Feb19	Feb20	Feb21	Feb22	Feb23	Feb24	Feb25	Feb26
1			4.1	1.7	1.4	0.7	1.6	2.1	3.9	3.8	2.2	3.6	4.1	4.0	6.2
2			1.0	-2.6	-2.2	-3.6	-2.9	-0.7	2.5	1.7	-0.4	0.3	1.0	0.6	0.0
			Feb27	Feb28	lon	lat									
1			4.9	3.3	8.2370	52.9335									
2			0.5	0.1	8.9784	48.2155									

```
sta <- st_as_sf(sta, coords = c("lon", "lat"))
head(sta, 2)
```

Simple feature collection with 2 features and 30 fields

Geometry type: POINT

Dimension: XY

Bounding box: xmin: 8.237 ymin: 48.2155 xmax: 8.9784 ymax: 52.9335

CRS: NA

	X	id	Feb1	Feb2	Feb3	Feb4	Feb5	Feb6	Feb7	Feb8	Feb9	Feb10	Feb11	Feb12	Feb13
1	1	44	0.4	0.6	0.2	-1.5	-1.4	-1.7	0.6	3.4	5.5	5.4	3.8	1.6	2.4
2	2	71	-2.0	-2.7	-6.8	-6.1	-4.9	-4.9	-6.2	-4.3	-3.0	-0.4	-1.9	-2.2	-0.3
			Feb14	Feb15	Feb16	Feb17	Feb18	Feb19	Feb20	Feb21	Feb22	Feb23	Feb24	Feb25	Feb26
1			4.1	1.7	1.4	0.7	1.6	2.1	3.9	3.8	2.2	3.6	4.1	4.0	6.2
2			1.0	-2.6	-2.2	-3.6	-2.9	-0.7	2.5	1.7	-0.4	0.3	1.0	0.6	0.0
			Feb27	Feb28	geometry										
1			4.9	3.3	POINT (8.237 52.9335)										
2			0.5	0.1	POINT (8.9784 48.2155)										

3.2.2 IO of spatial data

Often we want to read and write vector data from existing file:

```
ger <- st_read(here::here("data/ger/ger_states_3035.shp"))
```

Reading layer `ger_states_3035' from data source

`/Users/jsigner/ownCloud - jsigner@uni-goettingen.de@owncloud.gwdg.de/Documents/L_Lehre/M_

using driver `ESRI Shapefile'

Simple feature collection with 16 features and 1 field

Geometry type: MULTIPOLYGON

Dimension: XY

Bounding box: xmin: 4031295 ymin: 2684102 xmax: 4672253 ymax: 3551343

Projected CRS: Lambert_Azimuthal_Equal_Area

```
ger
```

Simple feature collection with 16 features and 1 field

Geometry type: MULTIPOLYGON

Dimension: XY

Bounding box: xmin: 4031295 ymin: 2684102 xmax: 4672253 ymax: 3551343

Projected CRS: Lambert_Azimuthal_Equal_Area

First 10 features:

	state	geometry
1	Hamburg	MULTIPOLYGON (((4333619 340...
2	Niedersachsen	MULTIPOLYGON (((4109540 339...

```

3           Bremen MULTIPOLYGON (((4222786 334...
4 Nordrhein-Westfalen MULTIPOLYGON (((4230505 326...
5           Hessen MULTIPOLYGON (((4298290 316...
6 Rheinland-Pfalz MULTIPOLYGON (((4169461 308...
7 Baden-Württemberg MULTIPOLYGON (((4221076 273...
8           Bayern MULTIPOLYGON (((4355236 271...
9 Saarland MULTIPOLYGON (((4109218 294...
10          Berlin MULTIPOLYGON (((4565934 327...

```

Writing vector data

Similarly, it is possible to write geographic data from R, using the function `st_write()`.

3.2.3 Transforming the coordinate reference system (CRS)

```
st_crs(ger)
```

Coordinate Reference System:

User input: Lambert_Azimuthal_Equal_Area

wkt:

```

PROJCRS["Lambert_Azimuthal_Equal_Area",
  BASEGEOGCRS["WGS 84",
    DATUM["D_unknown",
      ELLIPSOID["WGS84",6378137,298.257223563,
        LENGTHUNIT["metre",1,
          ID["EPSG",9001]]],
      PRIMEM["Greenwich",0,
        ANGLEUNIT["Degree",0.0174532925199433]]],
    CONVERSION["unnamed",
      METHOD["Lambert Azimuthal Equal Area",
        ID["EPSG",9820]],
      PARAMETER["Latitude of natural origin",52,
        ANGLEUNIT["Degree",0.0174532925199433],
        ID["EPSG",8801]],
      PARAMETER["Longitude of natural origin",10,
        ANGLEUNIT["Degree",0.0174532925199433],
        ID["EPSG",8802]],
      PARAMETER["False easting",4321000,
        LENGTHUNIT["metre",1],
        ID["EPSG",8806]],

```

```

        PARAMETER["False northing",3210000,
                  LENGTHUNIT["metre",1],
                  ID["EPSG",8807]]],
  CS[Cartesian,2],
    AXIS["(E)",east,
        ORDER[1],
        LENGTHUNIT["metre",1,
                    ID["EPSG",9001]]],
    AXIS["(N)",north,
        ORDER[2],
        LENGTHUNIT["metre",1,
                    ID["EPSG",9001]]]]

```

We did not set CRS for `sta` when creating it. We *know* that `sta` is in geographic coordinates (i.e., EPSG = 4326)

```

st_crs(sta) <- 4326 # epsg code
st_crs(ger) == st_crs(sta)

```

```
[1] FALSE
```

In order to work with both data sets `ger` and `sta`, we have to transform one of them.

Transform to projected CRS

```

sta <- sta |> st_transform(st_crs(ger))
st_crs(ger) == st_crs(sta)

```

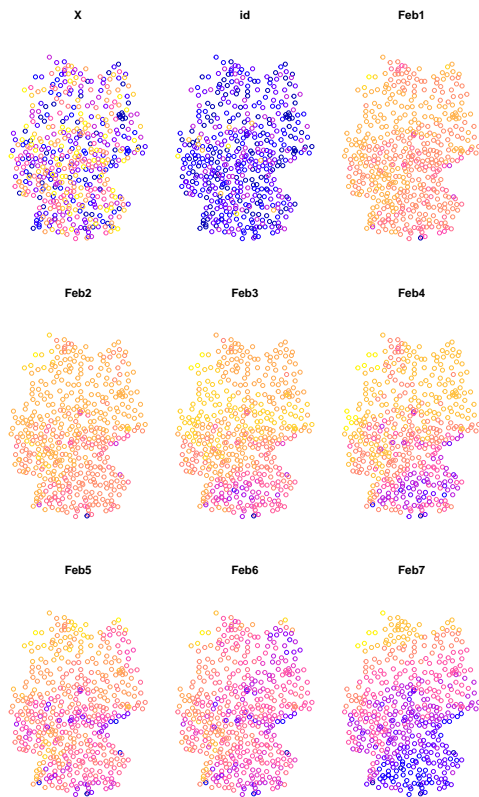
```
[1] TRUE
```

3.2.4 Plotting spatial data

The `plot` command works with `sf`-objects. By default up to 10 attributes are displayed.

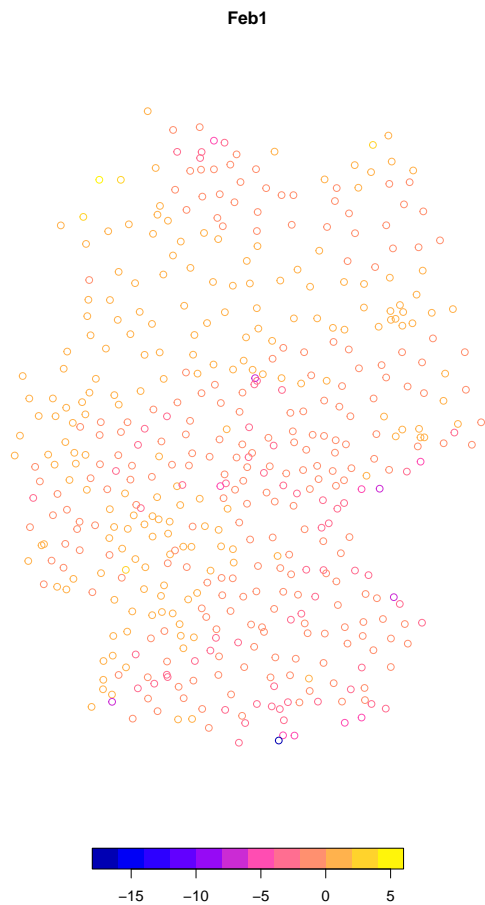
```
plot(sta)
```

Warning: plotting the first 9 out of 30 attributes; use `max.plot = 30` to plot all



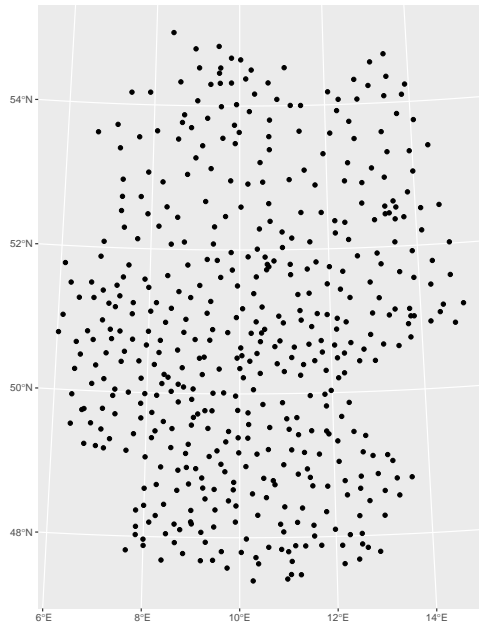
We can pick specific attributes (i.e, columns) with:

```
plot(sta[, "Feb1"])
```



`ggplot2` can be used plot spatial data. There is a `geom` for spatial data called `geom_sf()`. By default the geometry column is used (the one that is returned or set with `st_geometry()`).

```
ggplot(sta) + geom_sf()
```

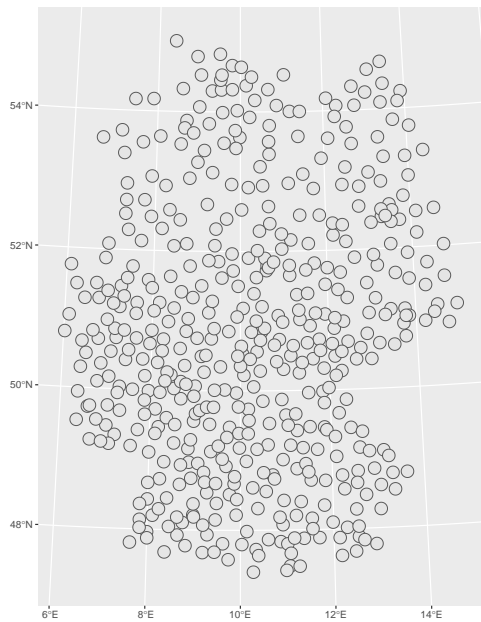



But different geometry columns can be used. We buffer each weather station with 10 km.

```
sta1 <- sta |>
  mutate(buffer = st_buffer(geometry, dist = 1e4))
```

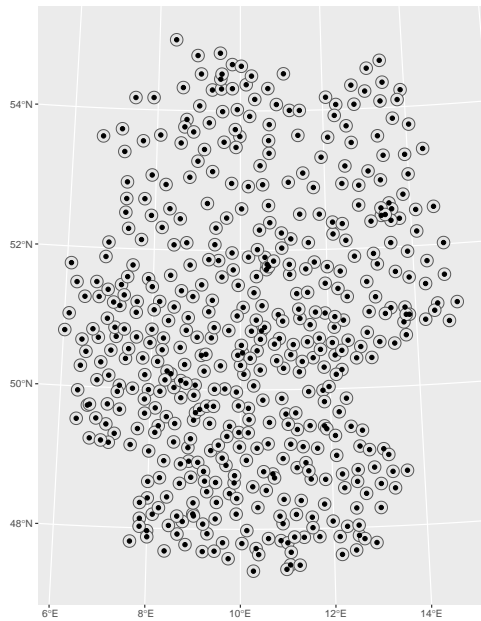
The geometry is still the column with the unbuffered points, but we can plot the buffered points by explicitly calling the new geometry.

```
ggplot(sta1) + geom_sf(aes(geometry = buffer))
```



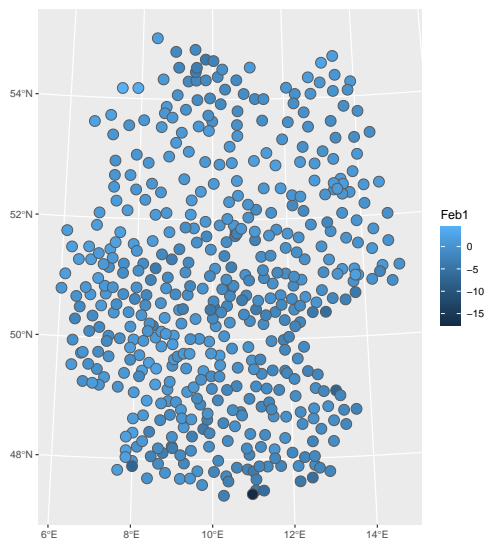
We could even plot geometries:

```
ggplot(sta1) +  
  geom_sf(aes(geometry = buffer)) +  
  geom_sf(aes())
```



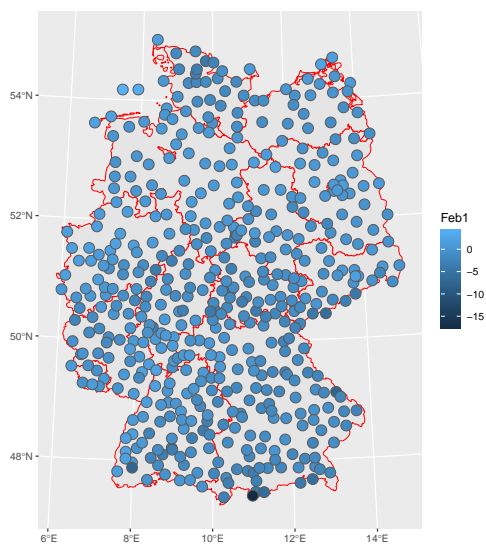
We can use aesthetics for colors and fill as with non-spatial data.

```
ggplot(sta1) +  
  geom_sf(aes(geometry = buffer, fill = Feb1))
```



We can also combine several layers:

```
ggplot(sta1) +  
  geom_sf(data = ger, col = "red") +  
  geom_sf(aes(geometry = buffer, fill = Feb1))
```

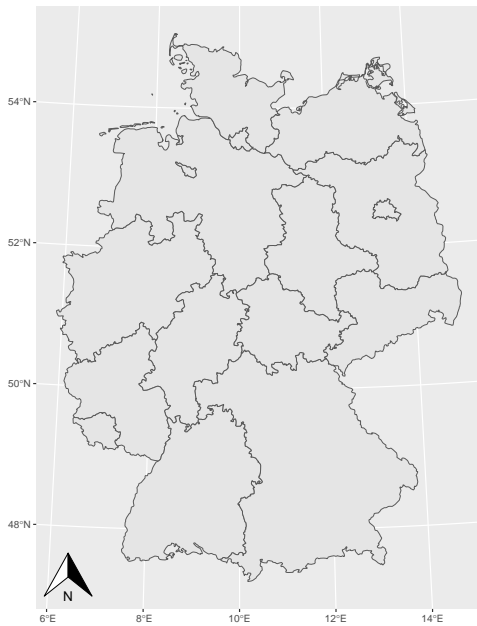


Functionally can be extended with `ggspatial`:

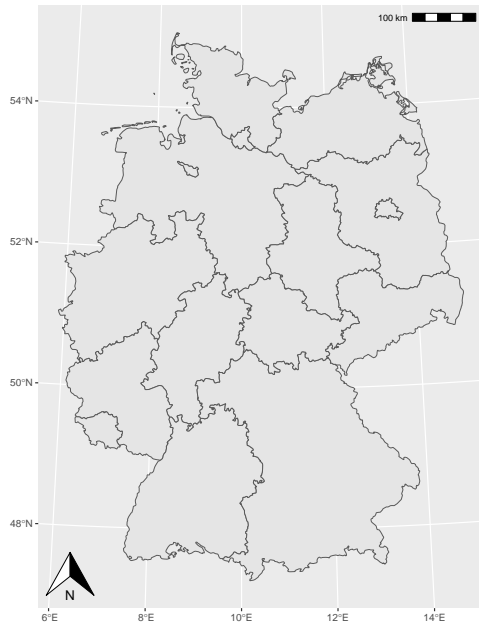
Often we need to annotate a map with north arrows and scale bars. This can be achieved with the package `ggspatial` or `ggsn`. We will use the package `ggspatial` here.

- `annotation_north_arrow()` adds a north arrow to the map.
- `annotation_scale()` adds a scale to the map.

```
library(ggspatial)
ggplot() +
  geom_sf(data = ger) +
  annotation_north_arrow()
```



```
ggplot() +
  geom_sf(data = ger) +
  annotation_north_arrow() +
  annotation_scale(location = "tr")
```



We can even add a map tile as background with the function `annotation_map_tile()`:

```
ggplot() + annotation_map_tile() +  
  geom_sf(data = ger, fill = NA)
```

Zoom: 5



- The package `tmap` offers many functions to create maps from geographic data in R. A good introduction can be found here: <https://geocompr.robinlovelace.net/adv-map.html>.
- The package `mapView` makes it very easy to create interactive maps.

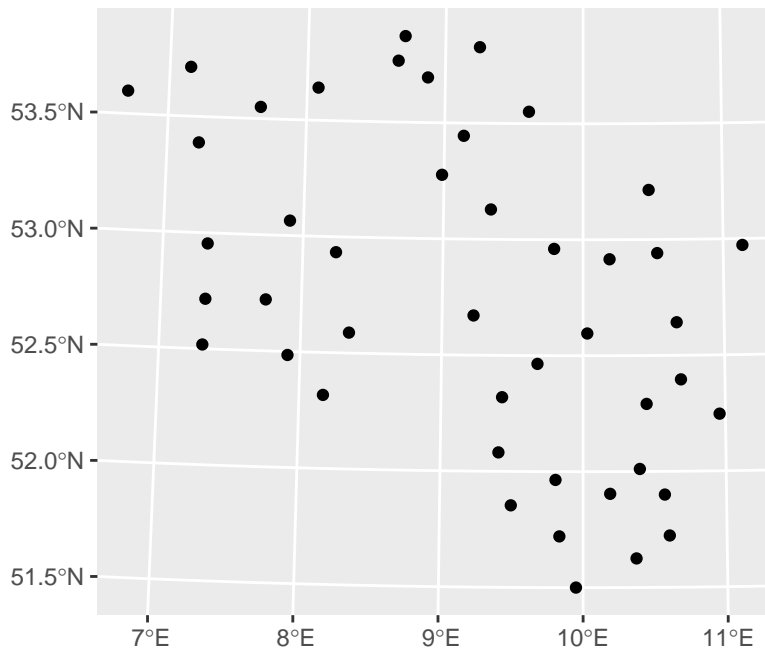
3.2.5 Spatial subsets

[works also for spatial data. E.g.,

```
nds <- filter(ger, state == "Niedersachsen")
```

Now we can get all weather stations from Lower Saxony with

```
ggplot(sta[nds, ]) + geom_sf()
```



3.2.6 Topological relations

These describe the *spatial* relationship between two or more features. Some of the most commonly used functions are:

- `st_intersects()`

- `st_disjoint()`
- `st_within()`
- `st_touches()`

All of these functions have an argument `sparse` that is by default set to `TRUE`.

For example, which weather stations are within which state?

```
st_within(sta, ger)
```

Sparse geometry binary predicate list of length 488, where the predicate was `'within'`

first 10 elements:

```
1: 2
2: 7
3: 8
4: 2
5: 5
6: 8
7: 13
8: 8
9: 6
10: 8
```

If `sparse = FALSE` we get

```
st_within(sta, ger, sparse = FALSE) |> head()
```

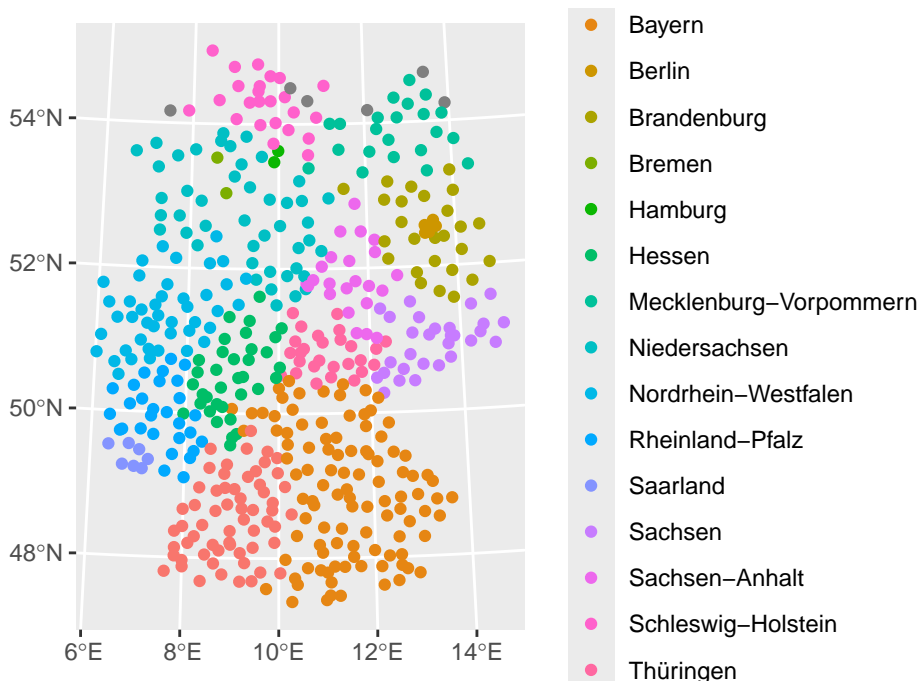
```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
[3,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
[4,] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[5,] FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[6,] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE
      [,13] [,14] [,15] [,16]
[1,] FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE
[3,] FALSE FALSE FALSE FALSE
[4,] FALSE FALSE FALSE FALSE
[5,] FALSE FALSE FALSE FALSE
[6,] FALSE FALSE FALSE FALSE
```

3.2.7 Spatial joins

Analogous to non-spatial joins, there is a function called `st_join()`, which *joins* attributes based on their locations from two data sets.

We do not know the state of each weather station in `sta`. We can get the state using a spatial join.

```
sta <- st_join(sta, ger)
ggplot(sta, aes(col = state)) + geom_sf()
```



3.3 Optional: What is GDAL?

The Geospatial Data Abstraction Library (GDAL) is a computer software library for reading and writing raster and vector geospatial data formats (e.g. shapefile), and is released under the permissive X/MIT style free software license by the Open Source Geospatial Foundation. As a library, it presents a single abstract data model to the calling application for all supported formats. It may also be built with a variety of useful command line interface utilities for data translation and processing. Projections and transformations are supported by the PROJ library. (from Wikipedia)

3.3.1 Accessing GDAL

- From command line (this can be tedious)
- Within R use the `gdalUtilities` package.
 - We can still do all the manipulation from within R.
 - **But** the data are not read into R.

```
library(terra)
```

```
terra 1.8.42
```

```
Attaching package: 'terra'
```

```
The following object is masked from 'package:tidyr':
```

```
extract
```

```
(clc <- rast(here::here("data/dem_goe1.tif")))
```

```
class       : SpatRaster
dimensions  : 1127, 1464, 1  (nrow, ncol, nlyr)
resolution  : 25, 25  (x, y)
extent      : 4297325, 4333925, 3140100, 3168275  (xmin, xmax, ymin, ymax)
coord. ref. : ETRS89_ETRS_LAEA
source      : dem_goe1.tif
name        : dem_goe1
```

```
system.time(
  clc <- project(clc, "epsg:4326")
)
```

```
user  system elapsed
0.334  0.021   0.363
```

```
The same with gdalwarp():
```

```
library(gdalUtilities)
```

Attaching package: 'gdalUtilities'

The following object is masked from 'package:sf':

gdal_rasterize

```
system.time(  
  gdalwarp(  
    srcfile = here::here("data/dem_goe1.tif"),  
    dstfile = here::here("data/dem_goe1_4326.tif"),  
    t_srs = 'EPSG:4326')  
)
```

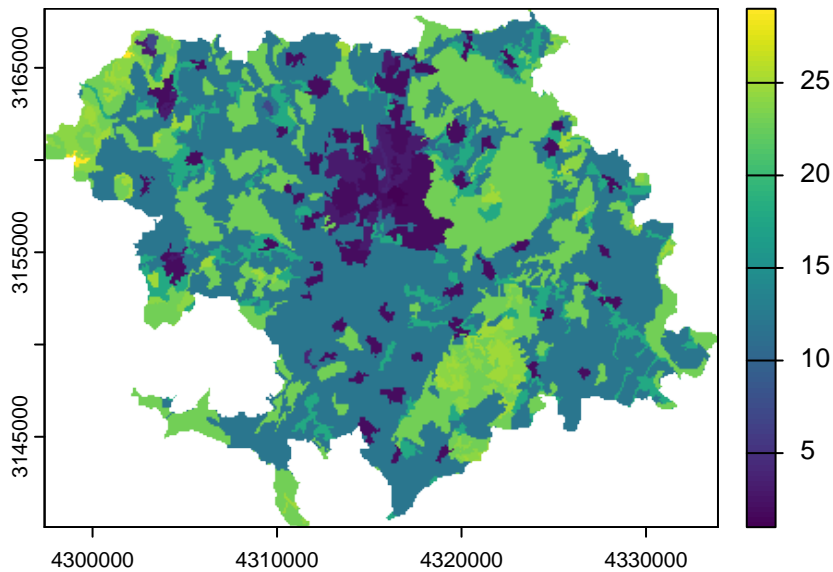
```
      user  system elapsed  
0.038   0.014   0.059
```

3.4 Cut to features

```
gdalwarp(here::here("data/clc_europe.tif"),  
  here::here("data/clc_goe.tif"),  
  ot = "Int16",  
  dstnodata = -128,  
  cutline = here::here("data/goe.gpkg"),  
  crop_to_cutline = TRUE,  
  overwrite = TRUE)
```

Warning in CPL_gdalwarp(source, destination, options, oo, doo, config_options,
: GDAL Message 1: The definition of projected CRS EPSG:3035 got from GeoTIFF
keys is not the same as the one from the EPSG registry, which may cause issues
during reprojection operations. Set GTIFF_SRS_SOURCE configuration option to
EPSG to use official parameters (overriding the ones from GeoTIFF keys), or to
GEOKEYS to use custom values from GeoTIFF keys and drop the EPSG code.

```
r1 <- rast("data/clc_goe.tif")
plot(r1)
```



🔥 Exercise 2: Vector data

Use the following data set

```
set.seed(123)

df1 <- data.frame(
  x = runif(100, 0, 100),
  y = runif(100, 0, 100),
  crown_diameter = runif(100, 1, 15),
  sp = sample(letters[1:4], 100, TRUE)
)
```

1. Use `df1` and create a geometry column.
2. Buffer each tree with its canopy radius.
3. Calculate the crown area of each tree and save it in a new column (hint, you may want to use the function `st_area()`).
4. Find the tree with the largest canopy area.
5. Find the tree with the largest canopy area for each species.

3.5 Working with raster data

The package **terra** (**raster** in the past) is dedicated to work with raster data.

```
library(terra)
r <- rast(here::here("data/raster/feb_1_2015.tif"))
r
```

```
class      : SpatRaster
dimensions : 170, 125, 1  (nrow, ncol, nlyr)
resolution : 5000, 5000  (x, y)
extent     : 4040863, 4665863, 2696181, 3546181  (xmin, xmax, ymin, ymax)
coord. ref.: ETRS89-extended / LAEA Europe (EPSG:3035)
source     : feb_1_2015.tif
name       : feb_1_2015
min value  : -15.064530
max value  :  4.273138
```

`rast()` is a very generic function to create all kind of rasters. It can also be used to create a raster from scratch:

```
x <- rast()
x
```

```
class      : SpatRaster
dimensions : 180, 360, 1  (nrow, ncol, nlyr)
resolution : 1, 1  (x, y)
extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref.: lon/lat WGS 84 (CRS84) (OGC:CRS84)
```

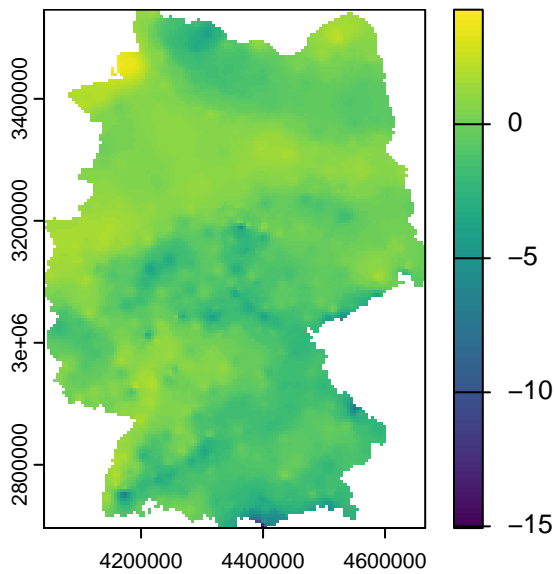
Or by defining an extent

```
x <- rast(xmin = 0, xmax = 100, ymin = 0, ymax = 100, res = 10)
x
```

```
class      : SpatRaster
dimensions : 10, 10, 1  (nrow, ncol, nlyr)
resolution : 10, 10  (x, y)
extent     : 0, 100, 0, 100  (xmin, xmax, ymin, ymax)
coord. ref.:
```

The generic `plot()` command works well for continuous raster data.

```
plot(r)
```

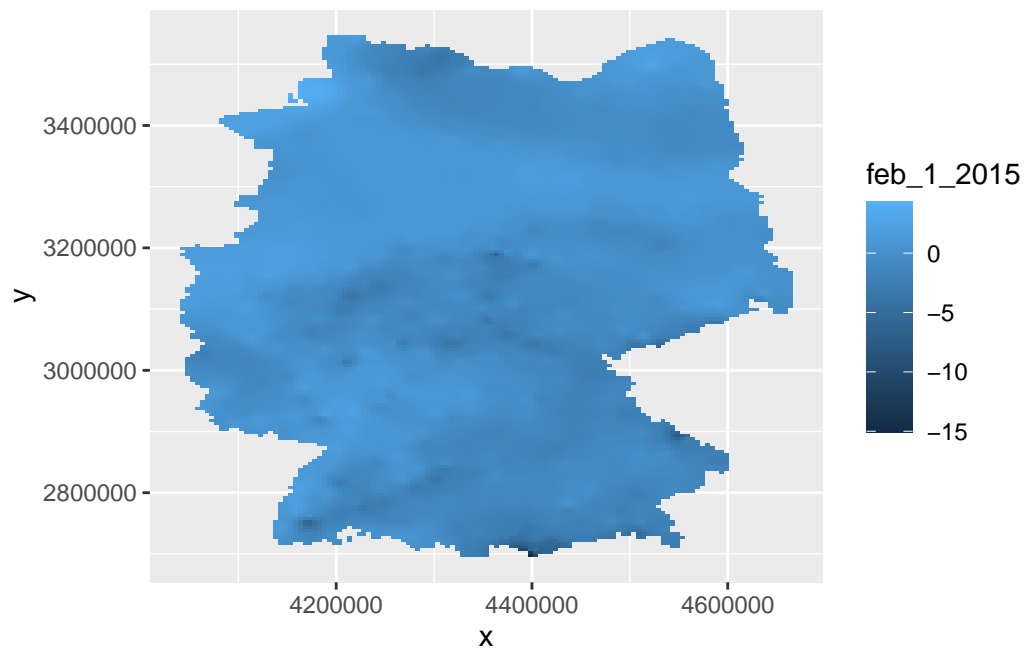


Much more flexibility to plot can be achieved with `ggplot2`. The way to plot raster data with `ggplot2` is to 1) convert the raster to a data frame (`rasterToPoints()`) and then 2) use `geom_raster()`.

```
d <- as.data.frame(r, xy = TRUE)
```

Now we can create a plot using `ggplot2` and `d`. Alternatively, `as.data.frame(r, xy = TRUE)` also works.

```
ggplot(d, aes(x, y, fill = feb_1_2015)) + geom_raster()
```



3.5.1 About a raster

We can get the resolution of a raster with the function `res()`

```
res(r)
```

```
[1] 5000 5000
```

The dimensions (i.e., the number of rows and columns) with `dim()`. Note, `nrow()` and `ncol()` also exist for raster.

```
dim(r)
```

```
[1] 170 125 1
```

The extent of a raster can be obtained with function `extent()`

```
ext(r)
```

```
SpatExtent : 4040863.15800447, 4665863.15800447, 2696180.93186771, 3546180.93186771 (xmin, xmax, ymin, ymax)
```

Finally, the CRS can be obtained with `projection()`

```
crs(r)
```

```
[1] "PROJCRS[\"ETRS89-extended / LAEA Europe\",\n      BASEGEOGCRS[\"ETRS89\",\n      ENSEMBL
```

- The projection of a raster can also be transformed using `project()`.
- Caution: if the values of the raster continuous set `method = "bilinear"` otherwise set `method = "near"` to use nearest neighbor interpolation. Other options include `cubic` and `cubicsplines`.

3.5.2 Accessing values of a raster

The functions `setValues()` and `getValues()` can be used to set or obtain the values of a raster, respectively.

```
head(values(r))
```

```
      feb_1_2015
[1,]      NaN
[2,]      NaN
[3,]      NaN
[4,]      NaN
[5,]      NaN
[6,]      NaN
```

Alternatively the `[]`-function also works.

```
head(r[])
```

```
      feb_1_2015
[1,]      NaN
[2,]      NaN
[3,]      NaN
[4,]      NaN
[5,]      NaN
[6,]      NaN
```

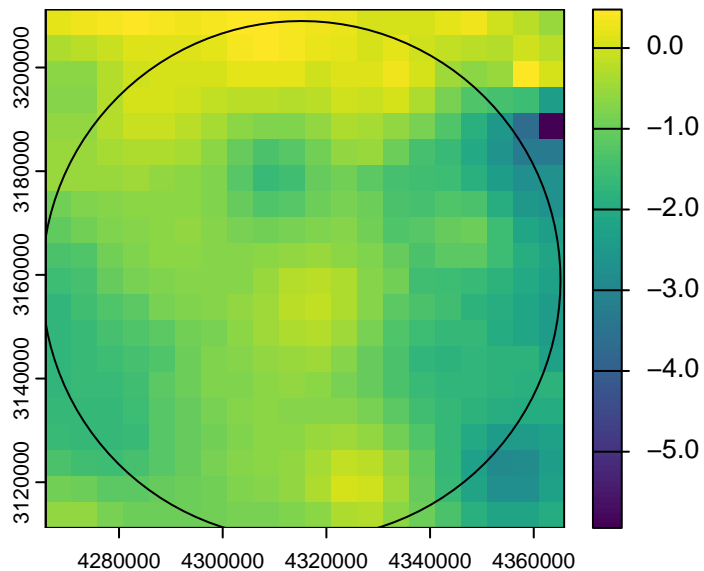
3.5.3 Cutting rasters

The function `crop()` allows to ‘cut’ a raster to an arbitrary bounding box.

```
g <- data.frame(x = 9.915803, y = 51.54128) # Goettingen
g <- st_as_sf(g, coords = c("x", "y"), crs = 4326)
g <- st_transform(g, crs = crs(r))
gb <- st_buffer(g, dist = 5e4)
```

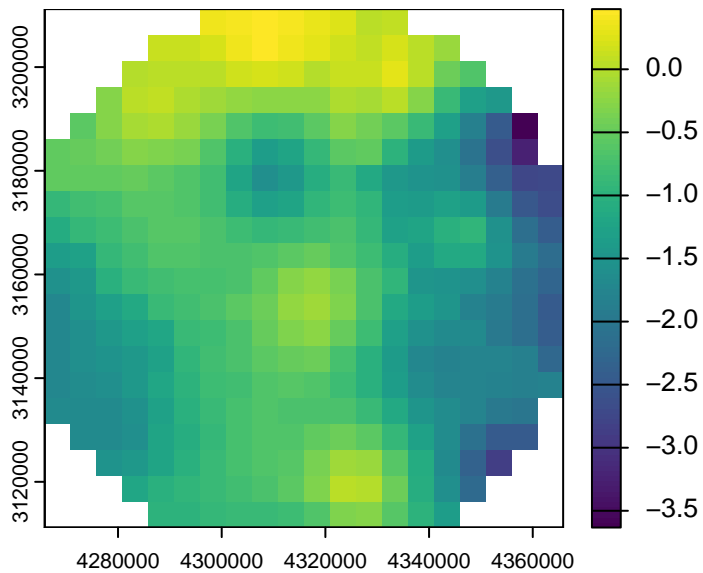
Next, we can crop the raster to the bounding box of Göttingen:

```
gr <- crop(r, gb)
plot(gr)
plot(gb, add = TRUE)
```



In order to set values that are outside the feature to NA, we will have to use the function `mask()`

```
g2 <- mask(gr, gb)
plot(g2)
```

The same also works with polygons.

```
bd1 <- st_read(here::here("data/ger/ger_states_3035.shp"))
```

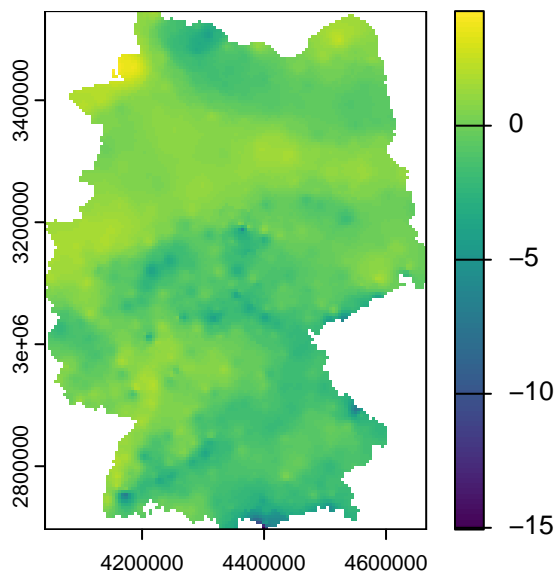
```
Reading layer `ger_states_3035' from data source
  `/Users/jsigner/ownCloud - jsigner@uni-goettingen.de@owncloud.gwdg.de/Documents/L_Lehre/M_
  using driver `ESRI Shapefile'
Simple feature collection with 16 features and 1 field
Geometry type: MULTIPOLYGON
Dimension:      XY
Bounding box:   xmin: 4031295 ymin: 2684102 xmax: 4672253 ymax: 3551343
Projected CRS:  Lambert_Azimuthal_Equal_Area
```

```
bd1 <- st_transform(bd1, crs(r))
by <- bd1[bd1$state == "Bayern", ]
temp_bayern <- mask(crop(r, by), by)
```

3.5.4 Raster algebra

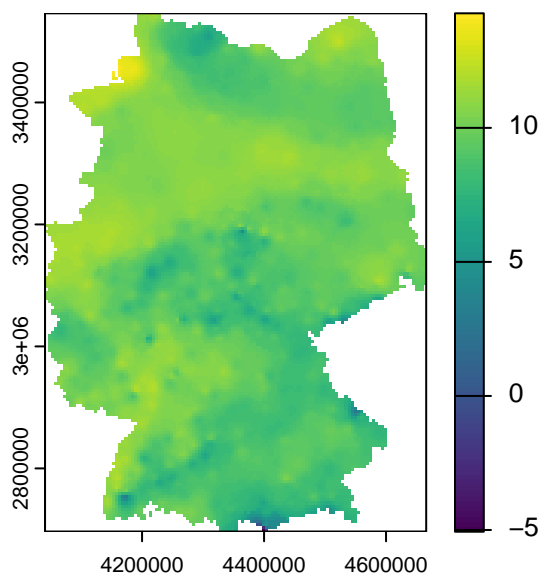
We can work with raster data like any other data. This is often referred to raster algebra.

```
plot(r)
```



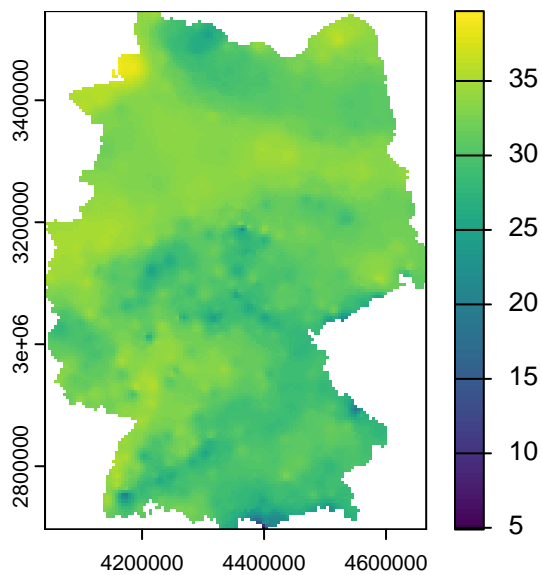
Arithmetic operators

```
plot(r + 10)
```



Temperature in degrees Fahrenheit:

```
plot(r * 1.8 + 32)
```

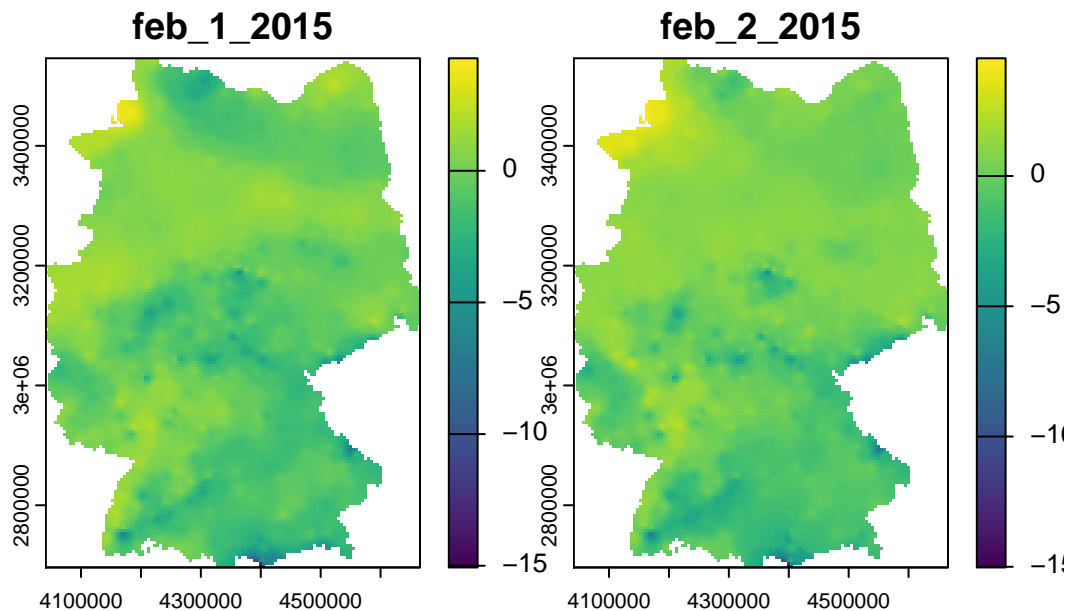


3.5.5 Working with many rasters simultaneously

The function `rast()` accepts a list of rasters. Alternatively different rasters can be combined with the `c()`-function.. It is important that all rasters have the same extent and resolution.

```
r1 <- rast(here::here("data/raster/feb_1_2015.tif"))
r2 <- rast(here::here("data/raster/feb_2_2015.tif"))
r <- rast(list(r1, r2))
r <- c(r1, r2)
```

```
plot(r)
```



The implementation of multiple rasters is very powerful and most functions that we have seen up to now also work with multiple rasters. E.g., `crs()`, `crop()`, `mask()` and raster algebra.

Note, `dim()` now gained a third dimension:

```
dim(r)
```

```
[1] 170 125 2
```

🔥 Exercise 3: Working with raster data

The aim of this exercise is to get you start working with raster data.

1. Load the Digital Elevation Model (DEM) of Germany saved in `data/raster/dem_3035.tif`.
2. What is the spatial resolution and the CRS of the raster?
3. Cut the DEM to the state of Lower Saxony (use the data set on German states for this; `data/ger/ger_states_3035.shp`).
4. What is the mean elevation of Lower Saxony?
5. Find all pixels in Lower Saxony that have an elevation of 100 m or more. What is the percentage of Lower Saxony with an elevation of 100 m or more?
6. Create a plot of Lower Saxony (using the elevation as a background with an appealing fill color; *hint: have a look at the `rcartocolor` package*), a scale and a north arrow.

3.6 Working with raster and vector data

One of the most frequent tasks is to **extract** raster data at certain locations.

Let use the raster stack from before:

```
temp1 <- rast(here::here("data/raster/feb_1_2015.tif"))
temp2 <- rast(here::here("data/raster/feb_2_2015.tif"))
rs <- c(temp1, temp2)
```

If we have a set of coordinates (e.g., Göttingen), then we can ask for the temperature in Göttingen.

```
xy <- cbind(4315159, 3158965) # x and y coord for Göttingen.
extract(temp1, xy)
```

```
feb_1_2015
1 -0.2839331
```

```
extract(temp2, xy)
```

```
feb_2_2015
1 0.9499924
```

We can apply the same on multiple rasters:

```
extract(r, xy)
```

```
feb_1_2015 feb_2_2015
1 -0.2839331 0.9499924
```

Or even many points

```
xy <- cbind(seq(4315000, by = 1e4, length = 30), 3158965)
head(xy)
```