Basic spatial R workshop - 1

Tobias Andermann

5/24/2020

Introduction to spatial data in R

(This tutorial is inspired by this very useful spatial R tutorial (http://rspatial.org/spatial/index.html) written by Robert Hijmans.)

Dependencies:

```
library(sp)
library(raster)
library(sf)
```

If you are completely new to R, have a look at this general R introduction tutorial (https://rspatial.org/intr/index.html). It is very lengthy but well written and easy to follow. You don't have to do all of it, but try to understand the basic R syntax and once you feel like you get a hang of it, you can go back to this spatial tutorial. Take your time understanding the basics of R programming, that way the spatial tutorial will make a lot more sense for you as well. It's okay if you are behind on the general course pace, the main point of this course is for you to get the most out of it, and not to copy-paste commands you don't really understand the purpose of.

Before we jump into working with real spatial data, let's first learn a bit more about the basic types of objects we will be working with. In general one can decide between two differnt types of spatial data: **vectors and rasters**.

Vectors are used to represent discrete objects with clear boundaries, e.g. sampling locations, rivers, roads, country borders etc.

Rasters are applied to represent continuous phenomena, or "spatial fields", e.g. elevation, temperature, or species diversity.

1. Vector data

Let's first go through the basic types of vector data that are used in spatial analyses, which are points, lines, and polygons.

First we create some fake data. Let's pretend we are creating data for 10 taxa with the names `A` - `J`. Let's first just create this list of fake taxon names. You can use the `LETTERS` default array and extract the 10 first elements of it as shown in the command below. We'll assign this to the variable `name` which will now contain the first 10 letters of the alphabet, which are going to be our taxon names.

```
# create taxon names
name <- LETTERS[1:10]
name
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

For each of these taxa we have a sampling location (`sampling_sites`) and a body size measurement in cm (`body_size`). To create coordinate data, we define one array with longitude and another with lattitude values of each point (made up data). You can use the `cbind()` command to pair them up into coordinate pairs, and we'll store these coordinate pairs as a new variable called `sampling_sites` . Print the content of the `sampling_sites` to the screen to understand what our fake coordinate data look like.

```
# generate sampling locations
longitude <- c(-116.7, -120.4, -116.7, -113.5, -115.5,
               -120.8, -119.5, -113.7, -113.7, -110.7)
latitude <- c(45.3, 42.6, 38.9, 42.1, 35.7, 38.9,
              36.2, 39, 41.6, 36.9)
# this command simply combines the two arrays longitude and latitude into a sh
ared matrix
sampling_sites <- cbind(longitude, latitude)

# define body sizes of sampled individuals
body_size = c(11,15,17,19,22,12,21,14,9,18)
```

A good way of dealing with all these different data arrays (names, longitude, latitude, boday size) is to join these arrays into one data frame in order to keep it together and sorted.

```
# join data in a single dataframe
wst <- data.frame(longitude, latitude, name, body_size)
wst
```

```
##    longitude latitude name body_size
## 1     -116.7     45.3    A        11
## 2     -120.4     42.6    B        15
## 3     -116.7     38.9    C        17
## 4     -113.5     42.1    D        19
## 5     -115.5     35.7    E        22
## 6     -120.8     38.9    F        12
## 7     -119.5     36.2    G        21
## 8     -113.7     39.0    H        14
## 9     -113.7     41.6    I         9
## 10    -110.7     36.9    J        18
```

We won't be working with this dataframe in this tutorial, but it's good to know how to store your data this way. Also it's handy to know how to extract individual data columns from a dataframe. We can do that by using square brackets `[]` and the index of the column (or sets of columns) that we want to extract. Within the square brackets, you can specify the lines and columns to extract, uisng the following indexing syntax `[lines,columns]` . E.g. if we want to extract the first two columns (and all lines) we can do it like this (the `,` separates the indices for line and columns, in this case we leave the lines part blank, which will lead to all lines being extracted):

```
wst[,1:2]
```

```
##    longitude latitude
## 1     -116.7     45.3
## 2     -120.4     42.6
## 3     -116.7     38.9
## 4     -113.5     42.1
## 5     -115.5     35.7
## 6     -120.8     38.9
## 7     -119.5     36.2
## 8     -113.7     39.0
## 9     -113.7     41.6
## 10    -110.7     36.9
```

Likewise if you want ot extract the first 3 lines and all columns for these lines you can type:

```
wst[1:3,]
```

```
##   longitude latitude name body_size
## 1    -116.7     45.3    A        11
## 2    -120.4     42.6    B        15
## 3    -116.7     38.9    C        17
```

Alternatively, you can also extract columns by their name by using the `$` sign. E.g. if you want to extract the body size columns you can extract if like this:

```
wst$body_size
```

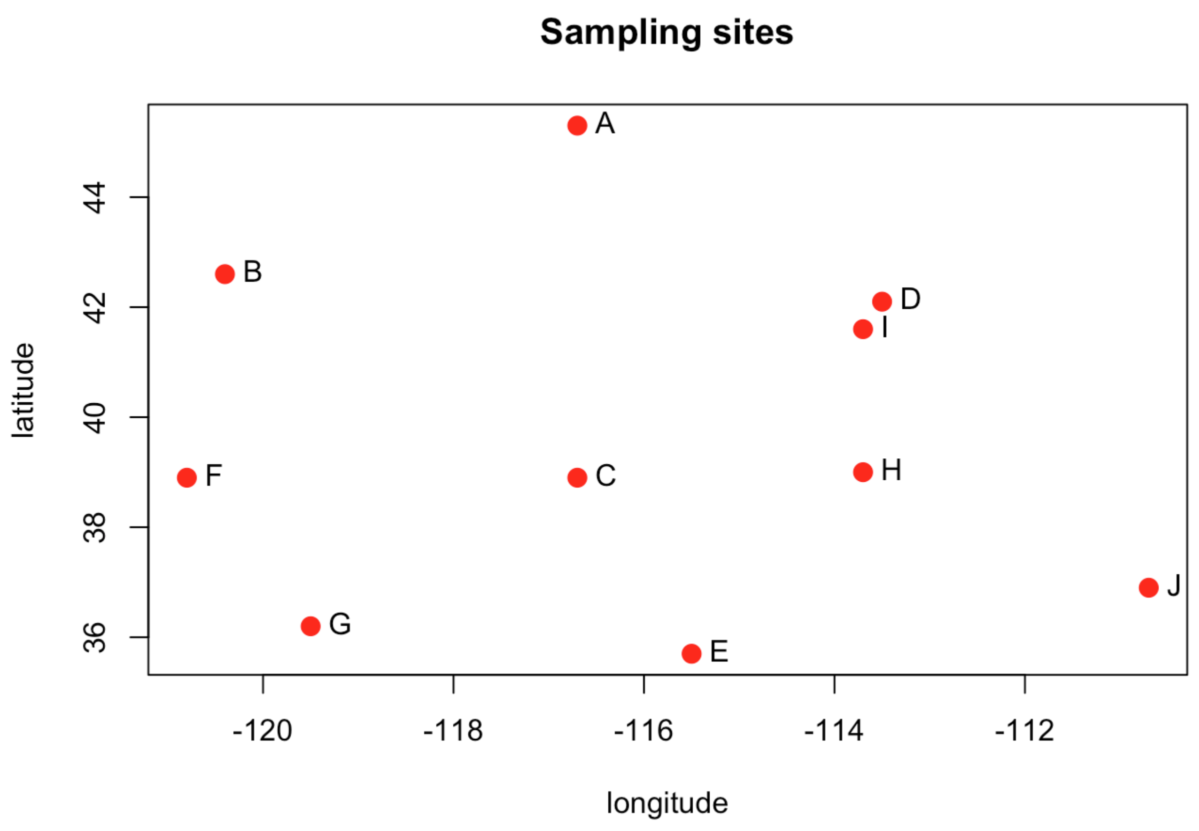
```
##  [1] 11 15 17 19 22 12 21 14  9 18
```

Points

Now let's plot the data **points** (coordinates) in a coordinate system. You plot in R by using the `plot()` function. The easiest way to plot our coordinate points is by using the `sampling_sites` object we created a little bit ago with the `cbind()` command. This object has the x and y coordinates (longitude and latitude) for each point already paired up and stored in the right format to plot. There are several settings you can use in the plot command, such as `cex` which alters the point size, `pch` which alters the shape of the points (see overview of available pch values here (http://coleoguy.blogspot.com/2016/06/symbols-and-colors-in-r-pch-argument.html)), and `col` where you can define the color of the points. Play around with these settings a bit if this is your first time plotting in R, it's good to know and fun to play with. The `main` argument in the plot command defines the title of the plot. You can add the names that we defined to each point in our `names` array by using the `text()` function as shown below.

General note on plotting: You can plot a variety of different types of objects in R using the `plot()` command. Everytime you call the `plot()` function it will overwrite the previous plot (unless you add `add=T` to the plotting command, in that case it will be added to the previous plot). In the following we will be using other plotting functions in combinations with the default `plot()`, such as e.g. the `text()`, `lines()`, `polygons()` functions. All of these functions will add information to existing plots without overwriting it, but they cannot be used alone without a preceeding `plot()` call.

```
plot(sampling_sites, cex=2, pch=20, col='red', main='Sampling sites')
# add names to plot
text(sampling_sites, name, pos=4)
```

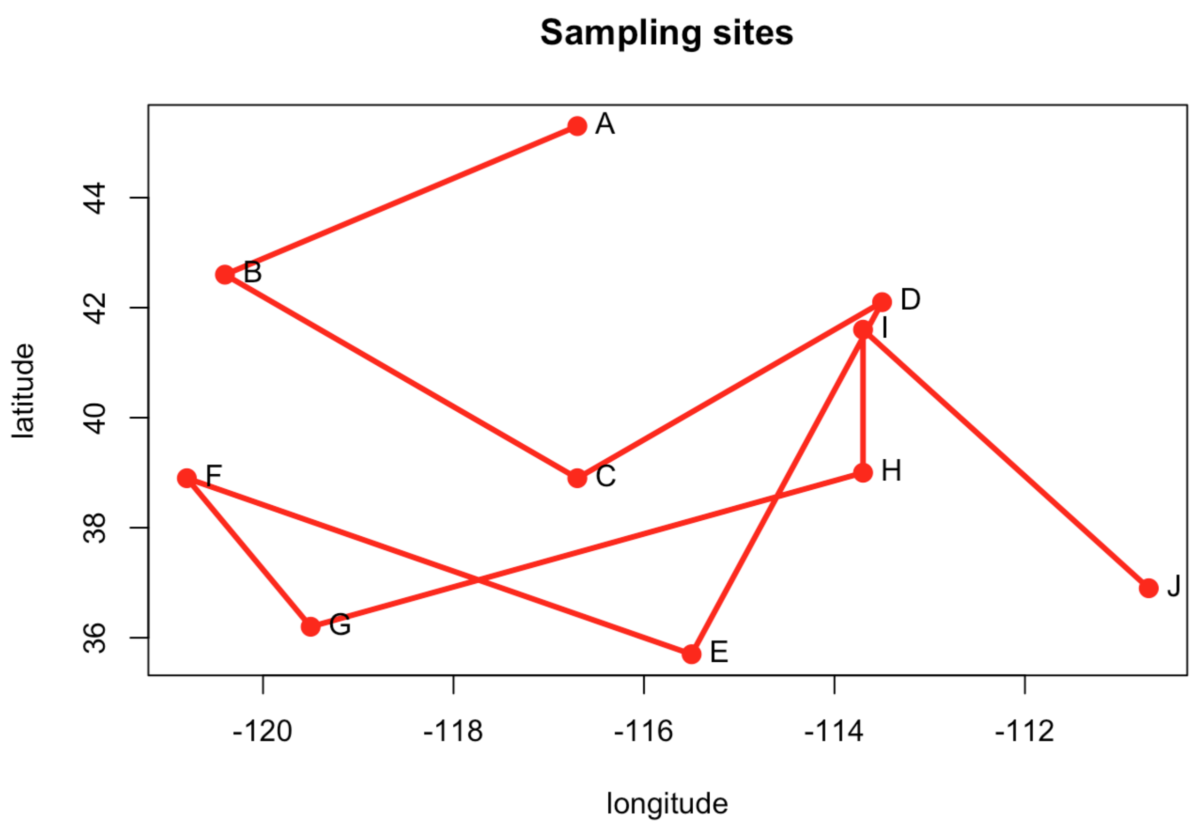


As you see in our plot we didn't plot a map, but instead a simple coordinate system. However the spatial relationships between the points are plotted in the correct ratios based on their latitude and longitude coordinates. In principle, plotting points on a map is just plotting in a regular coordinate system (as above) with a fancy background.

Lines

We can also plot **lines** connecting our dots from above, using the `lines()` function. Here we are only **plotting** our points as lines but later we will see how to define line objects, which besides the point coordinate information also carry the information in which order the points are connected. This can be used e.g. to visualize a time chronology of sampling.

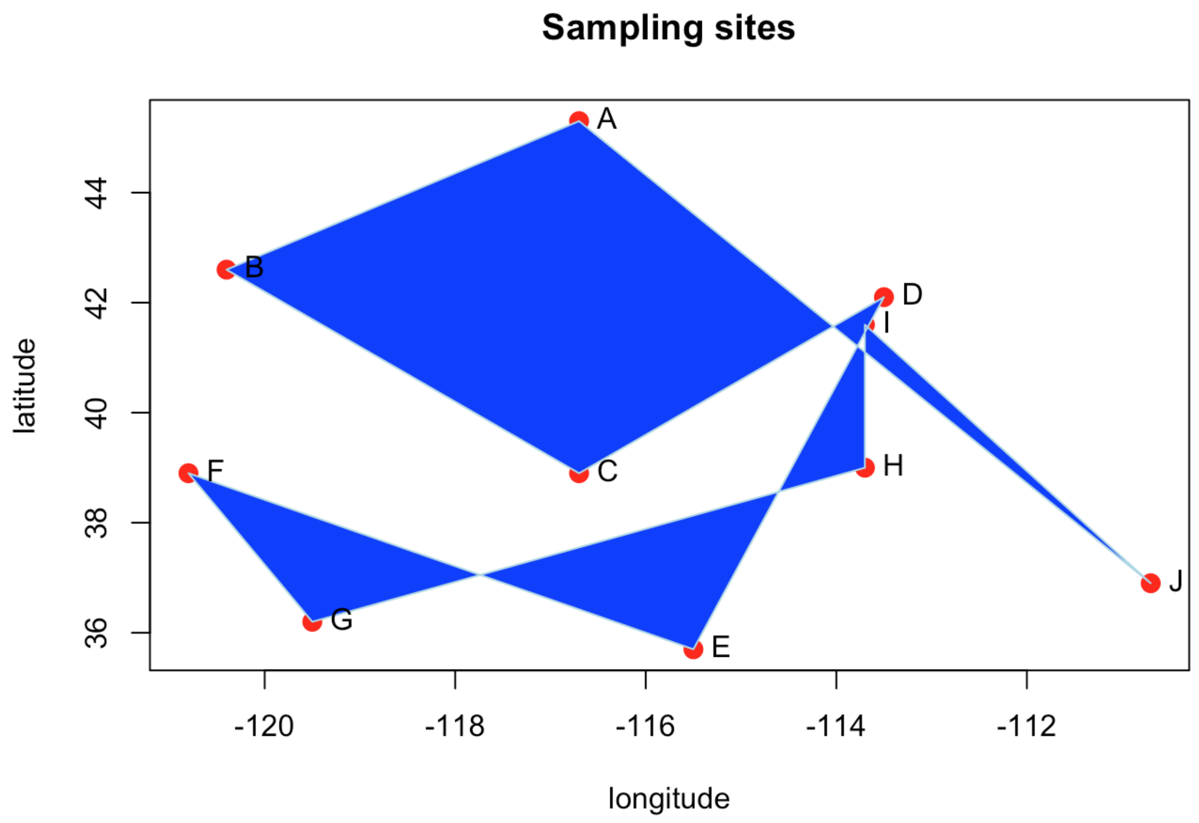
```
# first plot the points just as we did previously
plot(sampling_sites, cex=2, pch=20, col='red', main='Sampling sites')
# draw lines between data points
lines(sampling_sites, lwd=3, col='red')
text(sampling_sites, name, pos=4)
```



Polygons

We can also plot a **polygon** from the same data, using the `polygon()` function.

```
plot(sampling_sites, cex=2, pch=20, col='red', main='Sampling sites')
polygon(sampling_sites, col='blue', border='light blue')
text(sampling_sites, name, pos=4)
```



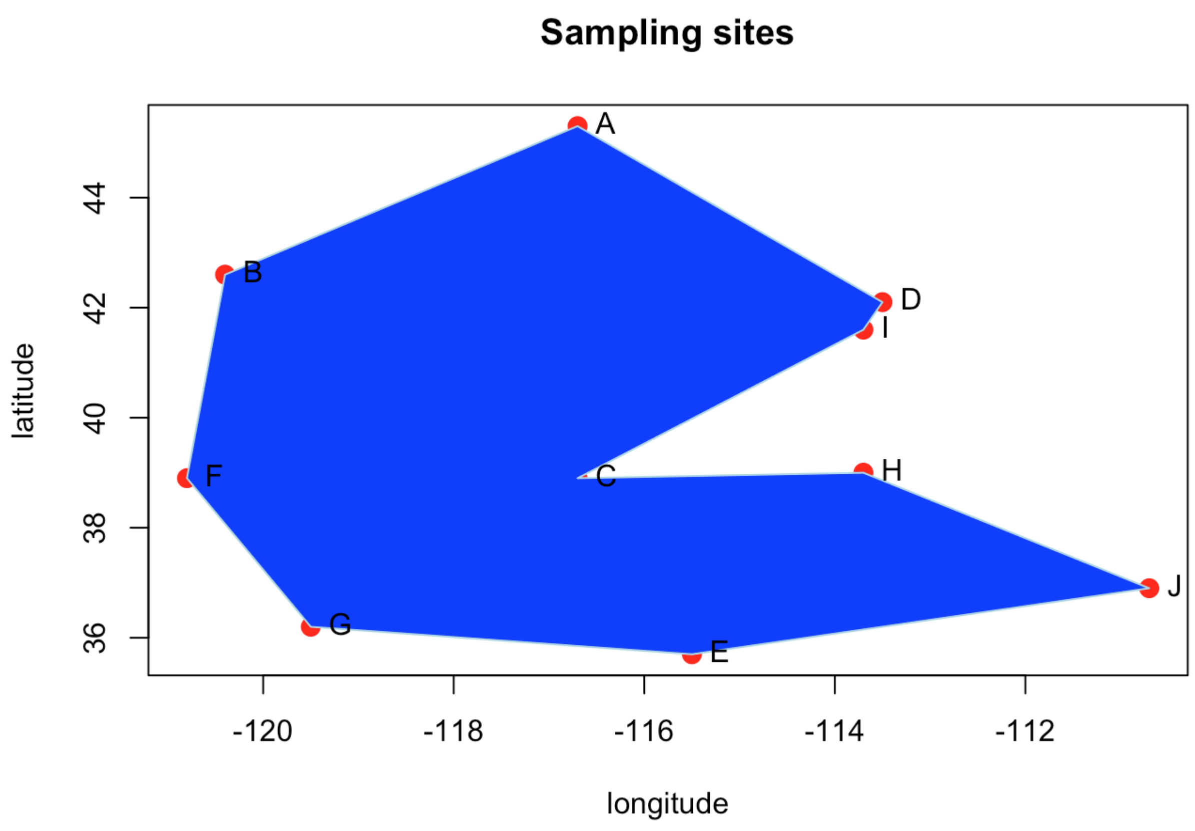
As you can see, also in case of polygons it matters in which order the points are stored. Let's reorder our data points and see how the polygon changes. For reordering an array we can just select the elements in the desired order by using their indices. For example for an array containing the 3 values 'a','b',and 'c' named `my_tiny_array` you can reorder the array having the 2nd element followed by the 3rd and then the 1st like this:

```
my_tiny_array = c('a','b','c')
my_tiny_array[c(2,3,1)]
```

```
## [1] "b" "c" "a"
```

Now we want to do the same for our coordinate array `sampling_sites`. Since this is a 2D array we need to specify rows and columns when selecting indices. Since we want to extract both columns and only reorder the lines in our `sampling_sites` array, we specify the new order of row indices followed by a comma.

```
reordered_sampling_sites = sampling_sites[ c(1,2,6,7,5,10,8,3,9,4), ]
plot(reordered_sampling_sites, cex=2, pch=20, col='red', main='Sampling sites
')
polygon(reordered_sampling_sites, col='blue', border='light blue')
text(sampling_sites, name, pos=4)
```



Defining spatial objects

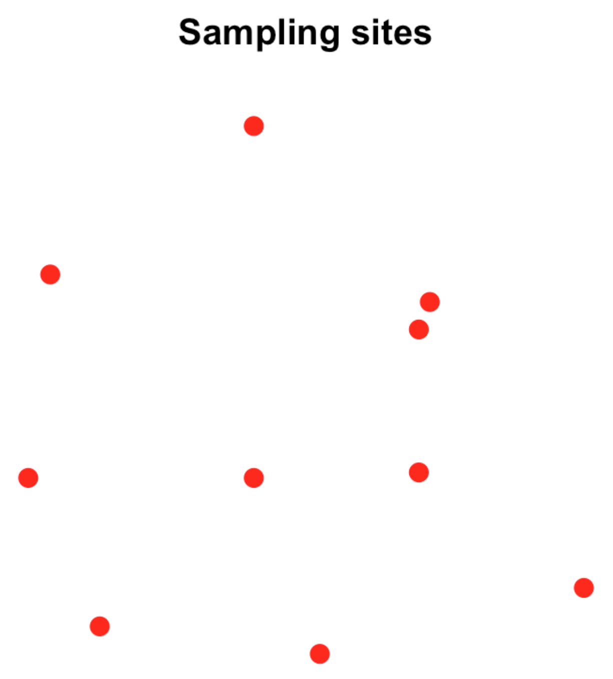
So far we have only worked with simple data points stored in arrays. However, different data object types exist that are specifically designed to handle spatial data. Some of the most commonly used objects are defined in the `sp` package. For vector data, the basic types are the `SpatialPoints()`, `SpatialLines()`, and `SpatialPolygons()`. These data objects only represent geometries. To also store attributes, additional data objects are available with these names plus 'DataFrame', for example, `SpatialPolygonsDataFrame()` and `SpatialPointsDataFrame()`. The advantage of this over e.g. just storing your coordinates in array form as we did so far, is that these data spatial objects have several useful functions, for example you can define the coordinate reference system of your coordinates in the spatial object, which makes it easy to later on transform your coordinates into another coordinate projection (you will see examples of that later on). Also most spatial functions you may be using to process or plot your data may require your data to be stored as spatial objects. Let's store our coordinates as a `SpatialPoints()` object and inspect the content and structure of this object with the `showDefault()` function.

```
library(sp)
pts <- SpatialPoints(sampling_sites)
# use the showDefault() function to view th content of the object (works for a
ny type of object in R)
showDefault(pts)
```

```
## An object of class "SpatialPoints"
## Slot "coords":
##       longitude latitude
##  [1,]    -116.7     45.3
##  [2,]    -120.4     42.6
##  [3,]    -116.7     38.9
##  [4,]    -113.5     42.1
##  [5,]    -115.5     35.7
##  [6,]    -120.8     38.9
##  [7,]    -119.5     36.2
##  [8,]    -113.7     39.0
##  [9,]    -113.7     41.6
## [10,]    -110.7     36.9
##
## Slot "bbox":
##              min    max
## longitude -120.8 -110.7
## latitude    35.7   45.3
##
## Slot "proj4string":
## CRS arguments: NA
```

You see that besides the coordinate pairs we defined, this object contains information about the bounding box `bbox` , which defines the smallest rectangle containing all of your points. It also contains the currently empty slot `proj4string` , which is where you can store the coordinate reference system of your points, which we will do in a little bit. You will also notice when plotting the points that the plot will look a bit different than before (e.g. it's lacking the x and y axis). The reason for that is that the default plotting options for `SpatialPoints()` objects are different than those of simple numeric arrays (which makes sense because often when you want to plot points on a map, you may not wany the x-axis and y-axis to show).

```
plot(pts, cex=2, pch=20, col='red', main='Sampling sites')
```



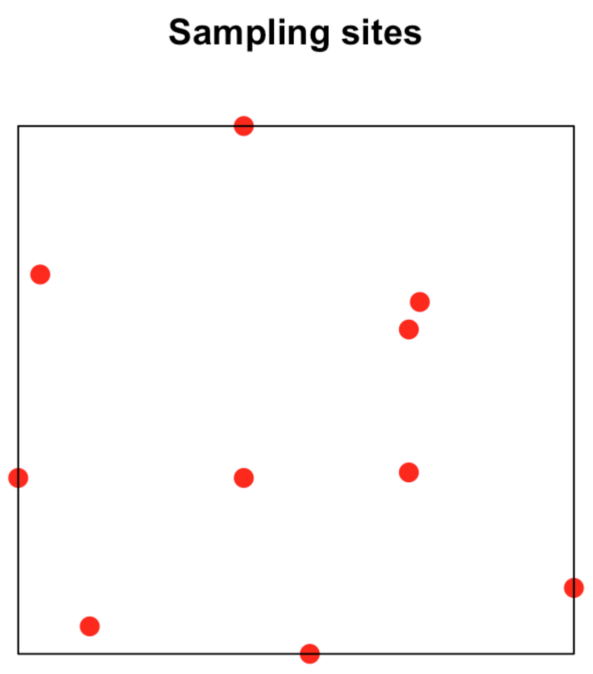
You can extract the additional values stored in the `SpatialPoints()` object, e.g. the coordinates of the bounding box, like this:

```
box = bbox(pts)
box
```

```
##                 min    max
## longitude -120.8 -110.7
## latitude     35.7   45.3
```

Let's plot the bounding box with our coordinates to see what it looks like. You can use the `rect()` function to plot a simple rectangle given the x and y coordinates of two opposing corner points:

```
plot(pts, cex=2, pch=20, col='red', main='Sampling sites')
rect(box[1],box[2],box[3],box[4])
```



Let’s now use the `SpatialPointsDataFrame()` function to define a spatial object containing additional information about the samples (sample name and measured body size). For this we first store the data in a dataframe and then assign it to the `SpatialPointsDataFrame()` object together with the coordinates:

```
additional_data <- data.frame(sample_name=name, body_size=body_size)
ptsdf <- SpatialPointsDataFrame(pts, data=additional_data)
showDefault(ptsdf)
```

```
## An object of class "SpatialPointsDataFrame"
## Slot "data":
##    sample_name body_size
## 1            A        11
## 2            B        15
## 3            C        17
## 4            D        19
## 5            E        22
## 6            F        12
## 7            G        21
## 8            H        14
## 9            I         9
## 10           J        18
##
## Slot "coords.nrs":
## numeric(0)
##
## Slot "coords":
##       longitude latitude
##  [1,]    -116.7     45.3
##  [2,]    -120.4     42.6
##  [3,]    -116.7     38.9
##  [4,]    -113.5     42.1
##  [5,]    -115.5     35.7
##  [6,]    -120.8     38.9
##  [7,]    -119.5     36.2
##  [8,]    -113.7     39.0
##  [9,]    -113.7     41.6
## [10,]    -110.7     36.9
##
## Slot "bbox":
##              min    max
## longitude -120.8 -110.7
## latitude    35.7   45.3
##
## Slot "proj4string":
## CRS arguments: NA
```

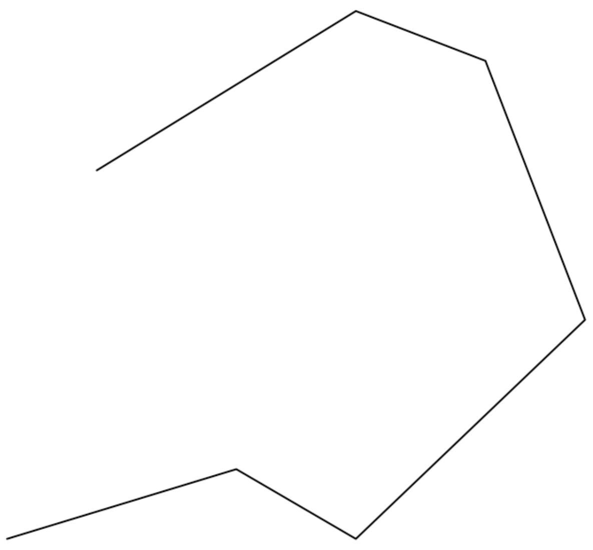
Now we'll use a different function to create a spatial line object using the `spLines()` function. For this we first need to load the package `raster`. We'll also make up some new data, in order to have different data points than for the `SpatialPoints` object from above:

```
library(raster)
# make up new data
lon <- c(-116.8, -114.2, -112.9, -111.9, -114.2, -115.4, -117.7)
lat <- c(41.3, 42.9, 42.4, 39.8, 37.6, 38.3, 37.6)
lonlat <- cbind(lon, lat)
# store data as spatial lines object
lns <- spLines(lonlat)
lns
```

```
## class       : SpatialLines
## features    : 1
## extent      : -117.7, -111.9, 37.6, 42.9  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
```

Now if you plot this `spLines()` object, the plot function will automatically know to plot it as lines and not as points:

```
plot(lns)
```



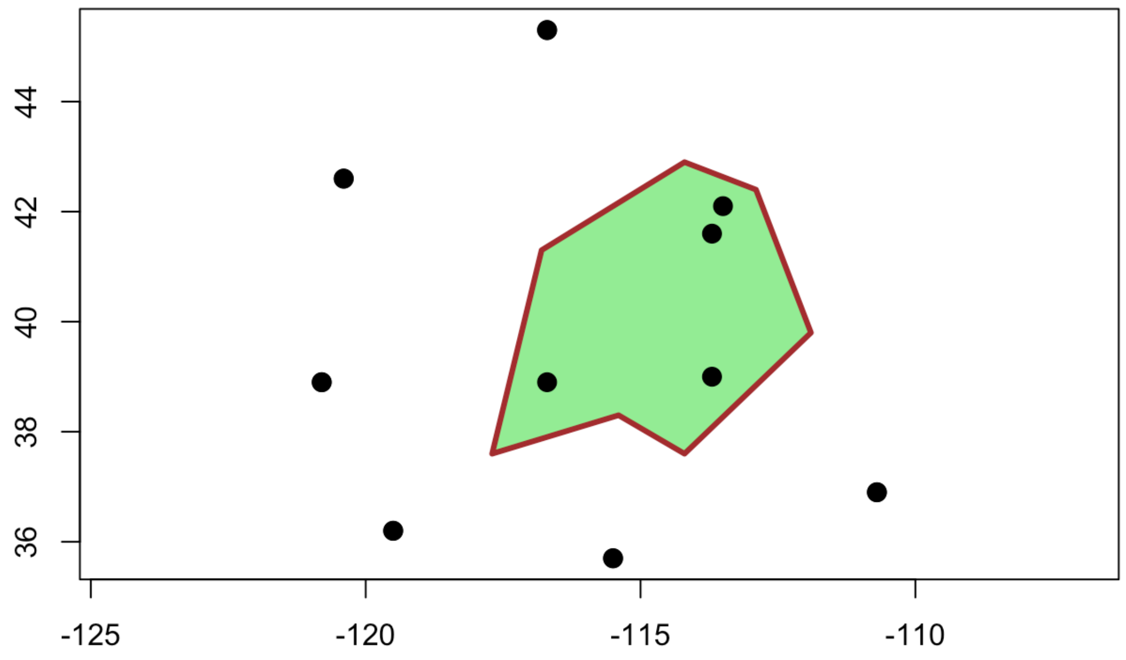
Similarly to the lines object, we can use the same points to create a polygon using the `spPolygons()` function:

```
pols <- spPolygons(lonlat)
pols
```

```
## class       : SpatialPolygons
## features    : 1
## extent      : -117.7, -111.9, 37.6, 42.9  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
```

Now let’s plot the polygon and on top of it the spatial points form earlier.

```
#plot(pols, axes=TRUE, las=1)
plot(pts, col='black', pch=20, cex=2,axes=TRUE)
plot(pols, border='brown', col='lightgreen', lwd=3,add=T)
plot(pts, col='black', pch=20, cex=2,add=T)
```



You may have noticed that this time we don't have to use the specific plotting functions `polygon()` or `lines()` to tell R what kind of object we want to plot, but instead we are simply using the `plot()` function and R plots the data in the correct form by default, since this information is contained in the specific types of spatial objects we defined.

2. Rasters

Now let'swork a bit with rasters. The first type of raster object we will work with is called `RasterLayer()` and is defined in the `raster` library.

A RasterLayer object represents single-layer raster data. A RasterLayer object always stores a number of fundamental parameters that describe it. These include the number of columns and rows, the spatial extent, and the Coordinate Reference System.

Here we create a RasterLayer from scratch. But note that in most cases where real data is analyzed, these objects are read from a file. We define a raster with 10 rows and 10 columns and we define the extent of the raster on the x-axis (longitude) and on the y-axis (latitude).

```
library(raster)
rast <- raster(ncol=10, nrow=10, xmn=-150, xmx=-80, ymn=20, ymx=60)
rast
```

```
## class       : RasterLayer
## dimensions  : 10, 10, 100  (nrow, ncol, ncell)
## resolution  : 7, 4  (x, y)
## extent      : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

You can see in the last line of the output that by default the `raster()` function created the raster using the coordinate reference system `+proj=longlat +datum=WGS84` . We will learn later what that means.

So far the object we created represents only a skeleton of a raster data set. That means, we have defined information about its location, resolution, etc., but there are no values associated with it. Let's fill the skeleton with some made up data. In this case we assign a vector of random numbers (generated from a uniform distribution using the `runif()` function) with a length that is equal to the number of cells of the RasterLayer.

Exploring the raster object: Just as we did earlier, we can use the `showDefault()` function again to explore the contents of the raster object. Executing `showDefault(rast)` will show you all the available information stored in the raster object that we just defined. Any of the 'Slots' that are listed as output of that function are values that can be accessed by calling the name of the slot followed by the name of the raster object in parantheses. For example you will find the slot `nrows` in the output of the `showDefault(rast)` command and you can extract that information specifically with `ncol(rast)` .

You can determine the number of cells in your raster using `ncell()` :

```
ncell(rast)
```

```
## [1] 100
```

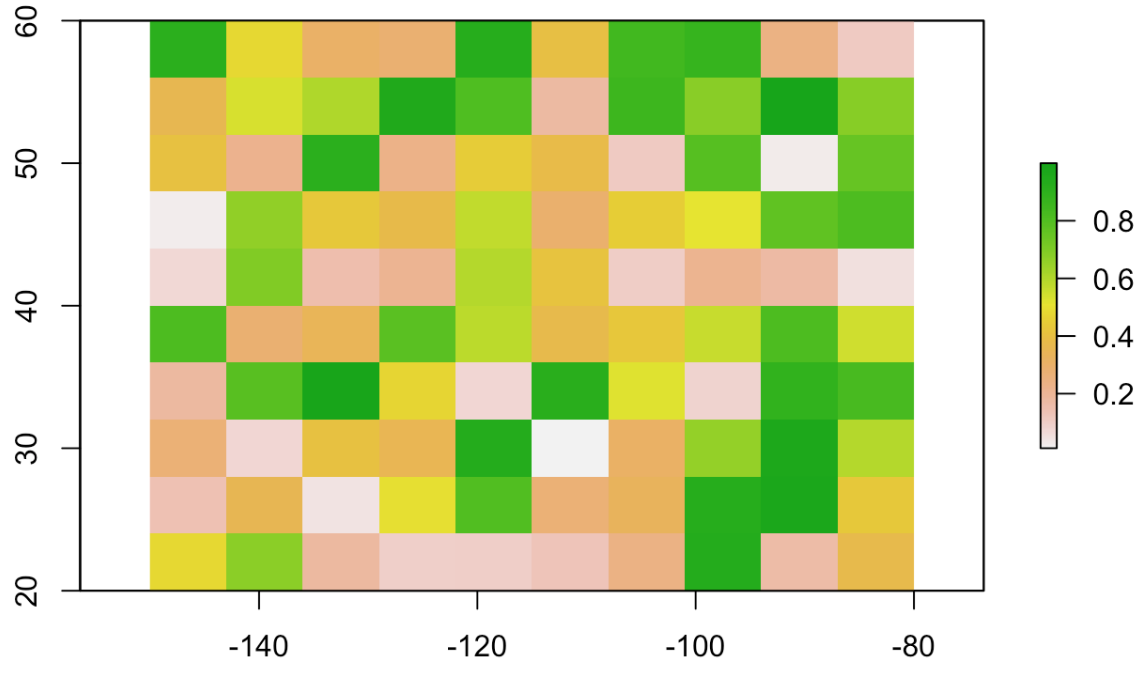
Now let's create as many random numbers (between 0 and 1) as we have cells in our raster:

```
raster_values = runif(ncell(rast))
raster_values
```

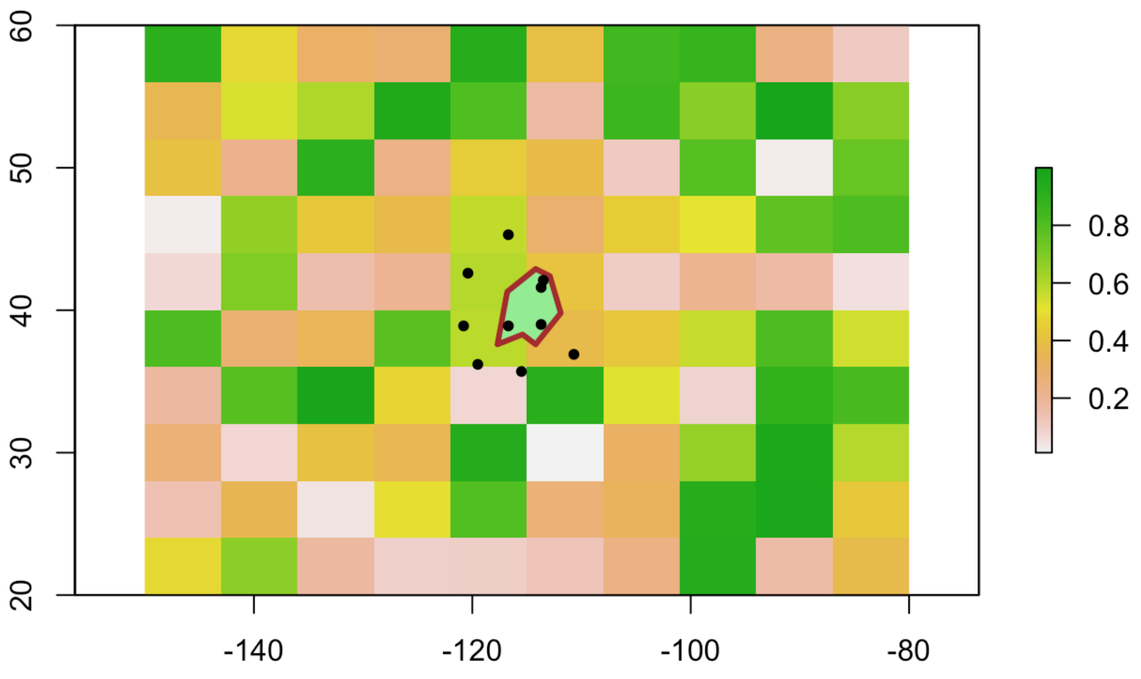
```
##   [1] 0.91108990 0.47888477 0.30078511 0.27978253 0.92996799 0.39740819
##   [7] 0.84376170 0.87608198 0.24175112 0.10705788 0.35789493 0.53102462
##  [13] 0.60811514 0.95866904 0.81020187 0.17453793 0.85758492 0.68181800
##  [19] 0.99979247 0.68585441 0.40379075 0.21724660 0.91055419 0.22843950
##  [25] 0.44483213 0.37876881 0.10645179 0.79382194 0.02610243 0.75444767
##  [31] 0.02076422 0.66349103 0.43087246 0.37642975 0.57318520 0.29279635
##  [37] 0.45144584 0.51080866 0.76831238 0.81178796 0.06723948 0.69647719
##  [43] 0.15204752 0.20870190 0.59601015 0.40984665 0.09868473 0.21330242
##  [49] 0.17148709 0.04791233 0.81679860 0.27790935 0.34239455 0.78330041
##  [55] 0.58256799 0.37189438 0.42602417 0.56338633 0.81492192 0.54748920
##  [61] 0.17972726 0.79086217 0.99118690 0.47583985 0.07103416 0.91935700
##  [67] 0.51903544 0.08232616 0.89058176 0.82657045 0.26656488 0.07300634
##  [73] 0.40448049 0.34869632 0.93241727 0.01065626 0.30535706 0.65566730
##  [79] 0.96384647 0.59405772 0.14076668 0.35427225 0.04207472 0.49692030
##  [85] 0.80508603 0.26708649 0.32837463 0.92772453 0.97525566 0.43178095
##  [91] 0.47854656 0.67609166 0.17835908 0.09348358 0.09772431 0.12890887
##  [97] 0.23640800 0.93260003 0.16068380 0.37415333
```

Now we assign these random values to the raster using the `values()` function and plot the raster with these new values.

```
values(rast) = raster_values
plot(rast)
```



Task: Just because we can, and to demonstrate that the coordinate systems of our created raster and our previously defined spatial vector object match, let's plot the spatial polygon `pols` and the spatial points `pts` on top of the raster. Tip: you can use `add=T` in the plot command to add additional layers on top of an existing plot without replacing the previous plot. That way you can add multiple elements into the same plot. The final plot should look like the one below.



3. Reading spatial data from file

We can use the `readShapePoly()` function in order to read a shape file as a vectorized polygon into R. A shape file is usually made up of at least four parts. The .SHP, the .DBF, the .PRJ and the .SHX.

- **SHP:** This file contains the geometry of each feature.
- **DBF:** This is a database file which contains the attribute data for all of the features in the dataset. The database file is very similar to a sheet in a spreadsheet and can even be opened in Excel.
- **SHX:** The .shx is the spatial index, it allows spatial programs to find features within the .SHP file more quickly.
- **PRJ:** The .prj is the projection file. It contains information about the “projection” and “coordinate system” the data uses.

Below we are going to read a shape file of the country shape of Germany, downloaded from DIVA-GIS (http://www.diva-gis.org/gdata). You can open the link and download your own shape file or work with the example data, which you can find in the data folder of the GitHub repo of this course (https://github.com/tobiashofmann88/spatial_R_course). I recommend you download the whole complete GitHub repo by clicking on the green ‘Clone or download’ button and then ‘Download as ZIP’.

Once you have downloaded a shape file from DIVA-GIS or from the course GitHub repo, you can read the file into R using the `st_read()` function. To use this function we need to load the `sf` package with `library(sf)`. To load the file, make sure that you provide the correct file path from your current working directory.

Finding files in R: Providing the right filepath to a file in R can be a bit difficult at the beginning when you're not used to finding files by navigating through the file path system. A good starting point is usually to check the current working directory. You can see your working directory with `getwd()`. From here you need to provide the path to where your file is stored. One nice feature about RStudio is that is autocompletes paths once you start writing them. For using this function just type './' and then press the Tab button, which will show you all available files and folders in your current directory. Choose the folder you want to navigate to and press Tab again to show the content of that folder. You can use `../` to navigate out of a folder inot the parent directory. **Important note for Windows users:** Instead of using `/` in the file path you will have to use `\\`!

```
library(sf)
germany = st_read('../data/DEU_adm/DEU_adm0.shp')
```

```
## Reading layer `DEU_adm0' from data source `/Users/tobias/GitHub/spatial_R_c
ourse/data/DEU_adm/DEU_adm0.shp' using driver `ESRI Shapefile'
## Simple feature collection with 1 feature and 70 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 5.871619 ymin: 47.26986 xmax: 15.03811 ymax: 55.05653
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

The object that is created by the `st_read()` function is a feature collection in `sf` format. You don;t need to bother fully understanding this format for our purposes, but instead we will just turn it into the now familiar `SpatialPolygons` format (`SpatialPolygonsDataFrame` to be precise, which we'll look at a bit closer below). You can do this by using the command `as(my_object, 'Spatial')`, where `my_object` in this case is our `sf` object called `germany`:

```
germany_spatial = as(germany, 'Spatial')
germany_spatial
```

```
## class       : SpatialPolygonsDataFrame
## features    : 1
## extent      : 5.871619, 15.03811, 47.26986, 55.05653  (xmin, xmax, ymin, ym
ax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,
0,0
## variables   : 70
## names       : ID_0, ISO,  NAME_0, OBJECTID_1, ISO3, NAME_ENGLI, NAME_ISO, N
AME_FAO,  NAME_LOCAL, NAME_OBSOL, NAME_VARIA, NAME_NONLA, NAME_FRENC, NAME_SPA
NI, NAME_RUSSI, ...
## value       :   86, DEU, Germany,         62,  DEU,    Germany,  GERMANY,
Germany, Deutschland,         NA,    Germany,         NA,  Allemagne,   Aleman
ia,   Германия, ...
```

```
(germany_spatial)
```

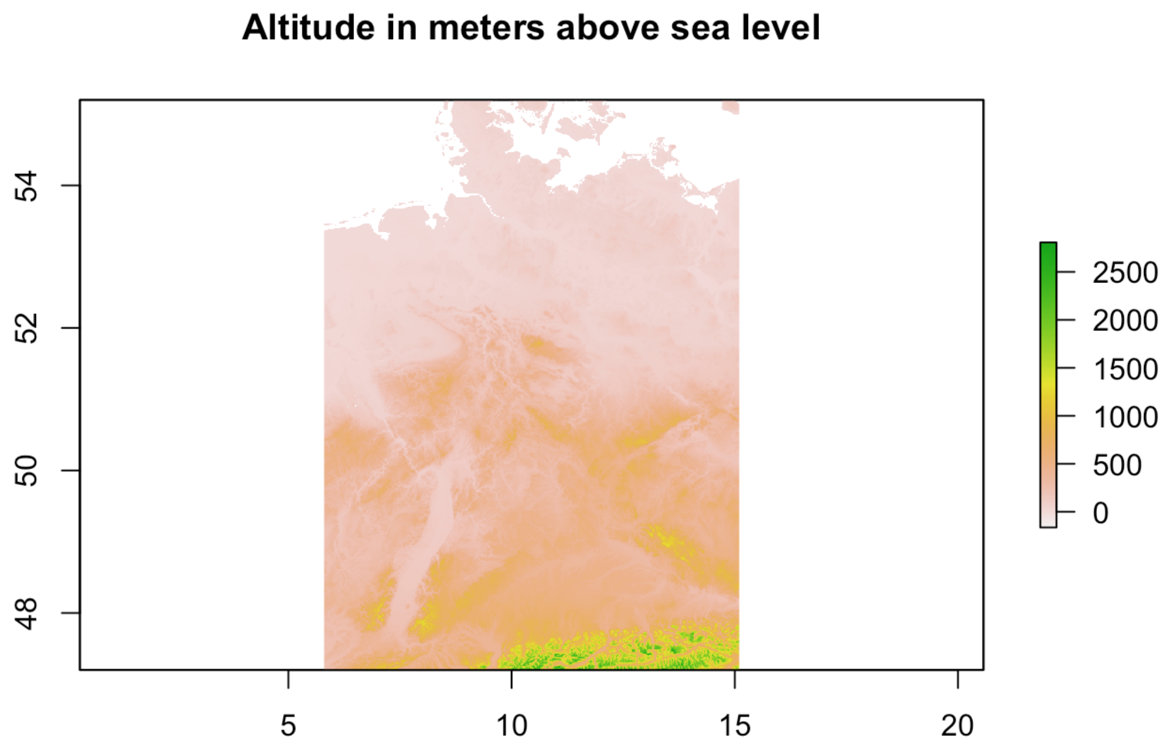
```
## class       : SpatialPolygonsDataFrame
## features    : 1
## extent      : 5.871619, 15.03811, 47.26986, 55.05653  (xmin, xmax, ymin, ym
ax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,
0,0
## variables   : 70
## names       : ID_0, ISO,  NAME_0, OBJECTID_1, ISO3, NAME_ENGLI, NAME_ISO, N
AME_FAO,  NAME_LOCAL, NAME_OBSOL, NAME_VARIA, NAME_NONLA, NAME_FRENC, NAME_SPA
NI, NAME_RUSSI, ...
## value       :   86, DEU, Germany,         62,  DEU,    Germany,  GERMANY,
Germany, Deutschland,         NA,    Germany,         NA,  Allemagne,   Aleman
ia,   Германия, ...
```

Now let’s plot the polygon object we just created:

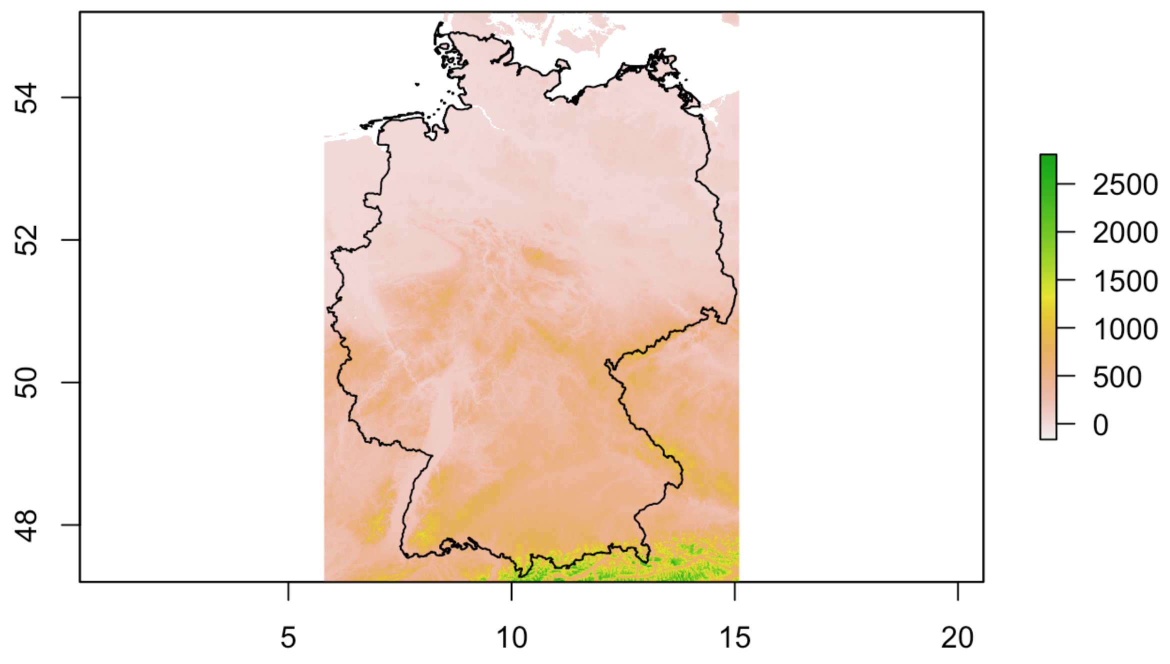
```
plot(germany_spatial,main='Germany')
```

Similarly we can read raster data, using the `raster()` function. In this example we are reading altitude information covering the area of Germany, which we will use in combination with the shape file from above:

```
germany_alt_raster = raster('../data/DEU_alt/deu_alt.grd')
plot(germany_alt_raster,main='Altitude in meters above sea level')
```



Task: The raster is a rectangular grid filled with values (just as we did manually for the random value raster earlier in the tutorial). Let's now plot the country borders of Germany (stored in the `germany_spatial` object) on top of the raster, just as you did before when plotting our hand-made polygon on top of the random value raster. Your plot should look like the one below.



4. Coordinate Reference Systems

The reason why plotting the country borders of Germany on top of the raster worked so smoothly is because both objects (the polygon and the raster) are scaled in the same coordinate reference system (CRS). We will work through an where the CRSs between two objects don’t match, and where they have to be adjusted first, in the next tutorial. For now, to get a basic understanding what this is all about, please read the sections 6.1 and 6.2 in this summary about coordinate reference systems (http://rspatial.org/spatial/rst/6-crs.html), which explains the background about different two-dimensional projections of geographic data.

5. Working with vector data

Now we are going to play around with a map of Luxembourg in order to demonstrate some of the basic polygon transformation and editing tools.

Let’s first read the shape file and turn it into a spatial object, just as we did before for the Germany shape file:

```
lux = st_read('../data/luxembourg/lux.shp')
```

```
## Reading layer `lux' from data source `/Users/tobias/GitHub/spatial_R_course
/data/luxembourg/lux.shp' using driver `ESRI Shapefile'
## Simple feature collection with 12 features and 5 fields
## geometry type:  POLYGON
## dimension:      XY
## bbox:           xmin: 5.74414 ymin: 49.44781 xmax: 6.528252 ymax: 50.18162
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

```
lux_spatial = as(lux, 'Spatial')
lux_spatial
```

```
## class       : SpatialPolygonsDataFrame
## features    : 12
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, yma
x)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,
0,0
## variables   : 5
## names       : ID_1,     NAME_1, ID_2,   NAME_2, AREA
## min values  :    1,   Diekirch,    1, Capellen,   76
## max values  :    3, Luxembourg,   12,    Wiltz,  312
```

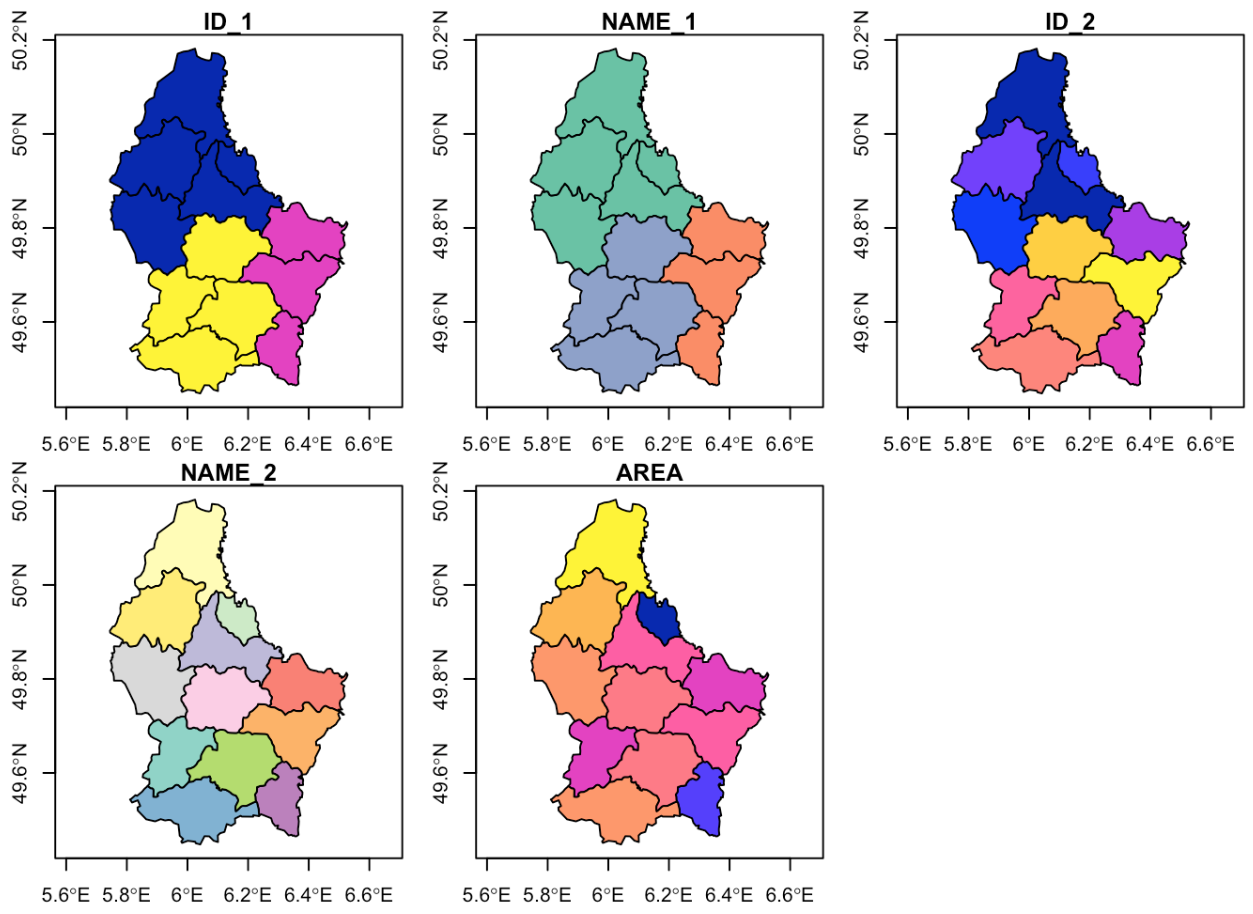
The shape object is formatted as a `SpatialPolygonsDataFrame` . Similarly to the `SpatialPointsDataFrame` from earlier in the tutorial, a `SpatialPolygonsDataFrame` contains a list of `SpatialPolygons` , as well as other data associated with these polygons. This could for example be the polygons of different regions within the country and the associated values could be things like name, population, size, etc. of these individual regions. You can see from the output of the above command (by just executing the name of the object: `lux_spatial`), that there are 12 features in this object. That means that this `SpatialPolygonsDataFrame` contains 12 different polygons, each based on it's own set of coordinates and associated with it's own data. You can see the names of the different data columns that are stored in the object using the `names()` function:

```
names(lux_spatial)
```

```
## [1] "ID_1"   "NAME_1" "ID_2"   "NAME_2" "AREA"
```

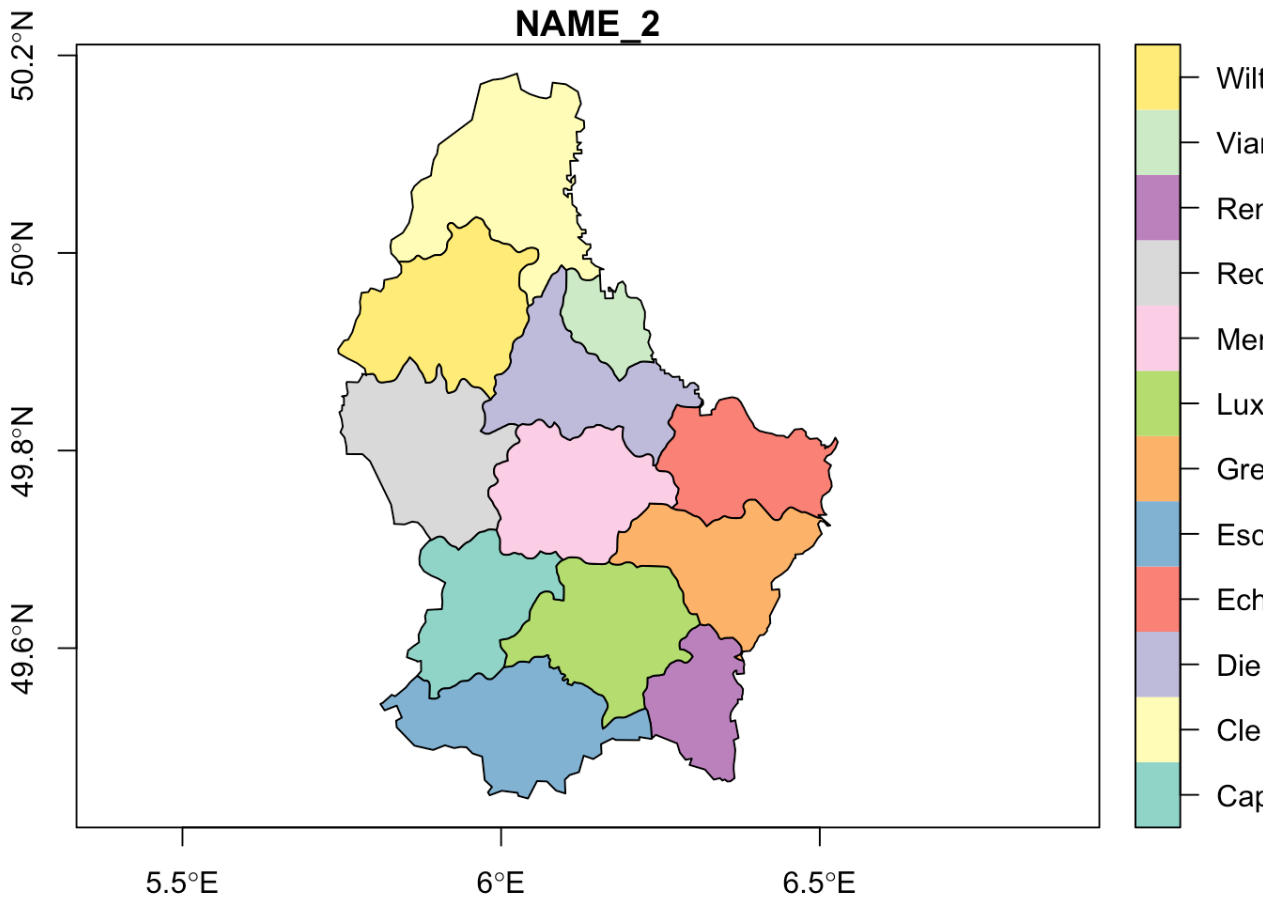
There are five different values associated with each polygon which are stored in the columns `ID_1` , `NAME_1` , `ID_2` , `NAME_2` , `AREA` . When plotting the actual `sf` object `lux` (instead of the `SpatialPolygonDataFrame` object `lux_spatial`), the default plotting function automatically puts out all layers and colors the polygons by their values:

```
plot(lux,axes = T)
```



You can see that e.g. `NAME_1` defines some larger regions, as several neighbouring polygons share the same value for that data column, while `NAME_2` seems to define the names of the polygons we have in our dataframe (more regional delimitations). We can select a single layer by using the respective index in square brackets `[]`:

```
plot(lux[4],axes = T)
```



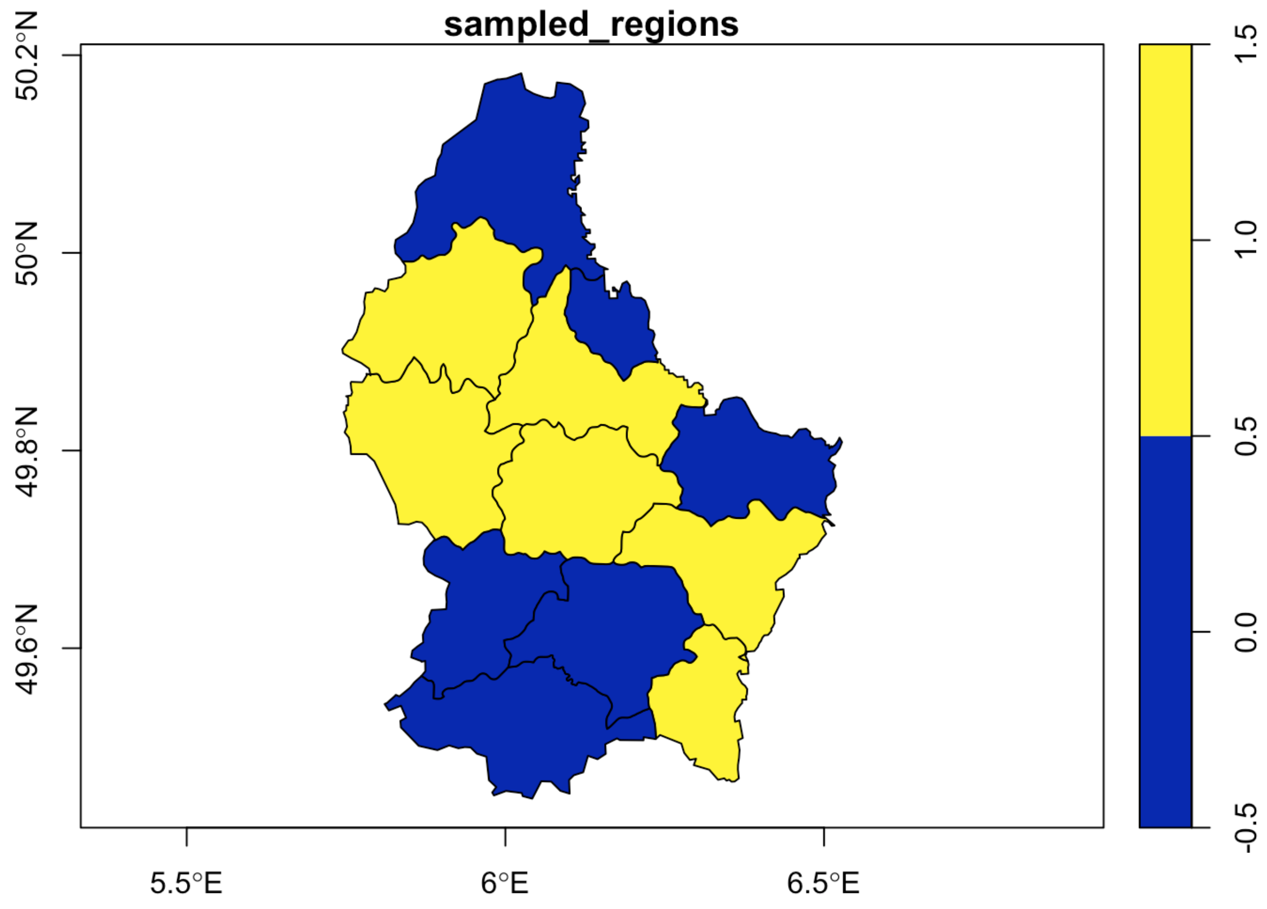
We can extract the values of a specific column like this:

```
lux$NAME_2
```

```
##  [1] Clervaux         Diekirch         Redange          Vianden
##  [5] Wiltz            Echternach       Remich           Grevenmacher
##  [9] Capellen         Esch-sur-Alzette Luxembourg       Mersch
## 12 Levels: Capellen Clervaux Diekirch Echternach ... Wiltz
```

In some cases you may want to add a column of additional values to the polygon object, e.g. if you want to code which of these regions you have sampled (`1` stands for sampled and `0` stands for not sampled):

```
new_values = c(0,1,1,0,1,0,1,1,0,0,0,1)
lux$sampled_regions = new_values
plot(lux['sampled_regions'],axes = T)
```

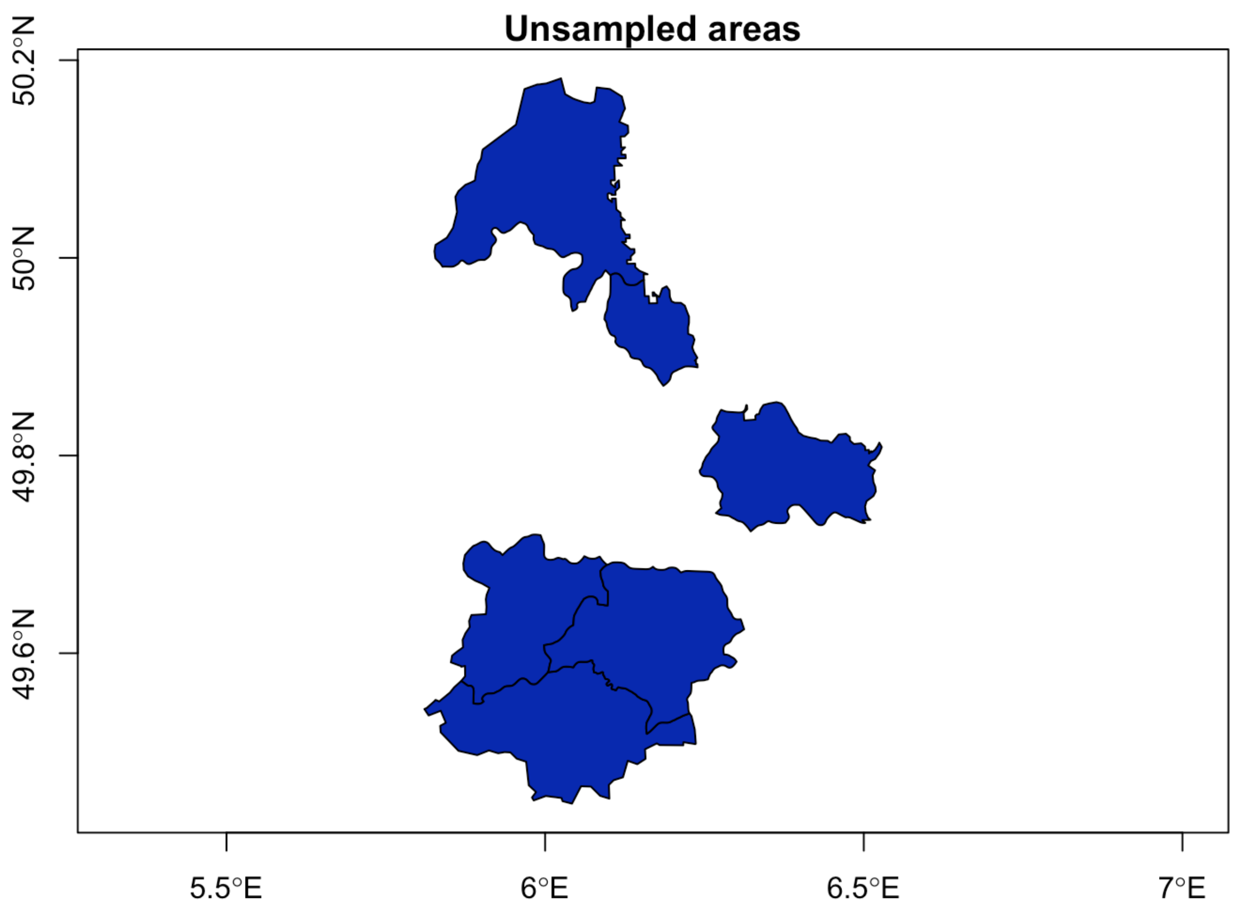


You can easily remove the column you just added above, using this command

```
lux$sampled_regions <- NULL
```

But for now let's add the new column again and selectively only plot those areas that have not yet been sampled. We can do that by finding the indexes `i` for the regions where our new column has the value 0 (using the `which()` function and then extracting all of those rows from the spatial data object:

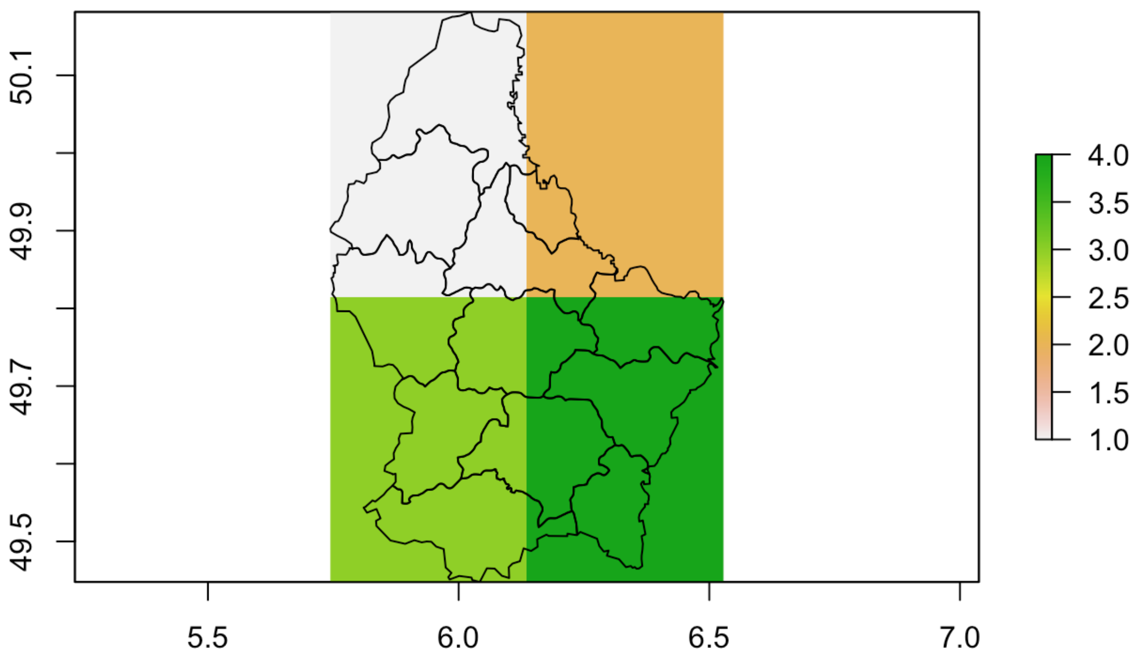
```
new_values = c(0,1,1,0,1,0,1,1,0,0,0,1)
lux$sampled_regions = new_values
i <- which(lux$sampled_regions == 0)
g <- lux[i,]
plot(g['sampled_regions'],main='Unsampled areas',axes = T)
```



Creating raster over polygon and convert raster back to polygon

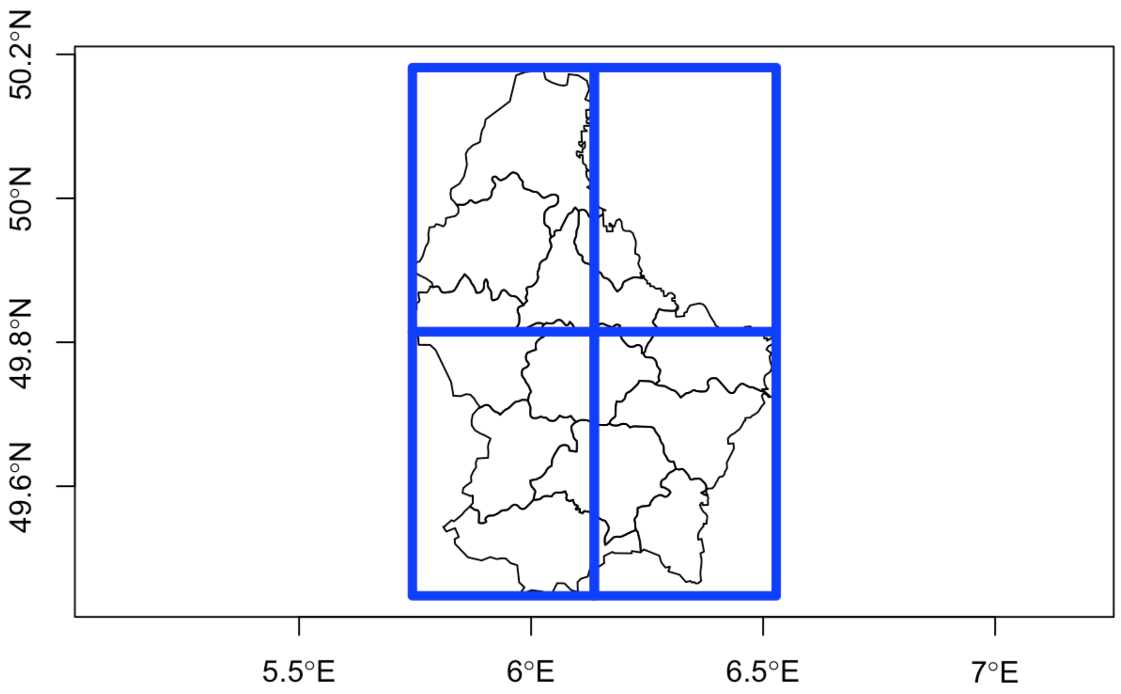
Let’s say we want to divide our Luxembourg polygon into 4 equally sized cells. The easiest way to do this is to define a raster with 4 cells that encompasses the whole polygon. You cna actually provide the raster function with a spatial polygon and it will automatically create a raster using the bounding box coordinates fo the polygon (note that we are using the spatial object `lux_spatial` again, not the sf object `lux`):

```
z <- raster(lux_spatial, nrow=2, ncol=2, vals=1:4)
plot(z)
plot(lux_spatial,add=T)
```



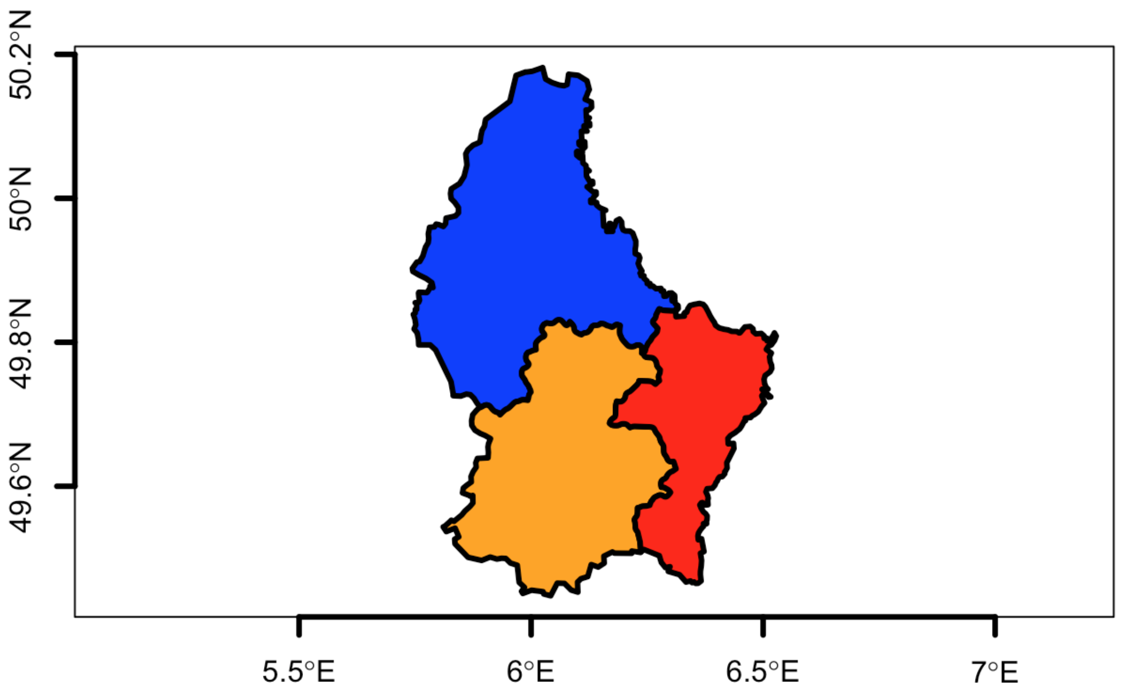
You can turn the grid of the raster into a spatial object itself, using the `as()` function, just as we did for the sf objects before. Let's do that and plot it on top of the Luxembourg polygon:

```
z_spatial = as(z, 'SpatialPolygonsDataFrame')
plot(lux_spatial,axes = T)
plot(z_spatial, add=TRUE, border='blue', lwd=5)
```



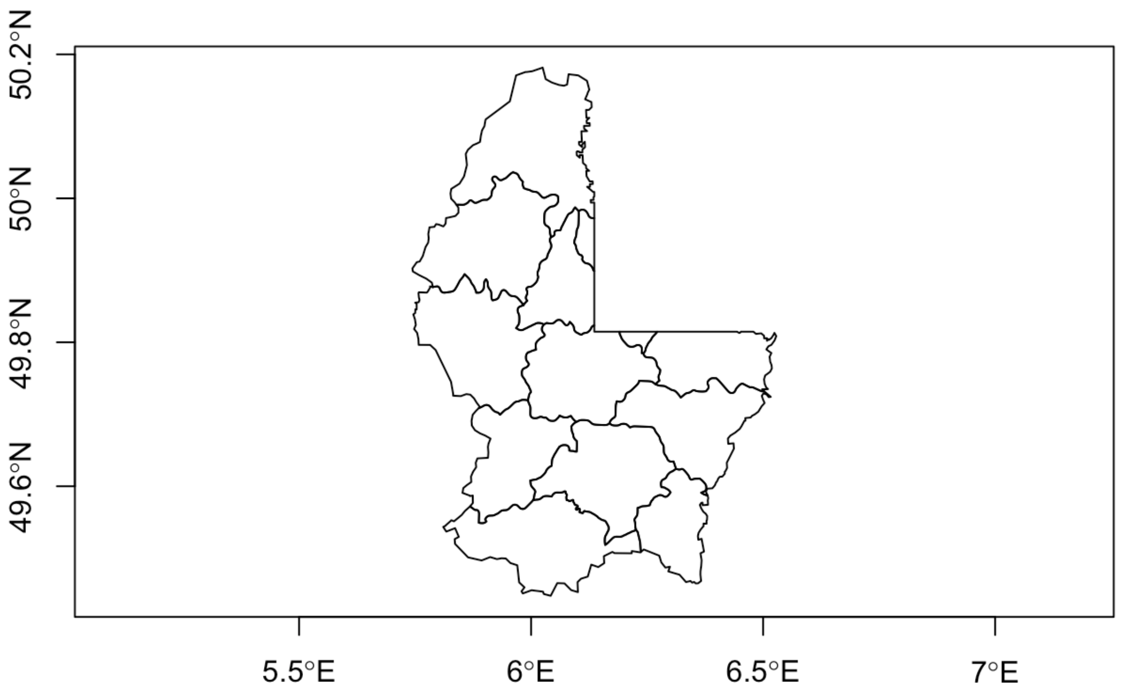
There are several useful functions when working with polygons, e.g. the `aggregate()` function, which can be used to join polygons based on their values. Let's use that function to join the area polygons by their values in the `NAME_1` column (larger regions):

```
counties = aggregate(lux_spatial, by='NAME_1')
plot(counties, col=c('blue','red','orange'), lwd=3, border='black',axes = T)
```



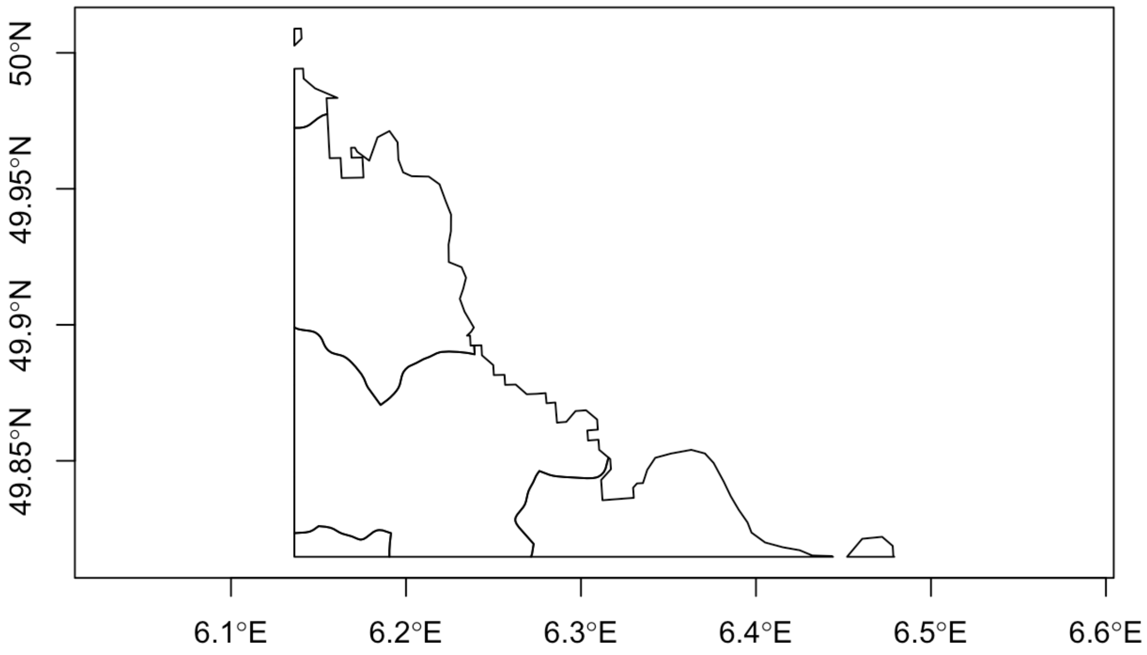
We can also determine the overlap of multiple spatial objects. E.g. we can remove the overlap between two polygons using the `erase()` functions. Here we remove the overlap between the map of Luxembourg and the second cell from our grid that we defined above:

```
e <- erase(lux_spatial, z_spatial[2,])
plot(e,axes = T)
```



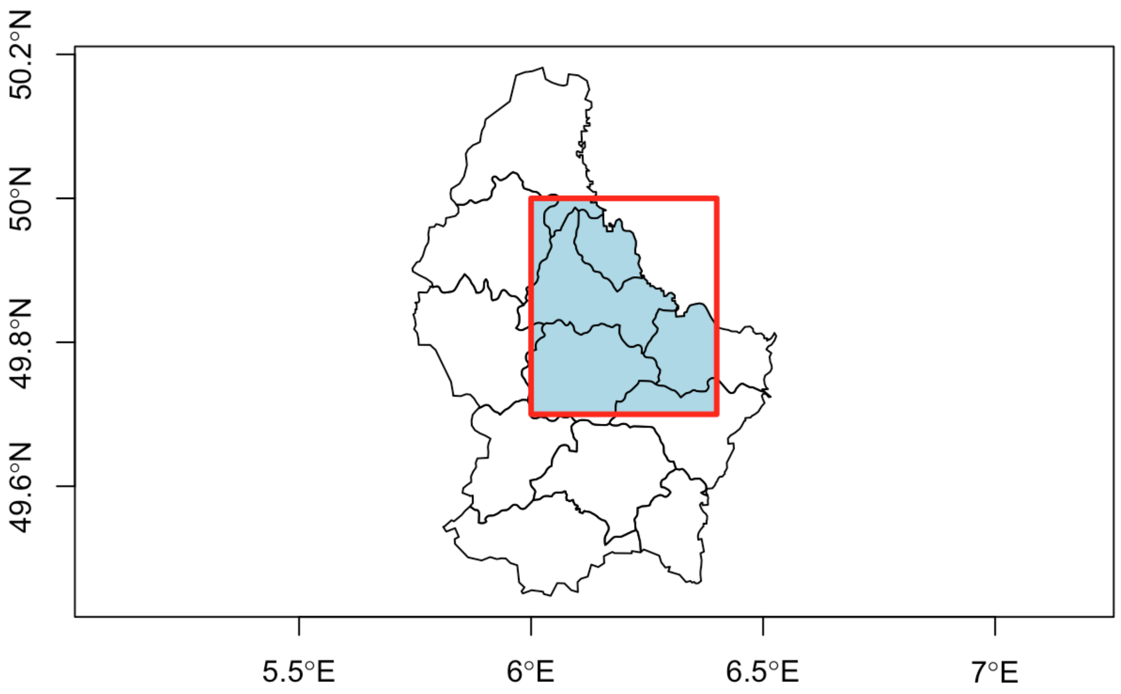
The `intersect()` function on the other hand does basically the opposite and keeps only what's shared between two polygons. Using this function we can e.g. extract the part of the map of Luxembourg that falls into our second cell of the made-up grid:

```
i <- intersect(lux_spatial, z_spatial[2,])
plot(i,axes = T)
```



We can also simply define a rectangle (here we made up some coordinates that are within the range of the coordinate system of the Luxembourg) and extract the region covered by that rectangle. We can use the `crop()` (or the `intersect()`) function to extract the region covered by the rectangle. We define this to a separate variable (`pe`) and then plot it in a different color on top of the rest of the map (in blue). Add the end we also plot the actual rectangle on top of everything (in red).

```
e <- extent(6, 6.4, 49.7, 50)
pe <- crop(lux_spatial, e)
plot(lux_spatial,axes = T)
plot(pe, col='light blue', add=TRUE)
plot(e, add=TRUE, lwd=3, col='red')
```



Another polygon-overlap function is the `union()` function, which can be used to merge the spatial information stored in two spatial objects. As you can see in the plot below, this function creates new polygon objects from the two input objects which are defined by the borders of both of these objects. E.g. a region that was defined within out `lux_spatial` object is split into two, if an edge of the vectorized grid-object `z_spatial` runs through it.

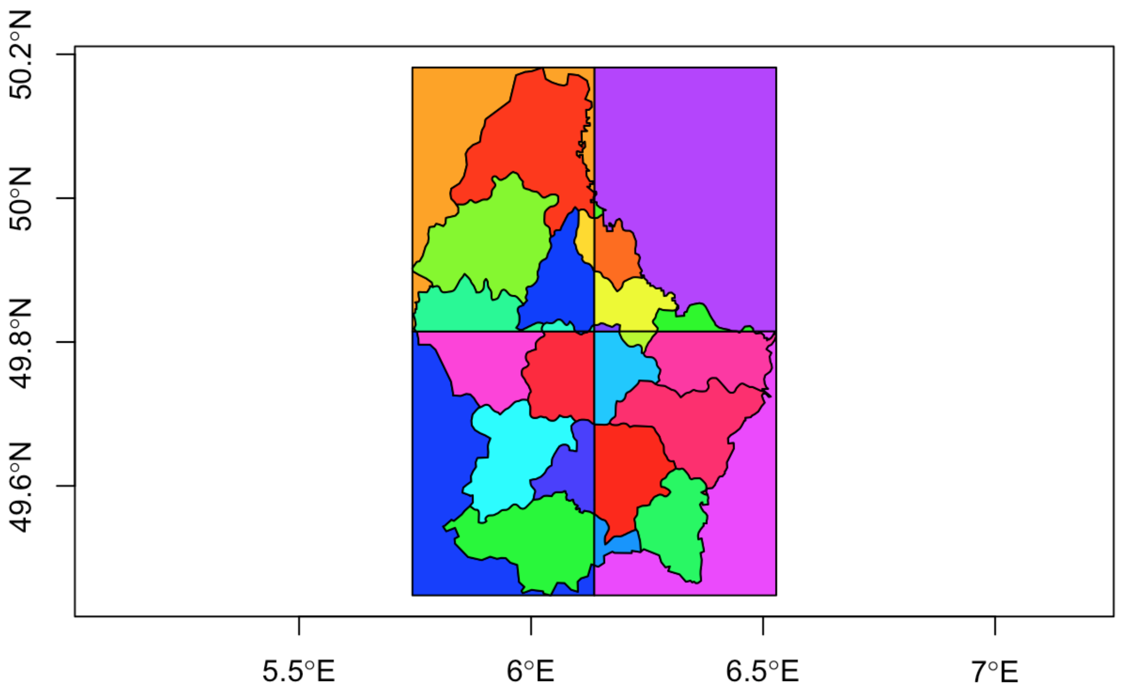
```
u <- union(lux_spatial, z_spatial)
u
```

```
## class       : SpatialPolygonsDataFrame
## features    : 28
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, yma
x)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,
0,0
## variables   : 6
## names       : layer, ID_1,     NAME_1, ID_2,   NAME_2, AREA
## min values  :     1,    1,   Diekirch,    1, Capellen,   76
## max values  :     4,    3, Luxembourg,   12,    Wiltz,  312
```

You can see that this created 28 different polygons (compared to the 12 we had before). To make them better visible let’s plot them in different colors.

Assigning random colors: When you have a bunch of different polygons stored in a `SpatialPolygonsDataFrame` object and you want to plot them all in different colors, it can be quite painstaking to manually define colors for each and single one of them. An easy way around that is to use a random color generator. E.g. you can sample 100 random colors from the whole color spectrum using the `sample(rainbow(100))` command. If you want to use this to assign colors to polygons in a `SpatialPolygonsDataFrame` the lost of colors needs to have the same length as the number of polygons in your dataframe. To achieve this you can use the `length()` function to get the exact number of polygons in your spatial object: `sample(rainbow(length(name_of_spatial_object)))`

```
plot(u,col=sample(rainbow(length(u))),axes = T)
```

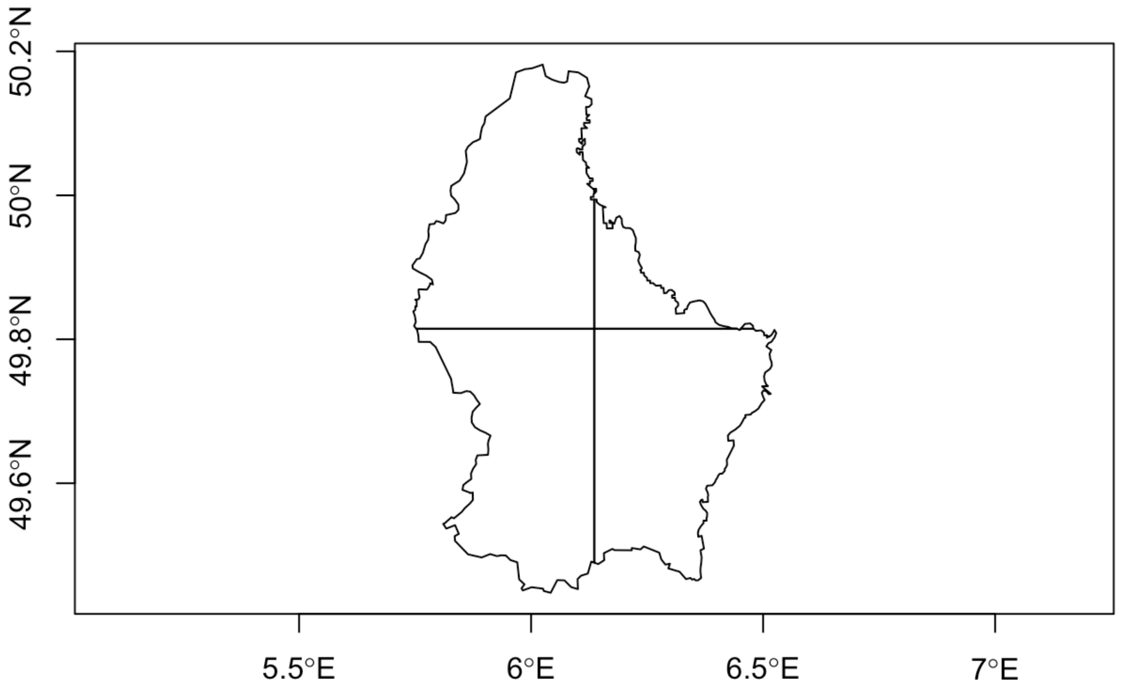


The `cover()` function is a combination between the `intersect()` and `union()` functions. It takes the outer boundary of the polygon of the first object and intersects it with the second object, defining new polygons based on the intersect.

```
cov <- cover(lux_spatial, z_spatial)
cov
```

```
## class       : SpatialPolygonsDataFrame
## features    : 6
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, yma
x)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,
0,0
## variables   : 6
## names       : ID_1,       NAME_1, ID_2,     NAME_2, AREA, layer
## min values  :    1,     Diekirch,    1,   Clervaux,  188,     1
## max values  :    2, Grevenmacher,    6, Echternach,  312,     4
```

```
plot(cov,axes = T)
```



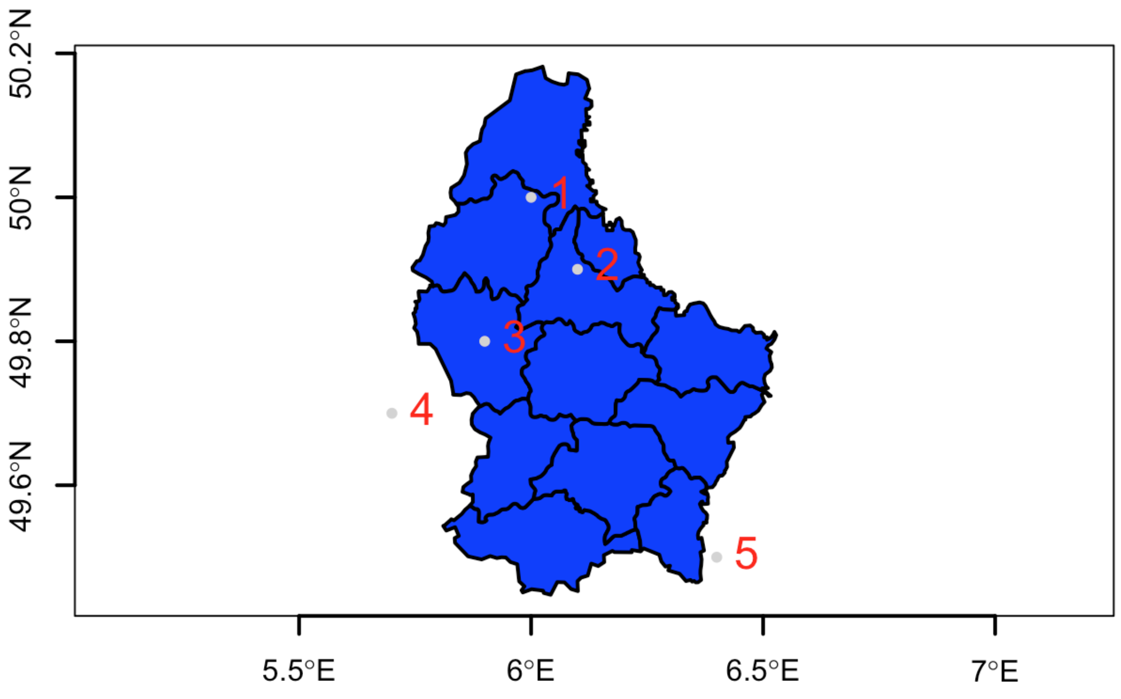
Spatial queries

Let's define a set of 5 points and then check which of the polygons they fall into. First we define the points and convert them into a `SpatialPoints` object:

```
lon = c(6, 6.1, 5.9, 5.7, 6.4)
lat = c(50, 49.9, 49.8, 49.7, 49.5)
pts <- cbind(lon,lat)
pts
```

```
##      lon  lat
## [1,] 6.0 50.0
## [2,] 6.1 49.9
## [3,] 5.9 49.8
## [4,] 5.7 49.7
## [5,] 6.4 49.5
```

```
spts <- SpatialPoints(pts)
plot(lux_spatial, col='blue', lwd=2,axes=T)
points(spts, col='light gray', pch=20)
text(spts, c(1,2,3,4,5), col='red', cex=1.5,pos=4)
```



Now we can use the `over()` function in order to check if and where each point in our `SpatialPoints` object intersects with the `lux_spatial` polygon. The function will return a dataframe that shows for each point the values of the intersecting polygon dataframe. It will return the value `NA` for all data columns if the point did not match any of the polygon

The `over()` function will complain if the two objects don't have the same assigned coordinate reference system (CRS). We therefore first need to assign the correct coordinate reference system to the `SpatialPoints` object by using the `proj4string=` setting in the `SpatialPoints()` command. We will assign the same reference system as the Luxembourg shape object, which we can call by using the `crs()` function.

```
spts <- SpatialPoints(pts, proj4string=crs(lux_spatial))
my_over_output = over(spts, lux_spatial)
my_over_output
```

```
##   ID_1   NAME_1 ID_2   NAME_2 AREA
## 1    1 Diekirch    5    Wiltz  263
## 2    1 Diekirch    2 Diekirch  218
## 3    1 Diekirch    3  Redange  259
## 4   NA     <NA>   NA     <NA>   NA
## 5   NA     <NA>   NA     <NA>   NA
```

Task: Plot only the regions that contain a point, based on the `over()` output. The final plot should look like the one below:

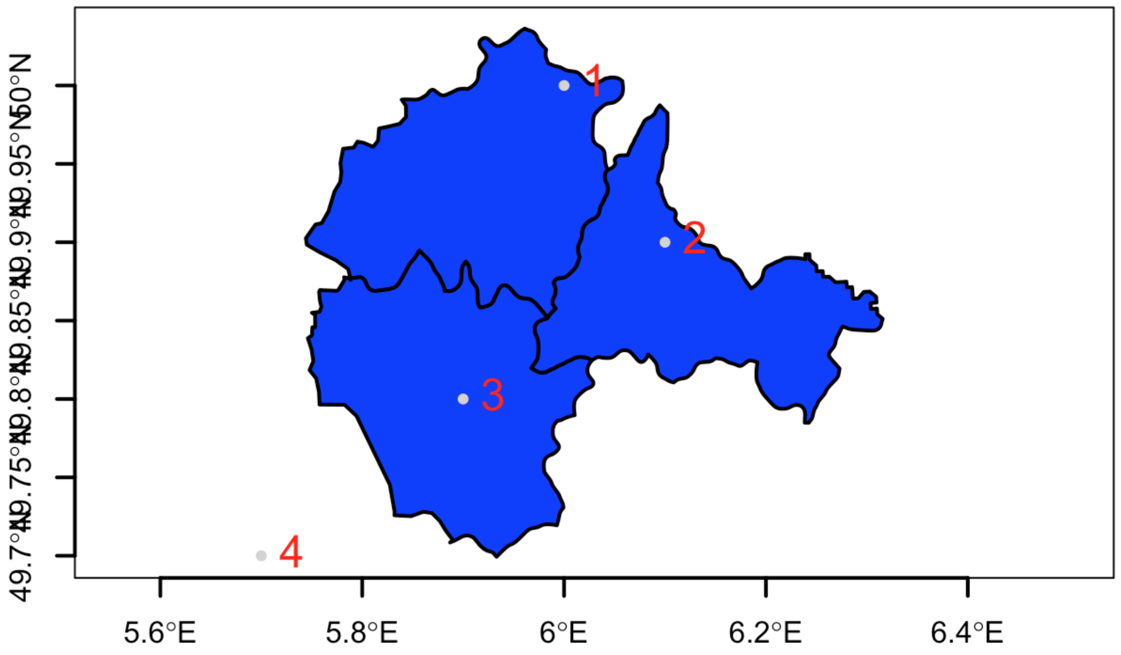
Tip: Depending on how familiar you are with R or other programming languages, you may be able to do this without the instructions below. Maybe first give it a shot and if you get stuck, have a look at the instructions below.

One approach would be:

1. Extract the list of region names of the matching polygons from the `over()` function output (make sure you extract a data column that enables you to uniquely identify each region). You extract a specific data column using the `$column_name` notation following the name of the object, e.g. `my_over_output$ID_1` extracts the column `ID_1` from the `my_over_output` object.

2. Identify which of the polygons in the `lux_spatial` object match this list of target regions. You can use the `which()` function to determine the polygon indeces that fulfil a given requirement. In order to define a requirement you can use the `%in%` notation to test which elements from one list match with elements from another, e.g. `which(lux_spatial$NAME_2 %in% my_list_of_target_regions)` returns the indeces of those polygons, whose name is present in the list `my_list_of_target_regions` .

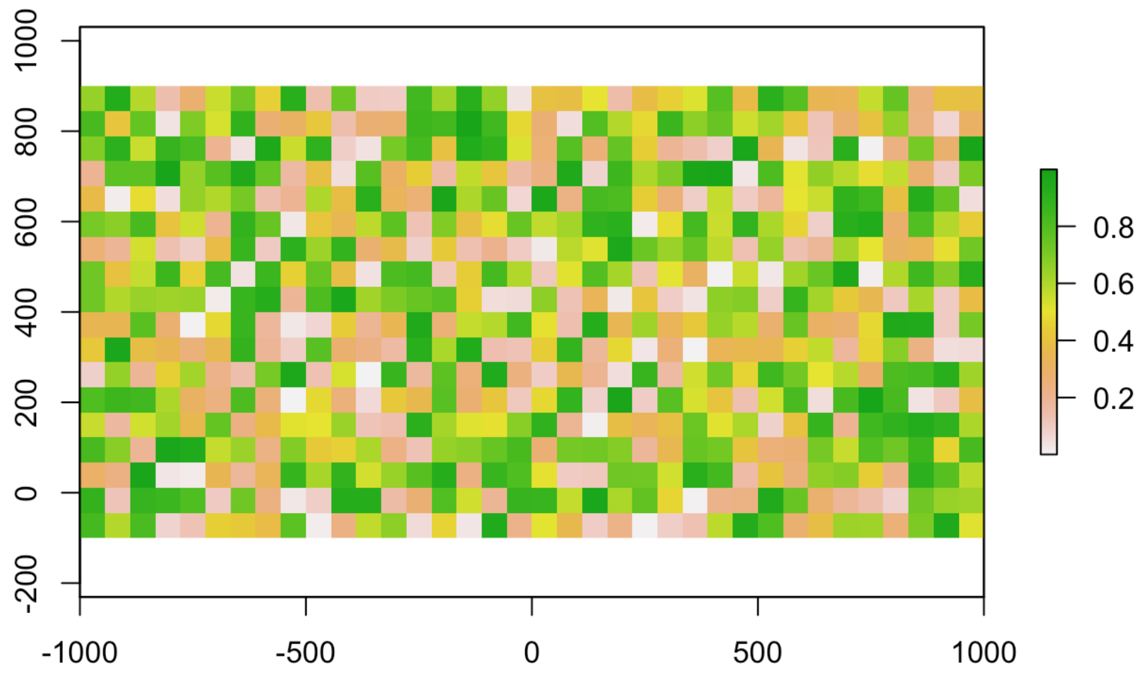
3. Extract the target polygons from the `lux_spatial` object. Use the indeces that where produced by the `which()` command to index the polygons you want to extract, e.g. `lux_spatial[list_indeces,]` extracts all indeces present in the list `list_indeces` .



6. Working with raster data

It’s rather straight-forward to generate a raster in R. As we did earlier in this tutorial we can just create a raster by using the `raster()` command and specifying the number of cells and extent in x and y direction. Remember that this command only creates the skeleton (empty raster). Here we just fill these cells with random numbers between 0 and 1 using the `runif()` command.

```
x <- raster(ncol=36, nrow=18, xmn=-1000, xmx=1000, ymn=-100, ymx=900)
values(x) <- runif(ncell(x))
plot(x)
```



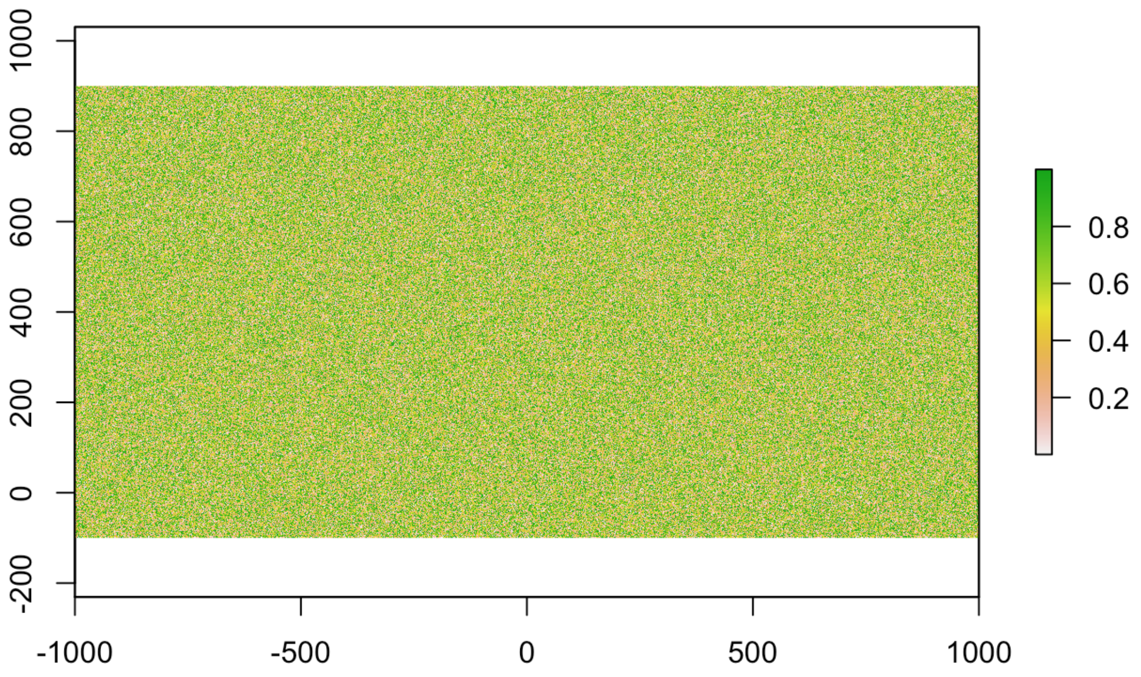
You can check the resolution of the raster (dimensions of each cell) using the `res()` command:

```
res(x)
```

```
## [1] 55.55556 55.55556
```

It turns out each of our cells has the size 55.5x55.5, i.e. it covers 55.5 units on the x-axis and 55.5 untis on the y-axis. You can change the resolution of the raster. Let’s change it to `1`, which will lead to many more cells in the raster, as the cells will now only have the size 1x1 and fill the cells with random values.

```
res(x) <- 1
values(x) <- runif(ncell(x))
plot(x)
```



We can access the coordinate projection of the raster using the `projection()` function and can assign a value to it (see this link from earlier (http://rspatial.org/spatial/rst/6-crs.html) to check again the notation for different coordinate reference systems). There will be a more thorough explanation of how to decide which projection to assign to your spatial object in the next tutorial.

```
projection(x) <- "+proj=utm +zone=48 +datum=WGS84"
projection(x)
```

```
## [1] "+proj=utm +zone=48 +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
```

Summary of some useful basic raster commands:

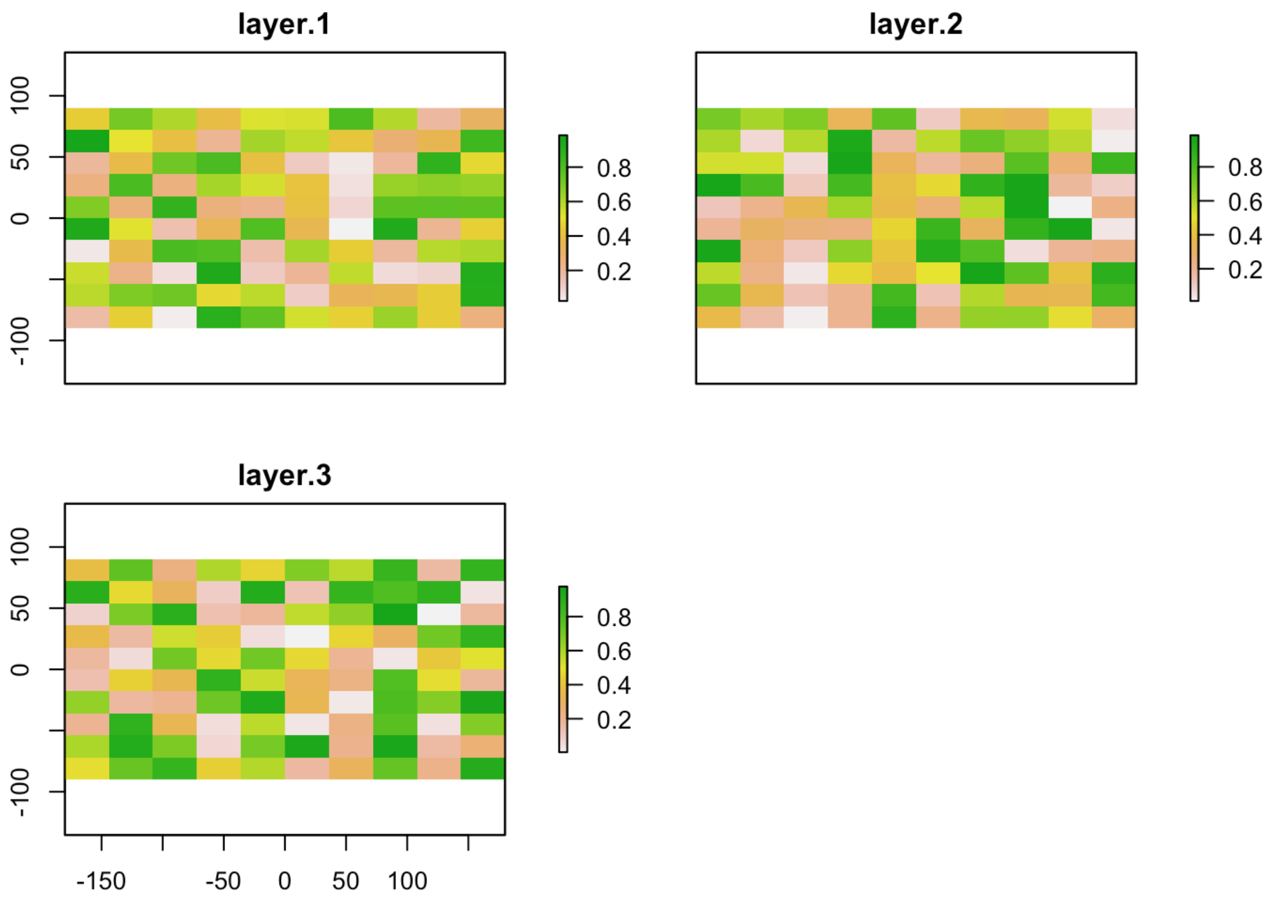
- **res()** - access the raster resolution (cell dimensions)
- **projection()** - access the coordinate projection of the raster
- **ncell()** - access the number of cells in the raster
- **ncol()** - access the number of columns in the raster
- **dim()** - access the dimension of the raster (cells in x and y direction)
- **values()** - access the values stored in the raster cells
- **hasValues()** - check if raster contains values or if it’s just an empty skeleton
- **xmax()** - access the maximum x value
- **xmin()** - …
- **ymax()** - …
- **ymin()** - …

We can stack multiple rasters into the same object by using the `stack()` command. Here we define 3 rasters with random numbers and stack them:

```
r1 <- r2 <- r3 <- raster(nrow=10, ncol=10)

# Assign random cell values
values(r1) <- runif(ncell(r1))
values(r2) <- runif(ncell(r2))
values(r3) <- runif(ncell(r3))

s <- stack(r1, r2, r3)
plot(s)
```

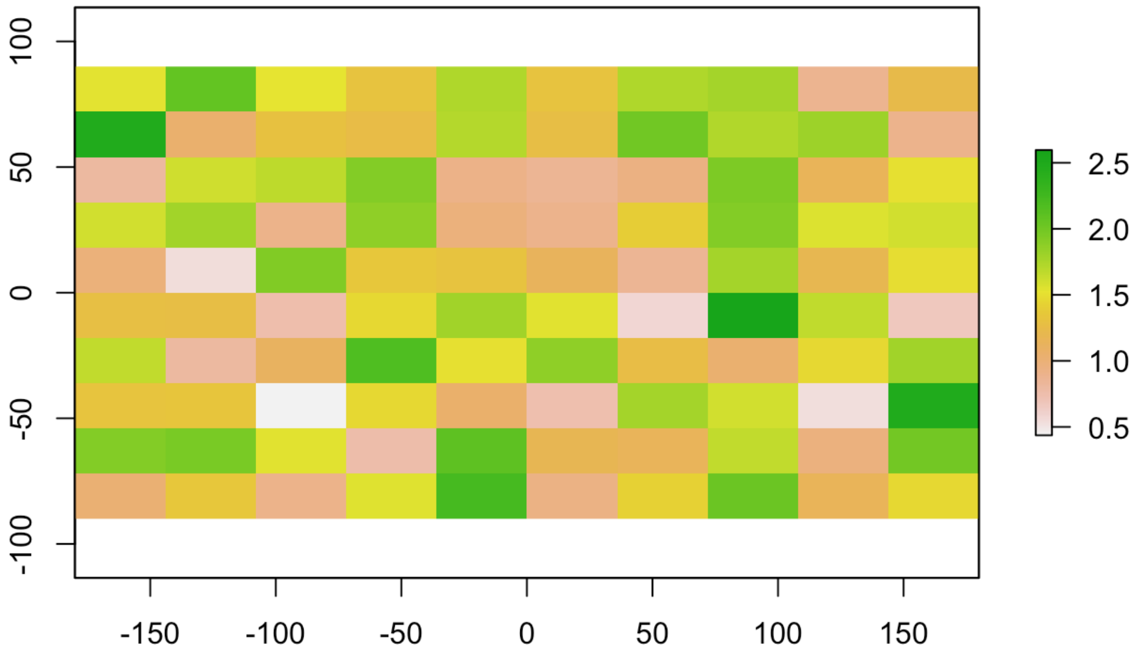


Algebraic operations with rasters

Rasters are very straightforward to work with, since they allow the use of simple algebraic operators such as +, -, *, /, as well as logical operators such as >, >=, <, ==, !.

For example if we want to add the values across all 3 rasters from above we can simply do that like this:

```
sum_rasters = r1 + r2 + r3
# or:
sum_rasters = sum(r1, r2, r3)
plot(sum_rasters)
```



Congratulations, you made it through the first tutorial! Here you can move on to the next tutorial (./tutorial_2.html), using real data.