Spatial R course day 1. Introduction to spatial analyses

30/08/2021

Introduction to spatial data in R

(This tutorial is a modification of an earlier version written by Tobias Andermann, most of the hard work is his, most of the errors are mine). Libraries only have to be loaded once in an R-session. In these tutorials I however load them a in each coding block to make it clear what specific library we are using at each step.

Dependencies:

```
library(sp)
library(raster)
library(sf)
library(rgeos)
library(rgdal)
```

If you are completely new to R, have a look at this general R introduction tutorial. 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 three different types of spatial data: **points, polygons and rasters**.

Points are used when we are analyzing specific occurrences with a latitude and longitude. This could be point observations, sample sites, e.t.c.

The other two classes are used to describe areas rather than specific sites:

Polygons are defined by the edges and describe something characterizing everything within the polygon. This can be species ranges, islands, country borders, borders of a forest etc.

Rasters are applied to represent continuous phenomena, or "spatial fields", e.g. elevation, temperature, or species diversity. It can be compared to an air photograph where each pixel has a value and corresponds to a given dimension in the x and y axis. Rasters can be seen as a computationally more efficient way to treat a specific type of polygon data and all rasters can be converted to polygons without any loss of information but doing so will be computationally demanding for even moderately sized rasters.

1. Point and polygon data

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 latitude 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
shared 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, body size) is to join these vectors into one data frame in order to keep it together and sorted.

```
# join data in a single dataframe
Fake_Data = data.frame(longitude, latitude, name, body_size)
Fake_Data

##    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, and we will return to this several times during the week. Within the square brackets, you can specify the rows and columns to extract, using the following indexing syntax `[rows,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 rows and columns, in this case we leave the lines part blank, which will lead to all lines being extracted):

```
Fake_Data[,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 to extract the first three rows and all columns for these lines you can type:

```
Fake_Data[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:

```
Fake_Data$body_size
```

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

Finally, it is worth noting that you can extract the data full filling a specific set of logical criteria. This would e.g. select the rows fulfilling the logical criteria that the body size is larger than 15.

```
Fake_Data[Fake_Data$body_size>15,]
```

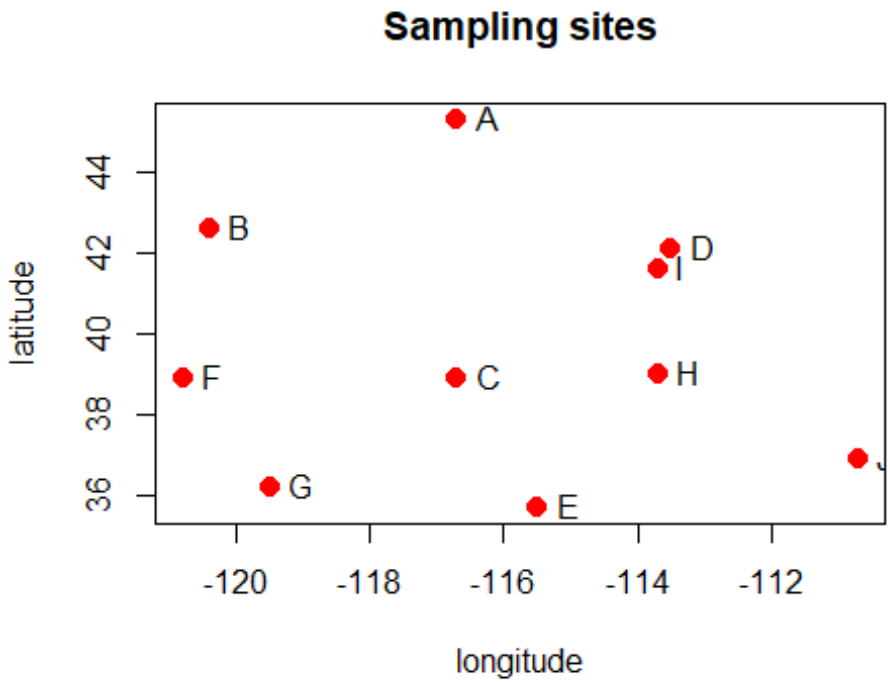
```
##    longitude latitude name body_size
## 3     -116.7     38.9    C        17
## 4     -113.5     42.1    D        19
## 5     -115.5     35.7    E        22
## 7     -119.5     36.2    G        21
## 10    -110.7     36.9    J        18
```

Now, let's plot the data 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), and `col` where you can define the color of the points. Colors can be defined by name for a relatively large number of colors or alternately in HEX using numbers and small letters where the reddest possible e.g. would be called "ff0000". You can e.g. see Colorbrewer for inspiration about colors to try.

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. Other settings worth exploding is adding **bty="n"** which removed the (in my mind) annoying black box around the figure or **xlim** and **ylim** which can change the dimensions of the plotting window. 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 preceding `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.

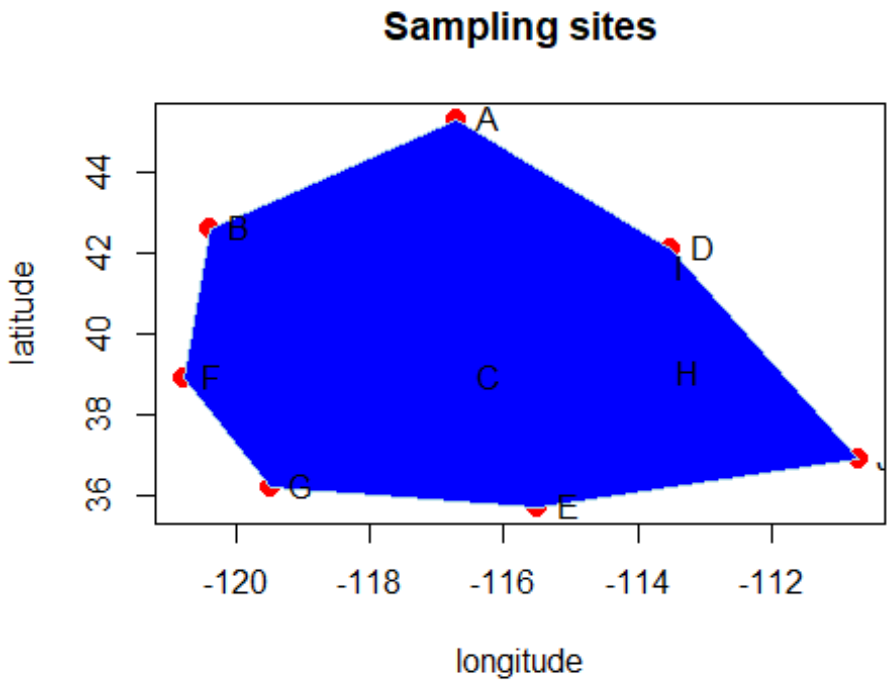
Polygons

We can also plot a **polygon** from the same data. For this we will first identify the convex hull which is the smallest area connecting all records in sample sites as straight lines. (The code here is a simplification and as we will return to on Wednesday it is marginally wrong because we do not account for the shape of the earth). We will first identify the edges of a polygon including all sample sites using `chull()`

```
chull(sampling_sites)

## [1] 10  5  7  6  2  1  4
```

After this we will plot a polygon connecting these sample sites. Note the use of the square bracket we discussed earlier

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



Defining spatial objects

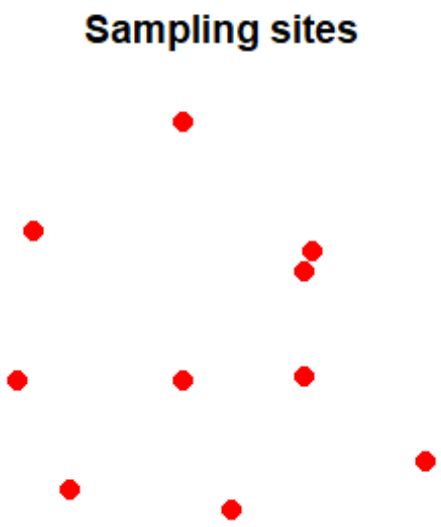
So far, we have only worked with simple data points stored in arrays. However, different data object types exist that are specifically designed to handle spatial data. Some of the most commonly used objects are defined in the `sp` package. For vector data, the basic types we will use are `SpatialPoints()` 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 the content of the object (works for
any 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 want the x-axis and y-axis to show). If you do want to add axes to your plot the command `axis()` lets you add axes to an existing plot and gives you a substantial control over how it can be done.

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



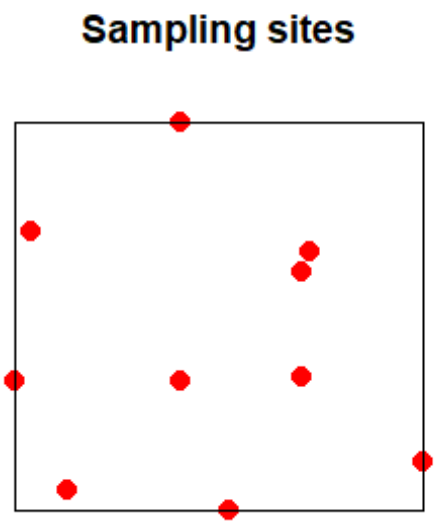
You can extract the additional values stored in the `SpatialPoints()` object, e.g. the coordinates of the bounding box, like this:

```
library(sp)
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:

```
library(sp)
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
```

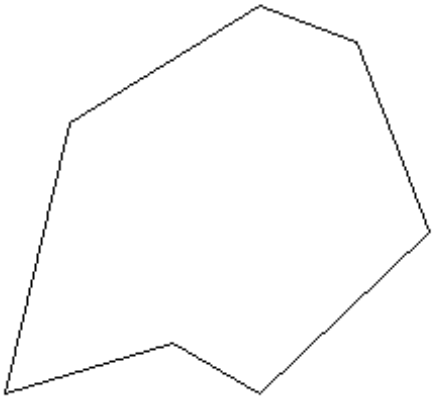
Now we'll use a different function to create a spatial polygon using the `spPolygons()` function. For this we first need to load the package `raster`. We'll also make up some new data, in order to have different data points than for the `SpatialPoints` object from above:

```
# make up new data
library(raster)
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
pol = spPolygons(lonlat)
pol
```

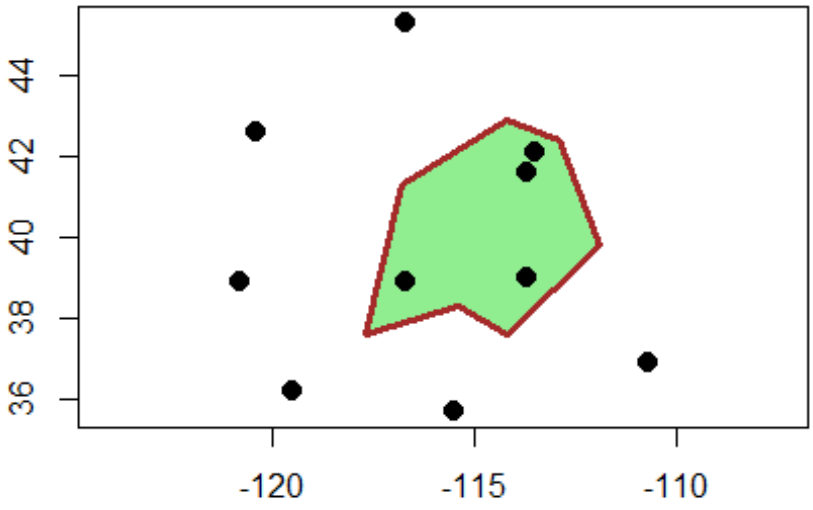
```
## class       : SpatialPolygons
## features    : 1
## extent      : -117.7, -111.9, 37.6, 42.9  (xmin, xmax, ymin, ymax)
## crs         : NA
```

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

```
plot(pol)
```



```
plot(pts, col="black", pch=20, cex=2,axes=TRUE)
plot(pol, 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's work 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 where real data is analyzed, these objects are read from a file, or created by conversion of data in other formats like e.g. polygons. 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)
## crs        : +proj=longlat +datum=WGS84 +no_defs
```

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()`:

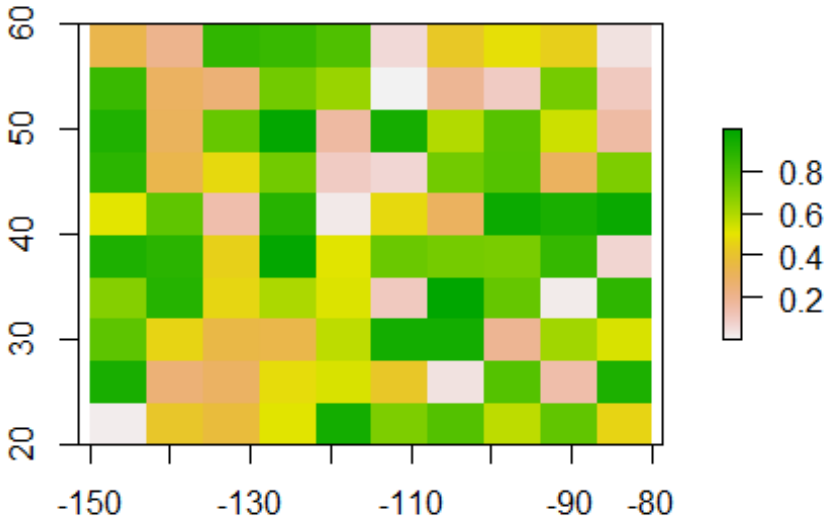
```
library(raster)
ncell(rast)

## [1] 100
```

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

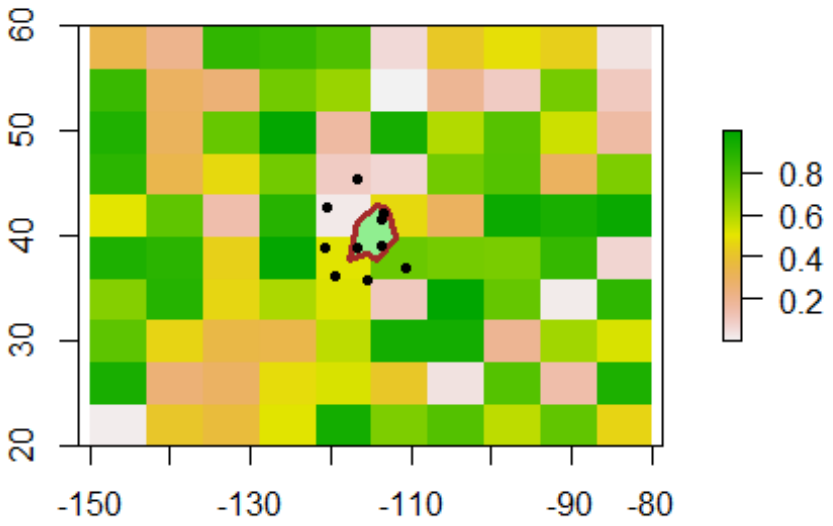
```
library(raster)
raster_values = runif(ncell(rast))
raster_values

##   [1] 0.339441154 0.208799341 0.869290028 0.851459981 0.793368434
0.058699856
##   [7] 0.421766845 0.489446728 0.447616328 0.038233457 0.847613038
0.295960065
##  [13] 0.256308827 0.719308938 0.641502787 0.004290845 0.194033718
0.096875538
##  [19] 0.712120329 0.106520254 0.912267604 0.310962068 0.744504768
0.992212833
##  [25] 0.165766557 0.941704874 0.596863768 0.779897395 0.550126119
0.161891553
##  [31] 0.882334693 0.339173121 0.472631837 0.720792478 0.099115843
0.067887592
##  [37] 0.721135085 0.782485622 0.300388940 0.697821596 0.510073599
0.760989281
##  [43] 0.143974955 0.893433733 0.021931402 0.475881952 0.300280014
0.967250860
##  [49] 0.929637281 0.971714810 0.916263712 0.883448105 0.450260920
0.989795926
##  [55] 0.513004449 0.735610253 0.714091167 0.705148983 0.857352224
0.070697779
##  [61] 0.681749310 0.896953596 0.467462383 0.607808318 0.521424304
0.106128443
##  [67] 0.998632984 0.748662542 0.021676325 0.872402011 0.763566899
0.461506296
##  [73] 0.354043632 0.345309692 0.575595442 0.945466141 0.943046801
0.195833623
##  [79] 0.626349802 0.528361816 0.933319005 0.253128927 0.294007037
0.482243472
##  [85] 0.527046975 0.420166134 0.040268634 0.782510296 0.150164694
0.918768750
##  [91] 0.017006392 0.413897425 0.375383548 0.513440752 0.941884025
0.697945492
##  [97] 0.786189755 0.575594833 0.754017439 0.461955233
```

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

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



Task 1: 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 `pol` 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 `st_read()` function in order to read a shape file as a vectorized polygon into R. A shape file is a weird name for the object since it is actually made up of multiple files having the same name but different suffixes. Usually at least four; the .SHP, the .DBF, the .PRJ and the .SHX. If you ever encounter issues reading shapefiles, the most common reason is that one or more of these files are missing.

- **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 Sweden, originally downloaded from DIVA-GIS. 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. 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 file path 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 into the parent directory. Alternatively, you can use list.files() to see what files are in whatever folder you are currently reading from and use this this to guide you. Another alternative is to find the relevant folder in file explorer or finder and copy the path as text into R. **Note that the way to specify paths vary depending on operating system. I am personally using windows and in the logic above I would need to use "\" rather than "/"**

```
library(sf)
Sweden = st_read("C:\\Bpoxsync\\Spatial_R\\Data\\SWE_adm\\SWE_adm0.shp")

## Reading layer `SWE_adm0' from data source
`C:\Bpoxsync\Spatial_R\Data\SWE_adm\SWE_adm0.shp' using driver `ESRI
Shapefile'
## Simple feature collection with 1 feature and 70 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 10.96139 ymin: 55.33625 xmax: 24.1724 ymax: 69.05904
## Geodetic CRS:  WGS 84
```

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 an `sf` object called Sweden:

```
Sweden = as(Sweden, "Spatial")
Sweden

## class       : SpatialPolygonsDataFrame
## features    : 1
## extent      : 10.96139, 24.1724, 55.33625, 69.05904  (xmin, xmax, ymin,
ymax)
## crs         : +proj=longlat +datum=WGS84 +no_defs
## variables   : 70
## names       : ID_0, ISO, NAME_0, OBJECTID_1, ISO3, NAME_ENGLI, NAME_ISO,
NAME_FAO, NAME_LOCAL, NAME_OBSOL, NAME_VARIA, NAME_NONLA, NAME_FRENC,
NAME_SPANI,                                          NAME_RUSSI, ...
## value       :  222, SWE, Sweden,        215,  SWE,     Sweden,   SWEDEN,
```

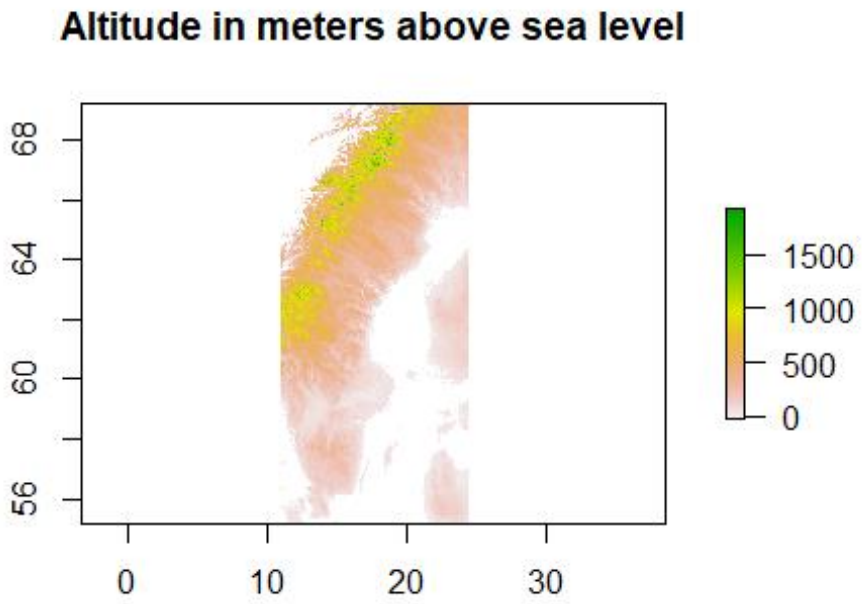
```
Sweden,    Sverige,         NA,     Sweden,         NA,      Suède,
Suecia, <U+0428><U+0432><U+0435><U+0446><U+0438><U+044F>, ...
```

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

```
plot(Sweden,main="Sweden")
```

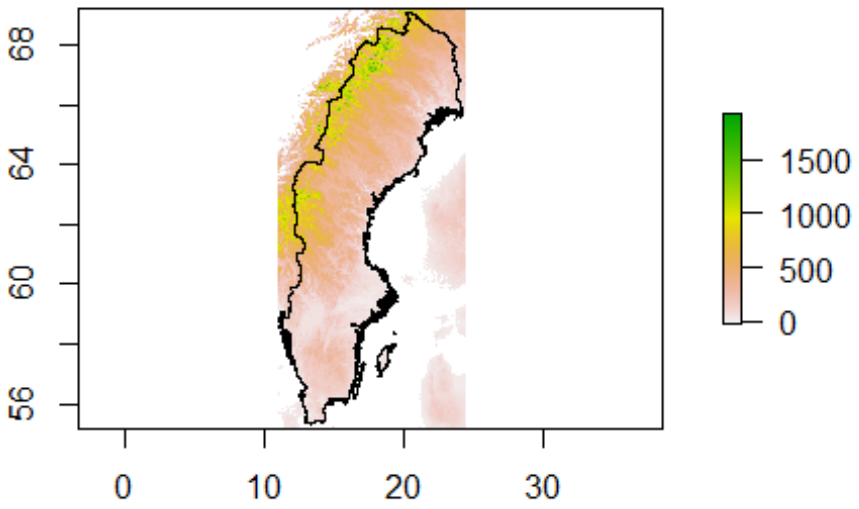
Similarly we can read raster data, using the `raster()` function. In this example we are reading altitude information covering Sweden, which we will use in combination with the shape file from above. There are multiple different file formats for rasters. The one we load here is a ".rgd" format which is a format splitting the information into separate files like we saw with shapefiles, but many other formats like ".tif" only relies on a single file. When creating your own data it is generally advisable to use such formats to reduce confusion among the end-users of your data:

```
library(raster)
Sweden_alt = raster("C:\\Bpoxsync\\Spatial_R\\Data\\SWE_alt\\SWE_alt.gri")
plot(Sweden_alt,main="Altitude in meters above sea level")
```



You may get a warning here. The underlying reason is substantially more complex than what we will deal with at this course. People clever enough can read details here (but please do not ask me any too hard questions).

Task 2: 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 Sweden (stored in the `Sweden` 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 Sweden 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). Later in the tutorial we will work through and where the CRSs between two objects don't match, and where they have to be adjusted first.

5. Working with vector data

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

Let's first read in the file "SWE_adm2" from the same path as before, convert it to a spatial object and give it the name "Sweden_2". You should be able to do that based on the commands above with few changes. After this we will plot it. If you succeed you should get something like this

```
## Reading layer `SWE_adm2' from data source
`C:\Bpoxsync\Spatial_R\Data\SWE_adm\SWE_adm2.shp' using driver `ESRI
Shapefile'
## Simple feature collection with 286 features and 11 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 10.96139 ymin: 55.33625 xmax: 24.1724 ymax: 69.05904
## Geodetic CRS:  WGS 84
```

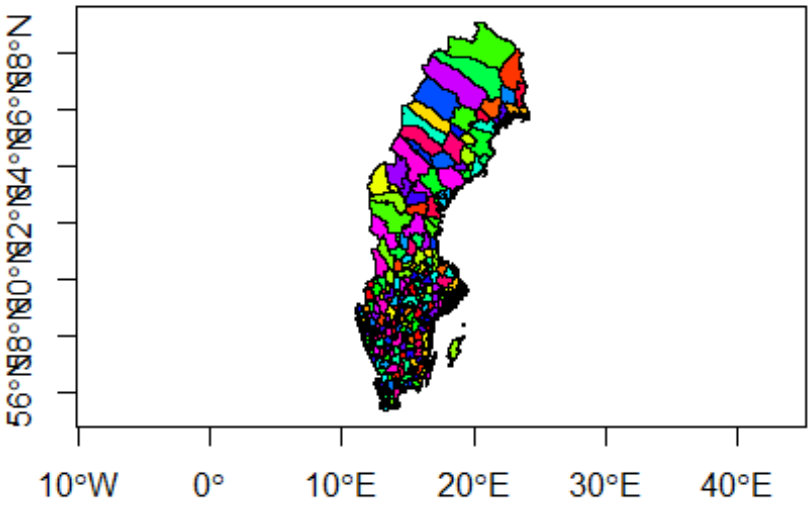
The shape object is formatted as a `SpatialPolygonsDataFrame`. Similarly to the `SpatialPointsDataFrame` from earlier in the tutorial, a `SpatialPolygonsDataFrame` contains a list of `SpatialPolygons`, as well as other data associated with these polygons. This could for example be the polygons of different 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: `Sweden_2`), that there are 286 features in this object. That means that this `SpatialPolygonsDataFrame` contains 286 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(Sweden_2)
```

```
##  [1] "ID_0"      "ISO"       "NAME_0"    "ID_1"      "NAME_1"    "ID_2"
##  [7] "NAME_2"    "TYPE_2"    "ENGTYPE_2" "NL_NAME_2" "VARNAME_2"
```

There are eleven different values associated with each polygon which are stored in columns. We can plot each of the regions in a separate color, using the command col. We will here just use 286 colors from the rainbow scale. The regions are ordered inside Sweden_2 so neighboring regions have similar values and colors close to each other on the rainbow scale looks very similar. In order to make the contrasts clearer we will therefore randomize the order of the colors which we will do with sample :

```
plot(Sweden_2, col=sample(rainbow(286), 286), axes=T, lwd=0.1)
```



We can extract the values of specific columns like this. Note that we use the command **head()** which means that only the first rows are plotted. This is normally a nice way to get a grasp of the structure of large objects in R. Note also that we here use two functions on the same line. There is no upper limit to how many functions you can call on the same line but in order to produce codes that can be understood by yourself or others it is normally advisable to only call one or at most two functions in one line. Bad coding habits are hard to quit (my internal codes regularly calls 3,4 or more functions in one line) but it is good to learn proper simple coding as soon as possible:

```
head(cbind(Sweden_2$NAME_2, Sweden_2$NAME_1))
```

```
##      [,1]         [,2]
## [1,] "Ödeshög"    "Östergötland"
## [2,] "Åtvidaberg" "Östergötland"
## [3,] "Boxholm"    "Östergötland"
## [4,] "Finspång"   "Östergötland"
## [5,] "Kinda"      "Östergötland"
## [6,] "Linköping"  "Östergötland"
```

You can see that e.g. `NAME_1` defines regions, and several neighboring polygons share the same value for that data column, while `NAME_2` define the names of the municipality (more regional delimitations).

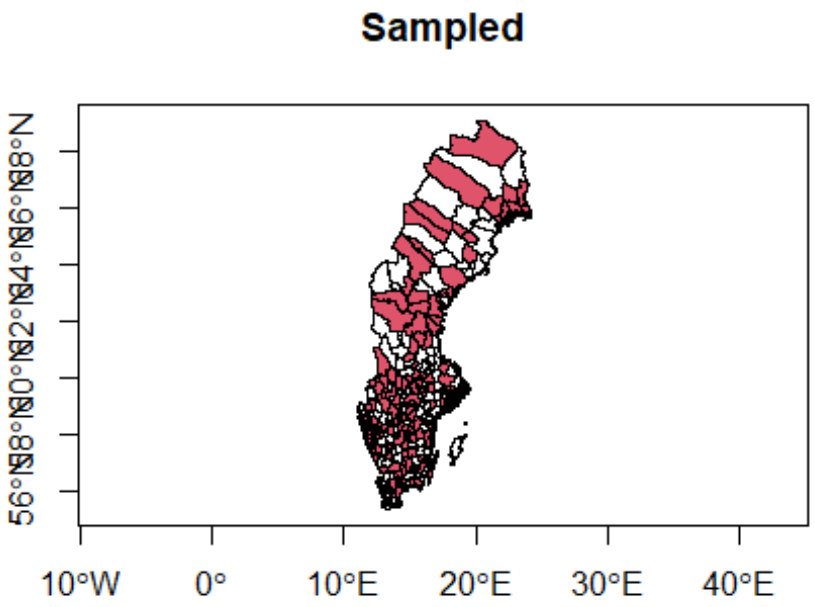
In some cases you may want to add a column of additional values to the polygon object, e.g. if you want to code which of these regions you have sampled (1 stands for sampled and

`0` stands for not sampled): In this case we will pretend that we have samples from 150 random regions

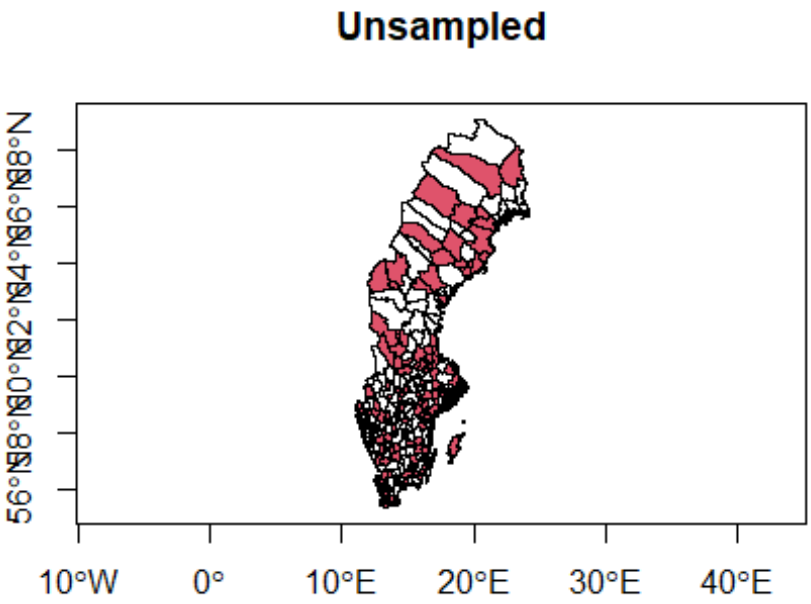
```
new_values =numeric(286)
new_values[sample.int(286,150)]=1
Sweden_2$sampled_regions = new_values
```

We might want to plot the regions with samples in a separate color, which we will try below

```
plot(Sweden_2,axes = T,  main="Sampled")
plot(Sweden_2[Sweden_2$sampled_regions==1,],col=2, add=T)
```



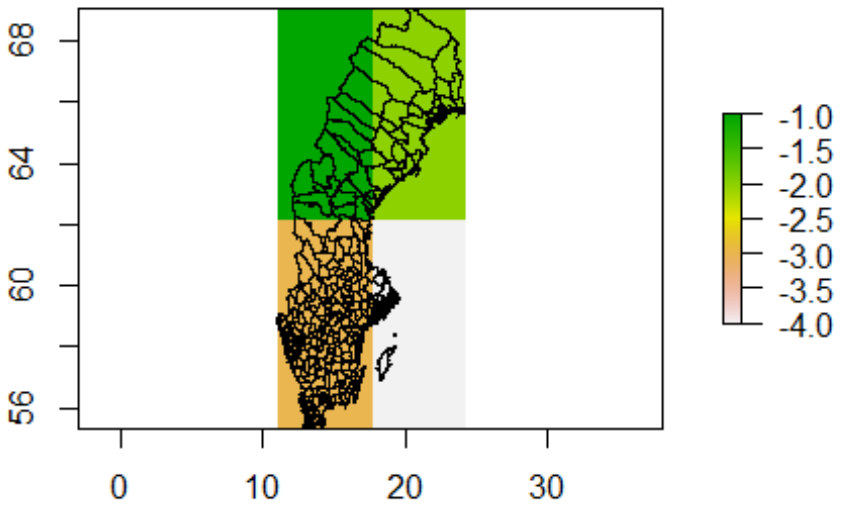
Task 3. Now let us selectively only plot those areas that have not yet been sampled in red. You can do so by modifying last line to plot regions fulfilling the criteria `Sweden_2$sampled_regions==0`. If you succeed, your figure could look something like this (since we selected random regions there will be differences), but you should also color the regions you did not color in the figure just above).



Basic modifications of rasters and polygons

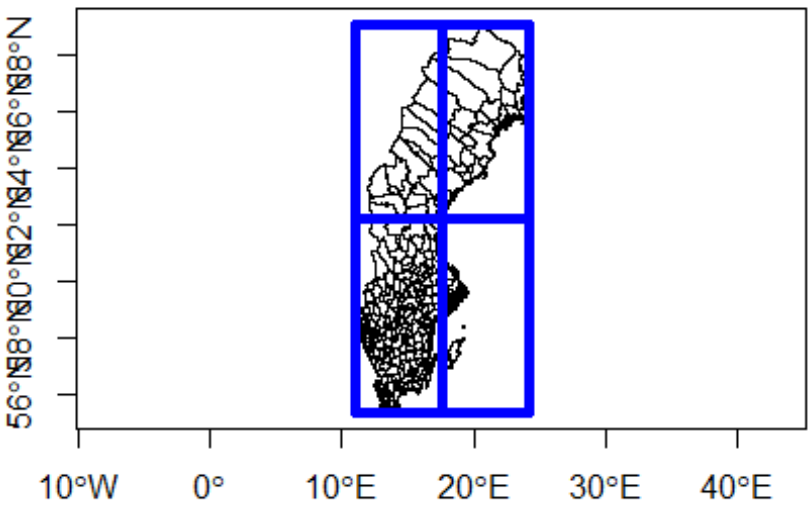
Let's say we want to divide our Sweden polygon into 4 equally sized cells. The easiest way to do this is to define a raster with 4 cells that encompasses the whole polygon. You can actually provide the raster function with a spatial polygon and it will automatically create a raster using the bounding box coordinates for the polygon:

```
library(raster)
z =- raster(Sweden_2, nrow=2, ncol=2, vals=1:4)
plot(z)
plot(Sweden_2,add=T)
```



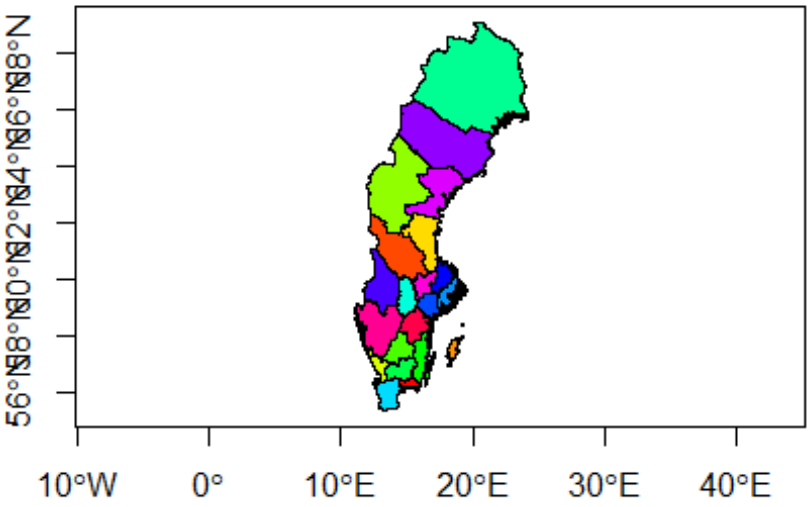
You can turn the grid of the raster into a spatial object itself, using the `as()` function, just as we did for the sf objects before. Let's do that and plot it on top of the Sweden polygon:

```
z_spatial = as(z, "SpatialPolygonsDataFrame")
plot(Sweden_2,axes = T)
plot(z_spatial, add=TRUE, border="blue", lwd=5)
```



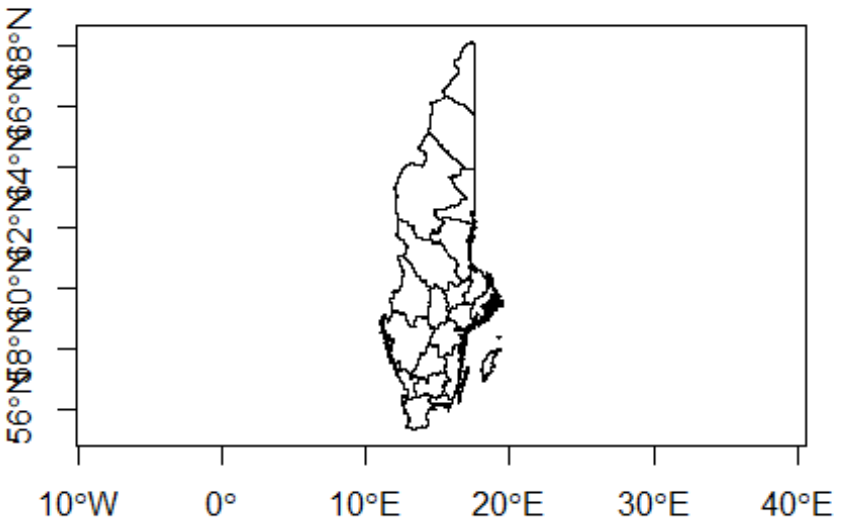
There are several useful functions when working with polygons, e.g. the `aggregate()` function, which can be used to join polygons based on their values. Let's use that function to join the municipalities by their values in the `Region` column (larger regions):

```
regions = aggregate(Sweden_2, by="NAME_1")
plot(regions, col=rainbow(21), border="black",axes = T)
```



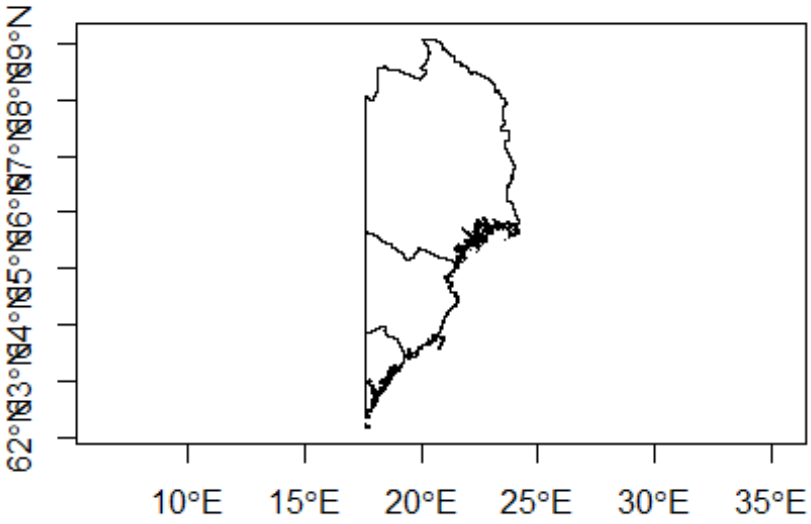
We can also determine the overlap of multiple spatial objects. E.g. we can remove the overlap between two polygons using the `erase()` functions. Here we remove the overlap between the map of Sweden and the second cell from our grid that we defined above:

```
library(raster)
e = erase(regions, z_spatial[2,])
plot(e,axes = T)
```



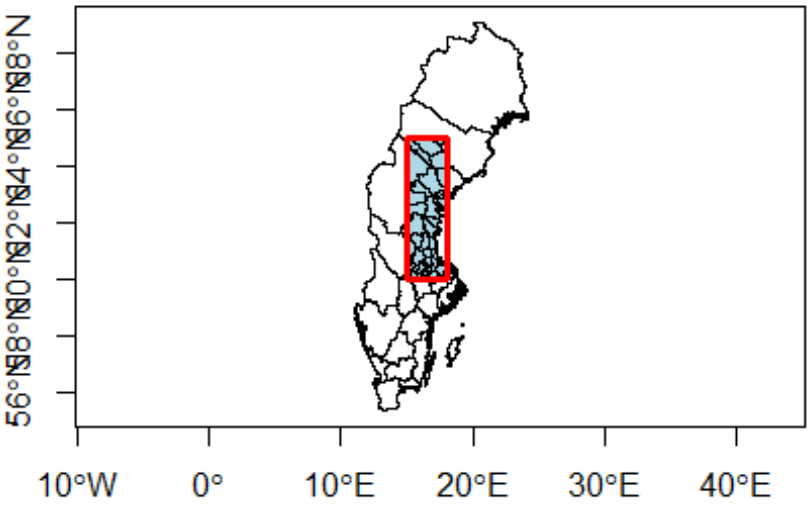
The `intersect()` function on the other hand does basically the opposite and keeps only what's shared between two polygons. Using this function we can e.g. extract the part of the map of Sweden that falls into our second cell of the made-up grid:

```
i = intersect(regions, 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 Sweden) 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). At the end we also plot the actual rectangle on top of everything (in red). In this case we plot municipalities within the rectangle but only the regions outside it.

```
library(raster)
ext = extent(15, 18, 60, 65)
pe = crop(Sweden_2, ext)
plot(regions,axes = T)
plot(pe, col='light blue', add=TRUE)
plot(ext, add=TRUE, lwd=3, col='red')
```



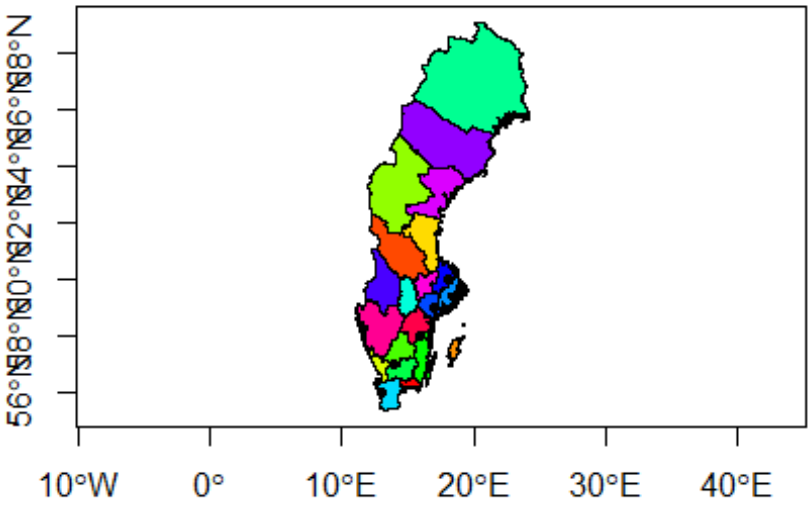
Spatial queries

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:

```
library(sp)
lon = c(13, 14, 16, 17, 18)
lat = c(56, 57, 58, 59, 60)
pts = cbind(lon,lat)
pts
```

```
##      lon lat
## [1,]  13  56
## [2,]  14  57
## [3,]  16  58
## [4,]  17  59
## [5,]  18  60
```

```
spts = SpatialPoints(pts)
plot(regions, col=rainbow(21), lwd=0.5,axes=T)
points(spts, pch=20)
```



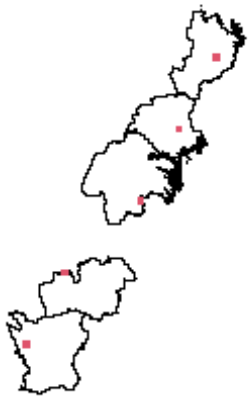
Now we can use the `over()` function in order to check if and where each point in our `SpatialPoints` object intersects with the `regions` polygon. The function will return a dataframe that shows for each point the values of the intersecting polygon dataframe. It will return the value `NA` for all data 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 Sweden shape object, which we can call by using the `crs()` function.

```
library(sp)
spts = SpatialPoints(pts, proj4string=crs(regions))
my_over_output = over(spts, regions)
my_over_output
```

```
##         NAME_1
## 1        Skåne
## 2    Kronoberg
## 3 Östergötland
## 4 Södermanland
## 5      Uppsala
```

Task 4 Plot only the regions that contain a point, based on the `over()` output, and add the points to see if it works. The final plot could 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:

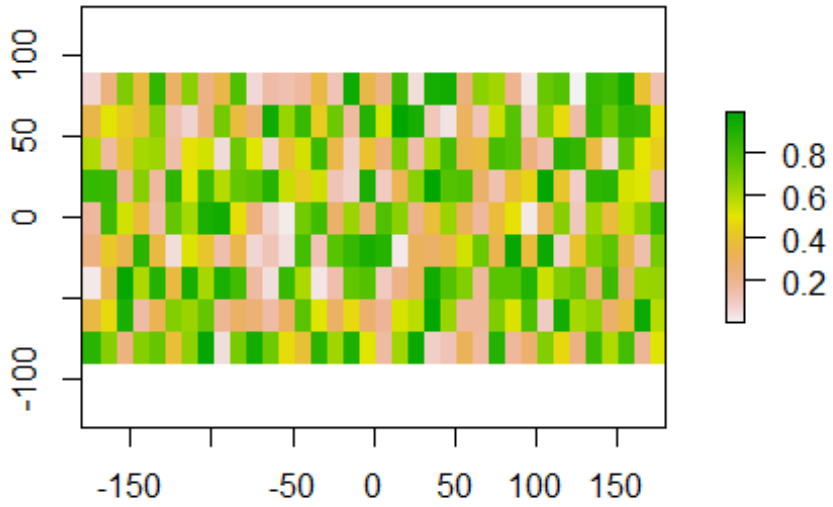
1. Identify the names of the regions with a point in. This can be seen with `my_over_output$NAME_1`
2. Identify which of the polygons in the regions fulfill the logical criterion that they contain a record. This is easiest done using the command %in%. To see how this works try writing c(1:10)%in%c(5:15)
3. Plot only the polygons fulfilling this criterion. This is done using the [] to subset the data.

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.

```
library(raster)
x = raster(ncol=36, nrow=9, xmn=-180, xm=180, ymn=-90, ymx=90)
```

```
values(x) = runif(ncell(x))
plot(x)
```

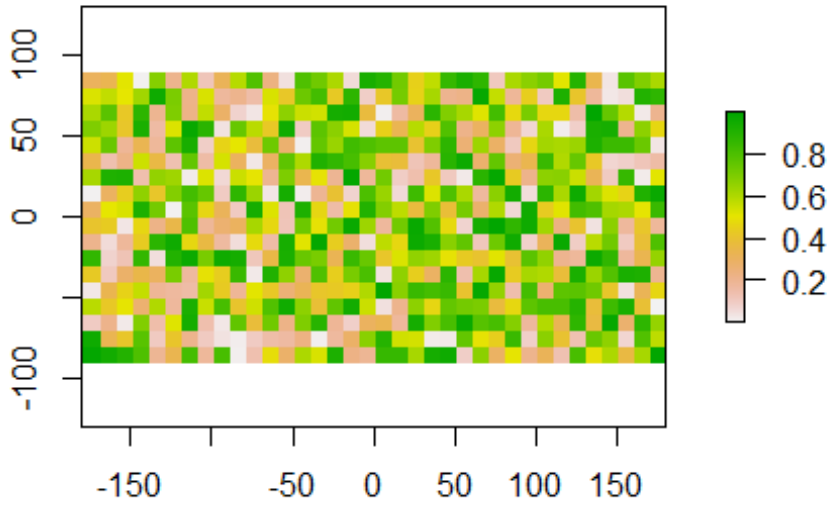


You can check the resolution of the raster (dimensions of each cell) using the `res()` command:

```
res(x)

## [1] 10 20
```

You will see that R allows rasters with different x and y extent. This is an example of R being a very forgivable program and it is generally bad practice to do so. Rasters should normally always be comprised of quadratic pixels and many programs will refuse to work on non-quadratic rasters. Let us create a raster fulfilling these criteria instead.

```
library(raster)
x = raster(ncol=36, nrow=18, xmn=-180, xm=180, ymn=-90, ymx=90)
values(x) = runif(ncell(x))
res(x)

## [1] 10 10

plot(x)
```



We can access the coordinate projection of the raster using the `projection()` function and can assign a value to it Wikipedia has a great article about map projections but we will also return to the topic later in the week.

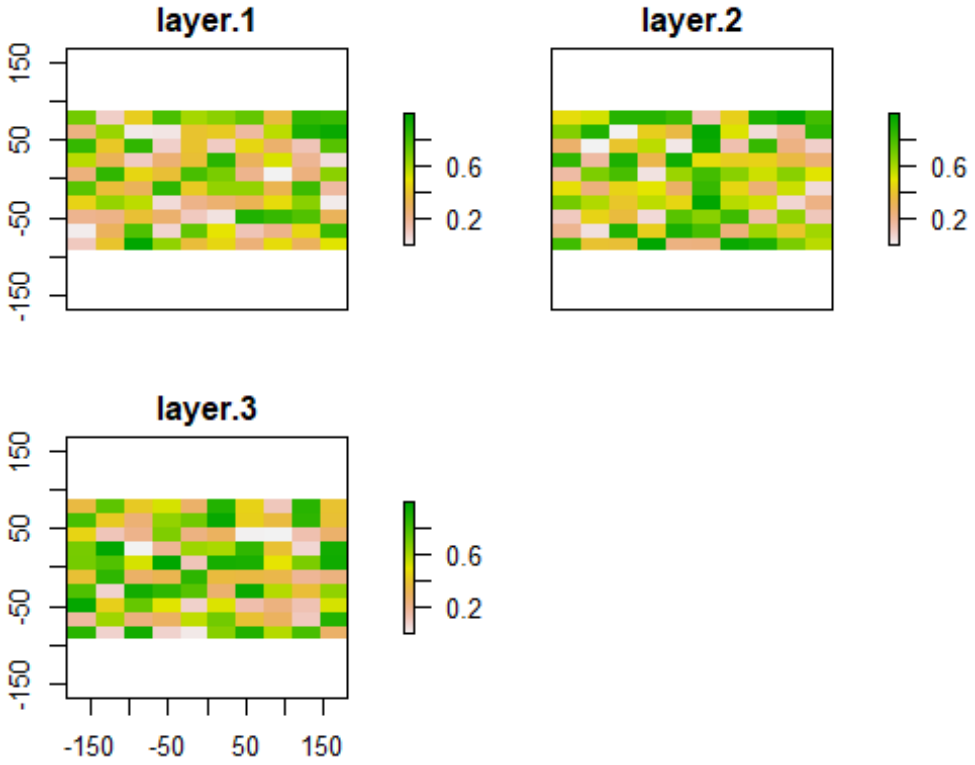
```
library(raster)
projection(x) = "+proj=utm +zone=48 +datum=WGS84"
projection(x)

## [1] "+proj=utm +zone=48 +datum=WGS84 +units=m +no_defs"
```

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:

```
library(raster)
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)
```



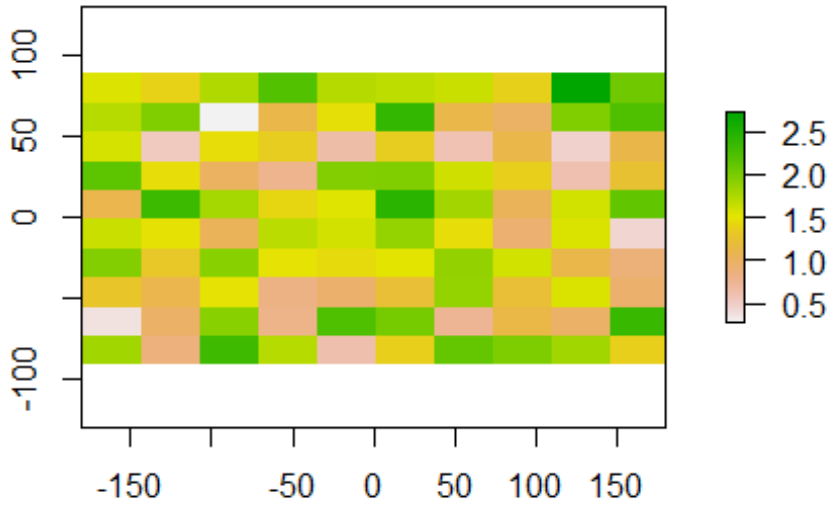
Algebraic operations with rasters

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)
# since we have created a stack:
```

```
sum_rasters = sum(s)
plot(sum_rasters)
```




```
product_rasters=r1*r2*r3
# or:
product_rasters=prod(s)
```

7. Occurrence count in raster

In this tutorial we will work with records of Carnivora where the record matches a specimen housed at a museum. I have already downloaded the data for you from GBIF. Tomorrow we will discuss how you can download records yourself from inside R. You should have downloaded this dataset if you downloaded all the course data from GitHub. First, we will load all records into R. We will use **read.table()** which is a general R function of vital use also for non-spatial stuff.

```
Carnivora_Data=read.table("C:\\Bpoxsync\\Spatial_R\\Data\\GBIF_Carnivora.txt"
)
```

We will start by looking at the structure of the data.

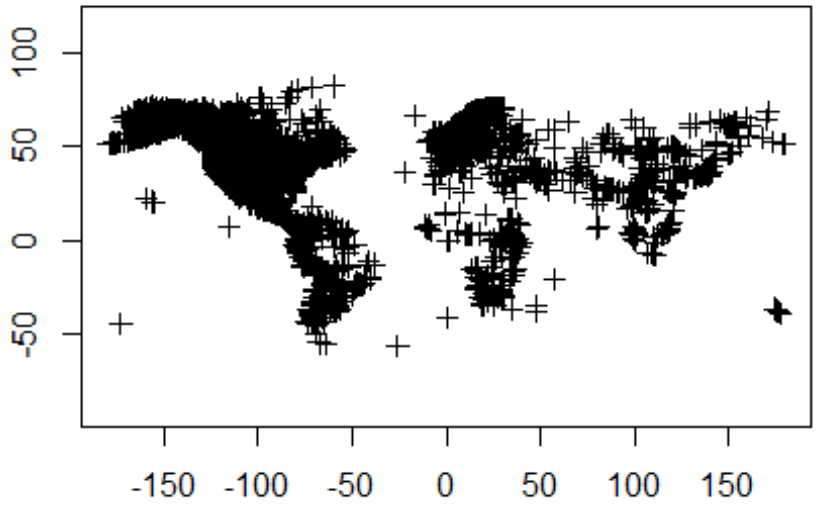
```
dim(Carnivora_Data)

## [1] 191078      5

summary(Carnivora_Data)
```

```
##     family             genus              species          decimalLatitude
##  Length:191078      Length:191078      Length:191078      Min.   :-89.00
##  Class :character   Class :character   Class :character   1st Qu.: 34.45
##  Mode  :character   Mode  :character   Mode  :character   Median : 46.95
##                                                           Mean   : 42.19
##                                                           3rd Qu.: 63.38
##                                                           Max.   : 90.00
##  decimalLongitude
##  Min.   :-180.00
##  1st Qu.:-147.55
##  Median :-107.18
##  Mean   : -85.90
##  3rd Qu.: -70.72
##  Max.   : 179.47
```

Now identify all records for species of group of species to focus on. This is done by specifying all records where either family, genus or species matches what you are looking for. In the command below, we e.g. identify the subset of records which belong to the weasel family (family Mustelidae)

```
Mustelidae=Carnivora_Data[Carnivora_Data$family=="Mustelidae",]
```

Task 5: Turn the coordinates of the records of whatever group you have selected into a `SpatialPoints` object, as we did earlier today. Store the resulting object under the name `species_occurrences`. You will need to have the `sp` package loaded for this (i.e. call `library(sp)`). Hint this is likely easiest done by starting with using **cbind()** to create a matrix with two columns with longitude and then latitude. After this you can transform them into a spatial object. Then plot the points. You can add the axes to your plot by adding `,axes=T` to your `plot()` command.



Loading a country map

I stored a file containing simple maps of all countries in the `data/ne_10m_admin_0_countries/` folder (available on the GitHub repo, which you already have downloaded).

Task 6: Read the shape data of all countries, as we did for the shape files in the first tutorial and store it under the name `all_countries`. Remember you need to load the `sf` package for this, if it's not already loaded.

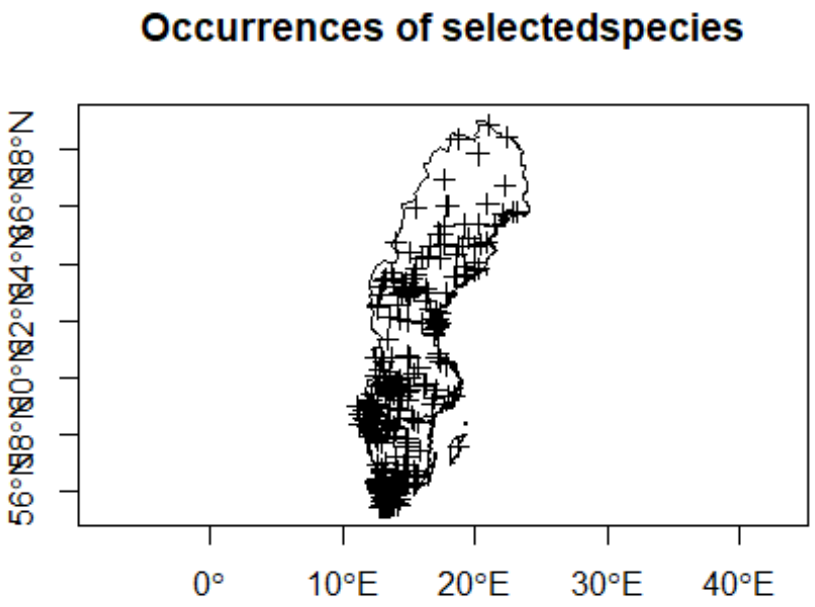
```
## Reading layer `ne_10m_admin_0_countries' from data source
`C:\Bpoxsync\Spatial_R\Data\ne_10m_admin_0_countries\ne_10m_admin_0_countries
.shp' using driver `ESRI Shapefile'
## Simple feature collection with 255 features and 94 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -180 ymin: -90 xmax: 180 ymax: 83.6341
## Geodetic CRS:  WGS 84
```

Extract the data for your country of choice from the `all_countries` object and convert the sf object into a spatial object using the `as(...,'Spatial')` function. The country names are stored in the `SOVEREIGNT` column of the dataframe. In the example below I first extract the index where the `SOVEREIGNT` column has the value `SWE` for Sweden. Look up here what the 3-letter country code for your country of choice is.

```
selected_country_index = which(all_countries$SOVEREIGNT == 'Sweden')
selected_country_sf = all_countries[selected_country_index,1]
selected_country = as(selected_country_sf, 'Spatial')
```

Now plot the data points on the map of your country. To do this we first need to do a little legwork. First, we will need to specify a CRS for our spatial points object (Here we use an "@" rather than "$" as you have seen multiple times before. The difference here is related to different underlying object types inside R. You do not need to understand this any deeper). After this we need to identify which of the records we had are within our country which we can do with **over()** as we used before. Finally we will specify that we only include the records within the country. **over()** give NA when the records is not seen in any polygon and we can therefore use the command **is.na()** which determines if a value is present to identify the records within the country :

```
library(sp)
species_occurrences@proj4string=selected_country@proj4string
Pres_Within=over(species_occurrences, selected_country)
plot(selected_country,main='Occurrences of selectedspecies',axes=T)
plot(species_occurrences[is.na(Pres_Within$featurecla)==F,],add=T)
```



Count occurrences per grid cell

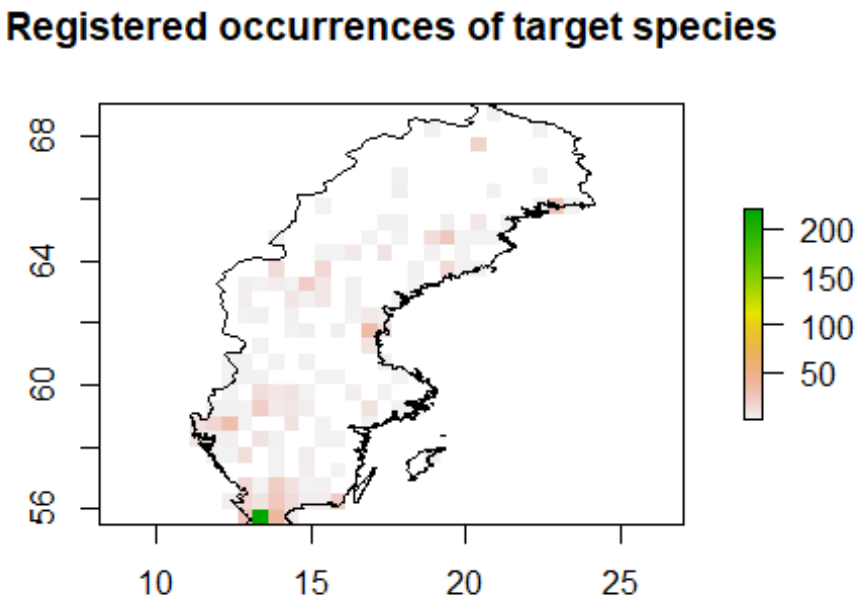
Now convert the country polygon into a raster and then count for each cell the number of occurrences of your species of choice. This will give you an idea where the species frequently is seen.

When using the `raster()` function you can set the resolution (in degrees in this case) to any value you like. The smaller the value, the smaller the cells and hence the more cells are being produced.

Then use the `rasterize()` function to rasterize the point data, which will count how many points fall into each cell. You can do this by calling the `rasterize()` function and providing the `SpatialPoints` object followed by the `raster` object. You also need to tell the function what needs to be done using the `fun=` settings, in our case you can specify `'count'`, which will count the number of points that fall within each cell.

```
library(raster)
raw_raster = raster(selected_country,resolution=0.5)
counts_per_cell =
rasterize(species_occurrences[is.na(Pres_Within$featurecla)==F,],raw_raster,f
un="count")
```

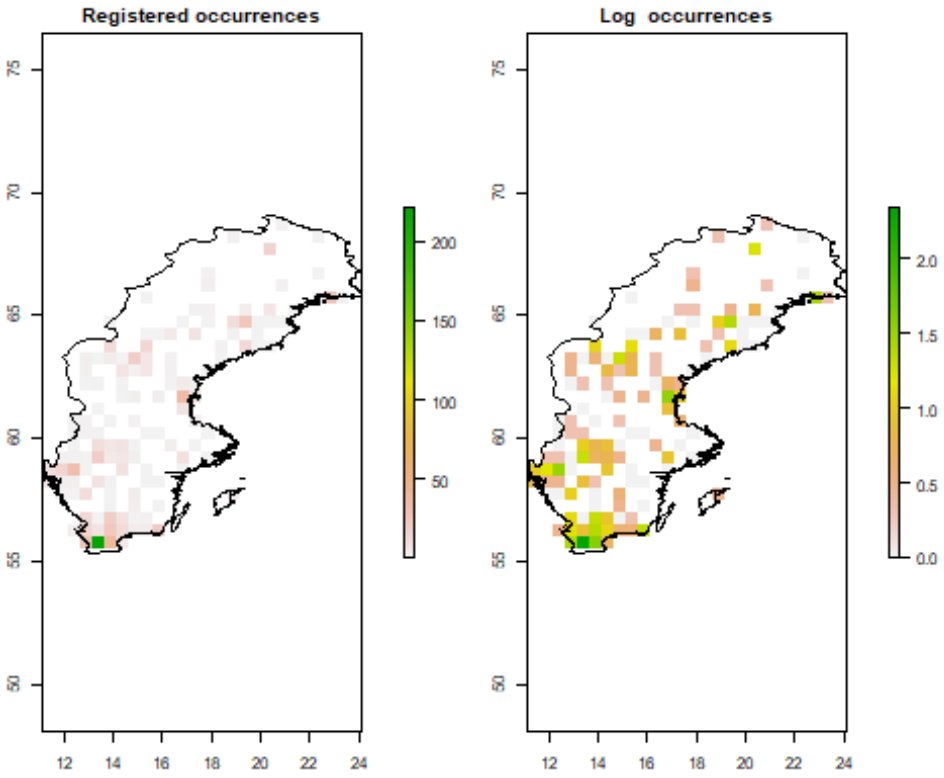
Task 7: Plot the resulting raster values of occurrence counts per cell and the polygon of your target country on top.



Sometimes the pattern might be difficult to see if the distribution of values is highly skewed and it might for plotting purposes be worthwhile to transform the data to plot the logarithm to the values. Notations for logarithms is confusing and vary between countries. In R when you want to use the base 10 logarithm you have to write **log10()**, whereas **log()** uses base e. To make these plots comparable we will use the command **par(mfrow=c(2,1))** which means that we split the plotting window into two columns. We will further add **asp=1** to the function which locks the aspect ratio between height and

width of the figure to be identical. This makes sense for Sweden but might need to be changed if you use another country. Finally we will reduce the size of the margins by calling **par(mar=c(2,2,2,2))** and reduce overall font size to ½ with **par(cex=0.5)** par() is a very useful function used to design overall design of the plot and there are several more things that can be changed inside if to make figures look nicer, but it takes a little to get used to.

```
par(mfrow=c(1,2))
par(cex=0.5)
par(mar=c(2,2,2,3))
plot(counts_per_cell, main="Registered occurrences", asp=1)
plot(selected_country,add=T)
par(cex=0.5)
plot(log10(counts_per_cell), main="Log  occurrences", asp=1)
plot(selected_country,add=T)
```



8) Plotting mammal diversity

In the exercise above we counted the number of occurrence points for each cell of a given raster. Now we are going to use the information of many individual presence/absence rasters (cells coded as 1 or 0) for all mammal species and add the values in each cell across all rasters in order to plot a map of global mammal diversity.

As input data we are going to use the "present natural" rasters, which was produced generated as part of the database Phylacine which is a project I have been heavy involved in since ~2013. For this project we estimated ranges of each mammal species under a

scenario of no human disturbance (i.e. where the species would occur today in a scenario where no human modifications of ranges has happened for the last 130,000 years).

The data is available via the Phylacine database Click on the link in the 'Download the data' section to start the download. Unzip the downloaded file and copy it into your data folder that you downloaded from the course GutHub repo. The raster data we're after is stored in the `Data/Ranges/Present_natural/` folder. You can see all files that are present in that folder using the `list.files()` command. We will also be using a metadata file called spatial metadata. You will see that it contains a lot of information which we will not go into here and we will only use its taxonomic skeleton:

```
head(list.files('C:\\Bpoxsync\\Phylacine\\1.21\\Data\\Ranges\\Present_natural
\\'))
```

```
## [1] "Abditomys_latidens.tif"   "Abeomelomys_sevia.tif"
## [3] "Abrawayaomys_ruschii.tif" "Abrocoma_bennettii.tif"
## [5] "Abrocoma_boliviensis.tif" "Abrocoma_budini.tif"
```

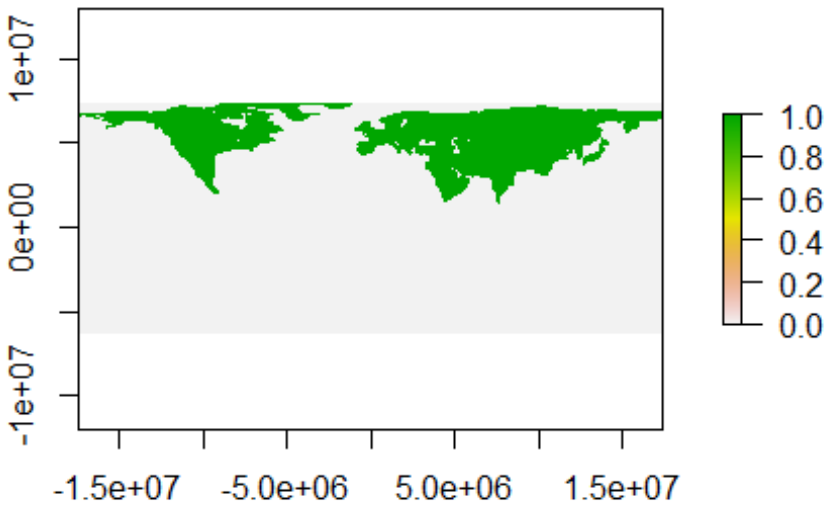
```
METADATA=read.csv("C:\\Bpoxsync\\Phylacine\\1.21\\Data\\Ranges\\Spatial_metad
ata.csv")
summary(METADATA)
```

```
##  Binomial.1.2        Order.1.2          Family.1.2         Genus.1.2
##  Length:5831        Length:5831        Length:5831        Length:5831
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##  Species.1.2        Certainty.Level    Modification        Motivation
##  Length:5831        Length:5831        Length:5831        Length:5831
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##
##
##
##  Last.Manual.Present.Natural.Modification Number.Cells.Current.Range
##  Length:5831                                Min.   :    0.0
##  Class :character                           1st Qu.:    6.0
##  Mode  :character                           Median :   39.0
##                                             Mean   :  341.4
##                                             3rd Qu.:  188.0
##                                             Max.   :37542.0
##  Number.Cells.Present.Natural.Range Change.In.Cells
##  Min.   :    0.0                    Min.   :-14047.00
##  1st Qu.:   11.0                    1st Qu.:     0.00
##  Median :   52.0                    Median :     0.00
##  Mean   :  379.1                    Mean   :    37.77
##  3rd Qu.:  226.5                    3rd Qu.:     0.00
##  Max.   :37542.0                    Max.   :  9239.00
```

```
head(METADATA)

##           Binomial.1.2 Order.1.2  Family.1.2    Genus.1.2 Species.1.2
## 1   Abditomys_latidens  Rodentia     Muridae    Abditomys    latidens
## 2    Abeomelomys_sevia  Rodentia     Muridae  Abeomelomys       sevia
## 3 Abrawayaomys_ruschii  Rodentia  Cricetidae Abrawayaomys     ruschii
## 4   Abrocoma_bennettii  Rodentia Abrocomidae     Abrocoma   bennettii
## 5 Abrocoma_boliviensis  Rodentia Abrocomidae     Abrocoma boliviensis
## 6      Abrocoma_budini  Rodentia Abrocomidae     Abrocoma      budini
##
Certainty.Level
## 1 3) Merger of likely human caused disjunct ranges by filling intervening
suitable habitats
## 2                                          0) Present Natural range
identical to IUCN range
## 3                                          0) Present Natural range
identical to IUCN range
## 4                                          0) Present Natural range
identical to IUCN range
## 5                                          0) Present Natural range
identical to IUCN range
## 6                                          0) Present Natural range
identical to IUCN range
##               Modification                                        Motivation
## 1 Connected via ecoregion. Disjunct range likely anthropogenic in cause.
## 2                     None                                              <NA>
## 3                     None                                              <NA>
## 4                     None                                              <NA>
## 5                     None                                              <NA>
## 6                     None                                              <NA>
##   Last.Manual.Present.Natural.Modification Number.Cells.Current.Range
## 1                              Version 1.0                          5
## 2                                     <NA>                         10
## 3                                     <NA>                         20
## 4                                     <NA>                         28
## 5                                     <NA>                          1
## 6                                     <NA>                          2
##   Number.Cells.Present.Natural.Range Change.In.Cells
## 1                                  9               4
## 2                                 10               0
## 3                                 20               0
## 4                                 28               0
## 5                                  1               0
## 6                                  2               0
```

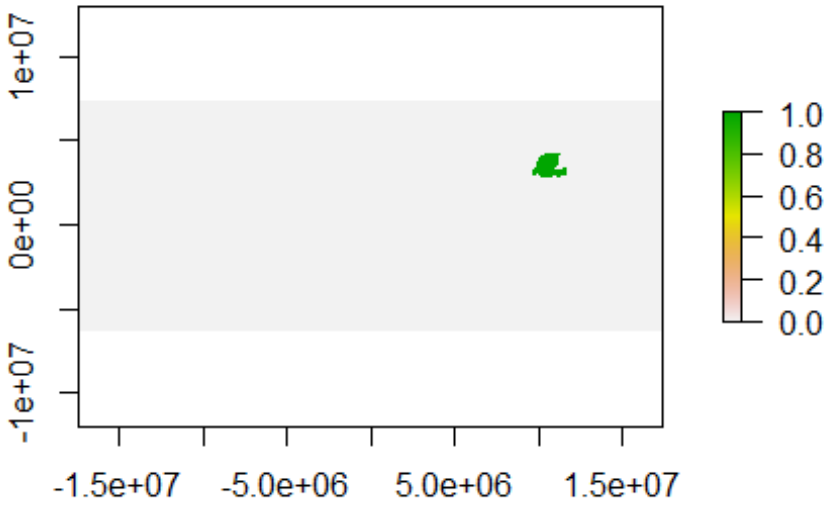
The folder contains a separate raster file for each species. To read a raster file we can just use the `raster()` function and the file name. We could e.g. want to read it the raster for wolf (canis_lupus), where the file would be called "canis_lupus.tif" In this and the other rasters in the folder, presense is coded as 1 and absense is coded as 0

```
library(raster)
Wolf =
raster("C:\\Bpoxsync\\Phylacine\\1.21\\Data\\Ranges\\Present_natural\\canis_lupus.tif")
plot(Wolf)
```



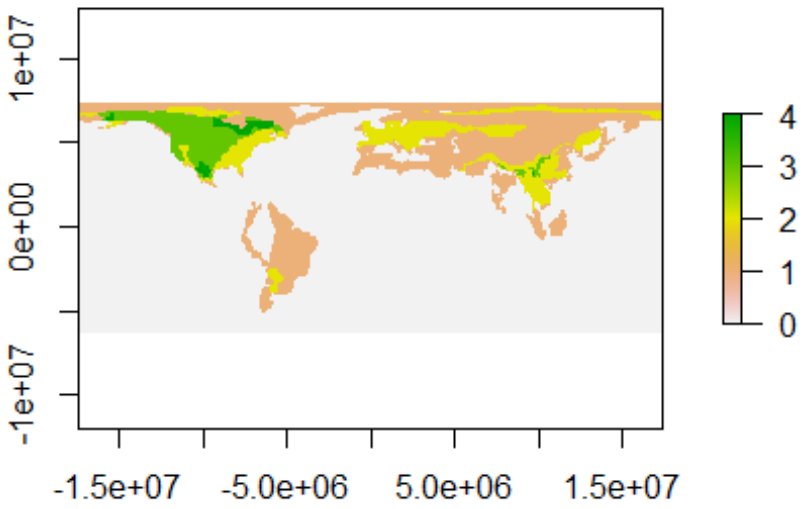
We will here instead create rasters for a number of species. In this code we will create a diversity map for all species of bears (family Ursidae) but you can change it to any other group you like. As part of this we will rely on the command paste0(). This is a command which is used to write a specific string of text and is often useful in loops involving reading and writing to files. Let us first show how to read and plot the first species. We will be relying on the species name stored as Binomial.1.2 and the description of the group which in this case is stored as Family.1.2 but depending on the group you focus on it could be relevant to rely on Order.1.2 or Genus.1.2 instead.

```
library(raster)
Selected_species=METADATA$Binomial.1.2[METADATA$Family.1.2=="Ursidae"]
filepath ="C:\\Bpoxsync\\Phylacine\\1.21\\Data\\Ranges\\Present_natural\\"
Raster1=raster(paste0(filepath, Selected_species[1], ".tif"))
plot(Raster1)
```



This gave us the range of the first species (which is this case is the giant panda). Now let us instead read in all the species fulfilling our criteria. For this we can simply iterate through all raster files (using a `for(){}` loop, see here for a quick tutorial on for-loops in R). We initially create a list() to store the data but afterwards we will transform the data into a rasterstack. Finally, we calculate the diversity raster as the sum of all the rasters in the stack

```
library(raster)
RASTERS=list()
for (i in 1:length(Selected_species)) {
    RASTERS[i]=raster(paste0(filepath, Selected_species[i], ".tif"))
}
RASTERS = stack(RASTERS)
Diversity_raster=sum(RASTERS)
plot(Diversity_raster)
```

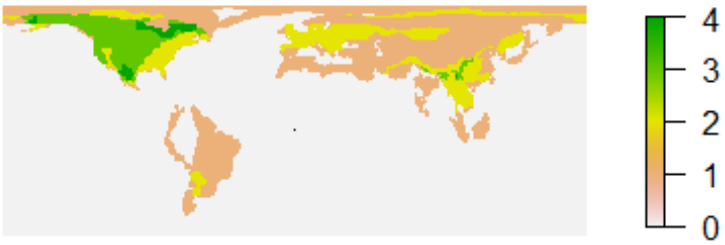


You can already see that diversity of any group you have chosen is distributed unevenly across the globe. However, we have not yet delimited the geographic features, the only reason why you can see where the continents are placed is because the structure of the diversity data. Now we want to delimit land from water, by plotting a world-map polygon on top of this raster.

Load the world map stored at `global\\ne_50m_land` and turn it into a spatial object.

```
library(sf)
world =
st_read("C:\\Bpoxsync\\Spatial_R\\Data\\global\\ne_50m_land\\ne_50m_land.shp"
)

## Reading layer `ne_50m_land' from data source
`C:\Bpoxsync\Spatial_R\Data\global\ne_50m_land\ne_50m_land.shp' using driver
`ESRI Shapefile'
## Simple feature collection with 1420 features and 3 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -180 ymin: -89.99893 xmax: 180 ymax: 83.59961
## Geodetic CRS:  WGS 84

world_spatial = as(world, "Spatial")
```

Task 8: Now plot the world polygon on top of the raster data (hint: it's supposed to not work out properly, see next section).



Transforming coordinate system

What went wrong here? Do you see the little black dot in the center of the plot? That's our world polygon! It looks like the coordinate system of our raster data does not match with that of the world map polygon (different coordinate reference systems). Let's check the projections of the two objects:

```
library(raster)
projection(Diversity_raster)

## [1] "+proj=cea +lat_ts=30 +lon_0=0 +x_0=0 +y_0=0 +datum=WGS84 +units=m
+no_defs"

projection(world_spatial)

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

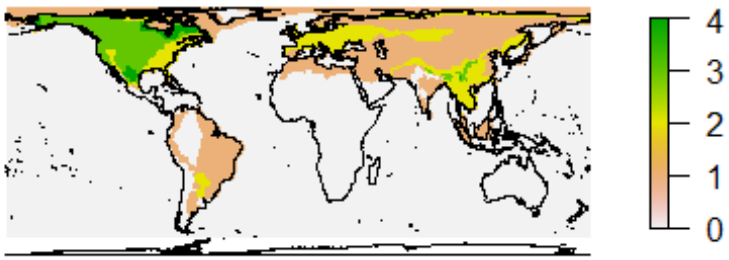
They are indeed different! You don't need to understand the different projections in detail (we will cover this in part on Thursday) but all you need to know is how to convert one projection into another. There are different tools to do this, some of the commonly used ones are the `spTransform()` function of the `rgdal` package or the `ptransform()` function of the `proj4` package, but unfortunately those two packages can be complicated to install on some systems, so in this tutorial we stick to the `st_transform()` function which is part of the `sf` package, which we already loaded above. The `st_transform()` function can be used to transform an `sf` object (output of `st_read()` function). This is why we are using the `world` object and not the `SpatialPolygon` object (`world_spatial`) in the following

command. All we need to provide is the target projection, so in this case `projection(merged_rasters)`:

```
library(sf)
transformed_world = st_transform(world,projection(Diversity_raster))
transformed_world_spatial = as(transformed_world, "Spatial")
```

Let's see what the plot looks like now:

```
plot(Diversity_raster,axes=F,box=FALSE)
plot(transformed_world_spatial,add=T)
```



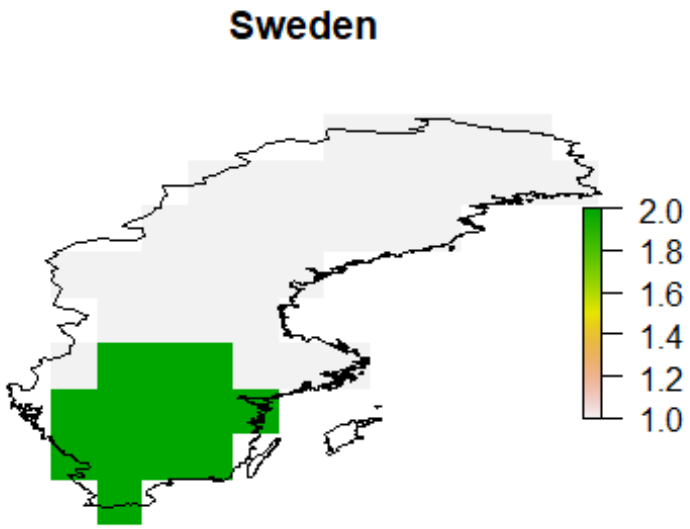
Using mask function to make cell selection

What if you only were interested in the diversity within a specific region like your selected country? In that case you can remove any part of a raster outside a desired polygon using the `mask()` function, which only keeps those cells that are within a given polygon like your selected country. This function is useful for many different analytical tasks and we will also return to it later in the course.

Note that we need to transform the selected country object first so it matches the projection of the raster. You should already know how this is done so I am not providing the codes. **In my future codes I will call the transformed country country_transformed_spatial , you can use another name but that requires you to change the code later when I am referring to country_transformed_spatial**

We can now create a raster only containing the Swedish part of the raster using mask(), and after this plot the results.

```
library(raster)
country_cells = mask(Diversity_raster,country_transformed_spatial)

plot(country_transformed_spatial,main=country)
plot(country_cells,add=T)
plot(country_transformed_spatial,add=T)
```



In this particular case you can see that the present-natural diversify in southern Sweden would be two species (which would be the brown bear and the extinct cave bear), while the natural diversity of the rest of Sweden would be a single species. Your maps will of course be different depending on the organism group and country you are plotting.

You can store the plot as a pdf file to look at it in some more detail: As default it will create the folder in your default working directory. But you can change it to wherever you want to story it on your computer. In R this works by first creating a pdf with the command **pdf()**. After this all files are plotted to this file rather than the plotting window. When you are done plotting you write the link **dev.off()** which finishes the pdf file.

```
pdf("Diversity.pdf")
plot(country_transformed_spatial,main=country)
plot(country_cells,add=T)
plot(country_transformed_spatial,add=T)
dev.off()
```

```
## png
##   2
```

Transform matrix of coordinates into desired projection

What if we want to transform a matrix of coordinates that we have loaded into R into any given projection? For example, let's try to plot the `species_occurrences` data from earlier on the transformed map from the previous section.

You will already have created a *SpatialPoints* object of records of a group of carnivores but since it was as a task I do not know what you called the object so we will create a new one here.

```
Mustelidae=Carnivora_Data[Carnivora_Data$family=="Mustelidae",]
locations = cbind(Mustelidae$decimalLongitude, Mustelidae$decimalLatitude)
species_occurrences= SpatialPoints(locations)
```

We first need to tell R what projection our coordinates are currently in. You might find this information stored in the metadata that came with your occurrence files or if you collected the data yourself you may be able to find the information in the settings of your GPS device. However, if you have no idea what the projection of your coordinates is, look at the values of your coordinates and try to make out what reference system they might stem from. Often a good first guess is `'+proj=longlat +datum=WGS84'`.

In our case the data indeed is in `'+proj=longlat +datum=WGS84'`, so we can use the `CRS()` command from the `sp` package to turn this string into a coordinate reference system object. The `class()` command shows you what object class a given object belongs to.

```
library(sp)
crdref = CRS('+proj=longlat +datum=WGS84')
class(crdref)

## [1] "CRS"
## attr(,"package")
## [1] "sp"
```

Now we use this information when transforming our coordinates into a `SpatialPoints` object, by setting the argument `proj4string=` to `crdref`, which we defined in the previous step. This will make sure that the information about the coordinate projection is stored with our spatial data.

```
library(sp)
species_occurrences = SpatialPoints(locations,proj4string=crdref)
```

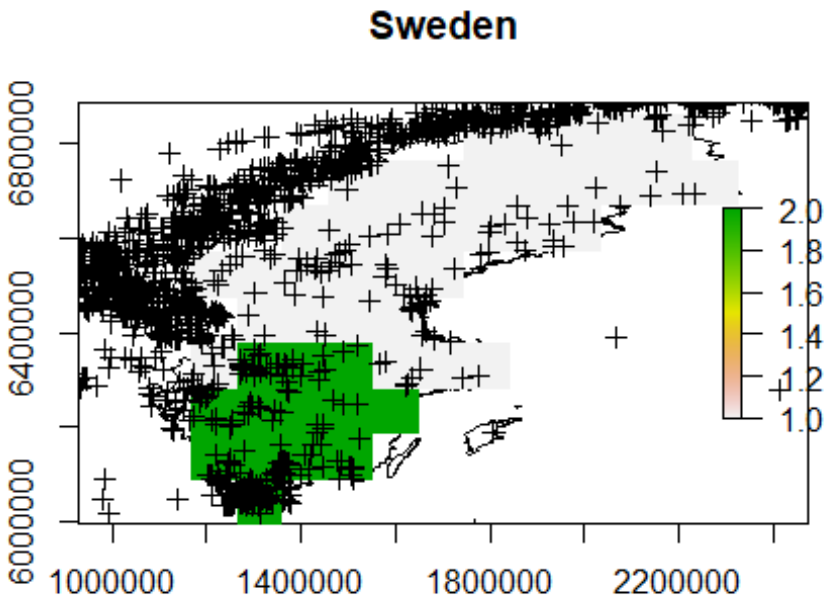
Now where your `SpatialPoints` object has CRS information, you can project it into any desired different projection. In this case this is the projection of our rasters from earlier (`projection(merged_rasters)`). We can again use the `st_transform()` function, just as we did earlier for the world map polygon. However before we can use that function we need to convert our `SpatialPoints` object into a `sf` object (the function `st_transform()` only

works for `sf` objects). We can simply do that using the `st_as_sf()` function, as shown below:

```
library(raster)
library(sf)
target_projection = projection(Diversity_raster)

species_occurrences_sf = st_as_sf(species_occurrences)
species_occurrences_transformed =
st_transform(species_occurrences_sf,target_projection)
```

Now let's plot the data and see if it matches with our map.

```
plot(country_transformed_spatial,axes=T,main="Sweden")
plot(country_cells,add=T)
plot(species_occurrences_transformed,pch=3,add=T)
```



You have now finished the tutorial for day one. You can spend the rest of the day trying to create the nicest possible graph for the group and country you are working on and show/ brag on slack.