R for spatial data analysis and mapping

Alan R. Pearse, Natural Resources Society

24 July 2018

Preamble

Welcome to the R for spatial data analysis and mapping workshop, run by the QUT Natural Resources Society. In this workshop, we will be learning:

- How to import and export spatial data
- How to deal with different spatial data formats in R
- How to perform basic GIS operations in R
- How to analyse and map spatial data, with a case study in soil concentrations of lead.

Loading packages needed for the workshop

In this section, you will find the code needed to load some popular spatial packages in R. We will need these packages throughout the workshop. Run the following commands to load the packages. Some of the packages will display special messages upon loading.

```r
library(sp) # Ubiquitous package for representing spatial data
library(rgdal) # For reading in shapefile data
```

```
## rgdal: version: 1.3-3, (SVN revision 759)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.2.3, released 2017/11/20
##  Path to GDAL shared files: C:/MySoftware/library/rgdal/gdal
##  GDAL binary built with GEOS: TRUE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: C:/MySoftware/library/rgdal/proj
##  Linking to sp version: 1.3-1
```

```r
library(raster) # Comes with raster functionality
library(rgeos) # More advanced spatial data analysis functionality
```

```
## rgeos version: 0.3-28, (SVN revision 572)
##  GEOS runtime version: 3.6.1-CAPI-1.10.1 r0
##  Linking to sp version: 1.3-1
##  Polygon checking: TRUE
```

```r
library(gstat) # For geostatistical modelling -- more on that later.
```

Types of spatial data

In this section, you will be introduced to two common spatial data formats and their representations in R. Of course, we are talking about

- shapefiles; and
- rasters.

Before the workshop, you should all have been sent a zip folder containing three spatial data layers.

1. meuse_pts.shp (and associated files)

2. meuse_riv.shp (and associated files)
3. meuse_grd.tif (and associated files)

The first task of the workshop is to load these data into R.

We'll start with the shapefile data. The use the `readOGR` command from the package `rgdal` to import shapefiles. The code block below is how the command should be set out.

```
meuse_pts <- readOGR(
  dsn = "meuse_pts.shp", # The file path, with extension,
  layer = "meuse_pts" # the file name
)
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "D:\Dropbox\NRS\meuse_pts.shp", layer: "meuse_pts"
## with 155 features
## It has 14 fields
```

We should do the same thing with the meuse_riv shapefile.

```
meuse_riv <- readOGR(
  dsn = "meuse_riv.shp",
  layer = "meuse_riv"
)
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "D:\Dropbox\NRS\meuse_riv.shp", layer: "meuse_riv"
## with 1 features
## It has 1 fields
```

Now we deal with the raster data. There are several ways to import raster data but we will use the `raster` command from the `raster` package. The code block below shows how to set out the command.

```
meuse_grd <- raster("meuse_grd.tif")
```

Understanding spatial data classes

R is an object-oriented programming language. This means that all data has a certain representation, and there are some well-defined operations that can be done on each data type. In this section, we investigate the classes of spatial data objects.

Shapefiles are typically represented as a Spatial* object. The Spatial* classes are R classes which allow for the storage and visualisation of datasets with a set of coordinates and a set of data or other information associated with each coordinate.

The code block below can be used to query the classes of different objects in your R environment.

```
class(meuse_pts)
```

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

```
class(meuse_riv)
```

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

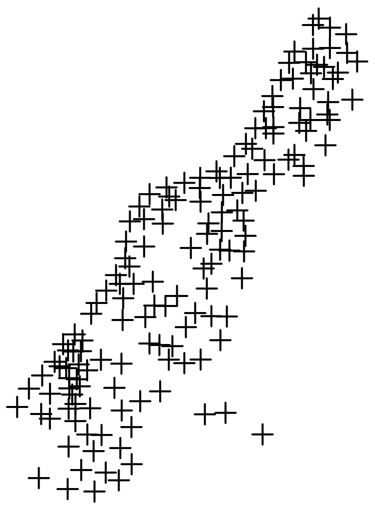
```r
class(meuse_grd)
```

```
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
```

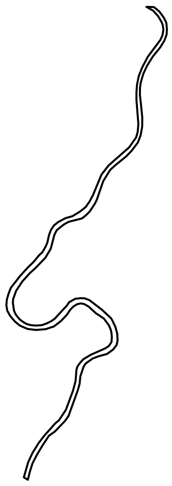
As previously mentioned, each class has a set of well-defined operations which can be performed on it. One such operation for spatial data is a `plot()` operation, which draws the object. The same function can be used to get different results from the different objects since they come from different classes.

Consider the plotting of the Spatial* objects. Note that `meuse_pts` is a SpatialPointsDataFrame, which means the coordinates represent points in space and the data stored in the object are for those points in space. By way of contrast, `meuse_riv` is a SpatialPolygonsDataFrame; the coordinates in this object represent the boundary of an area, and the data associated with the coordinates are for the conditions inside that area. Plotting them produces different results.

```r
plot(meuse_pts)
```

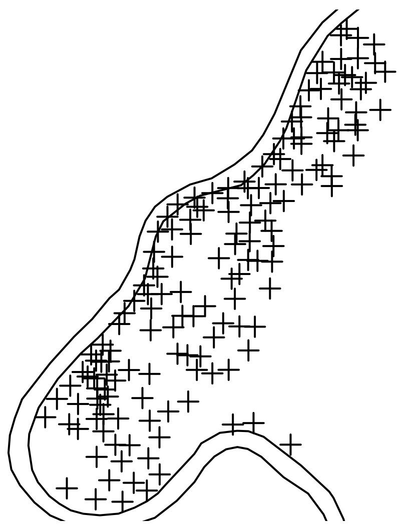


```r
plot(meuse_riv)
```



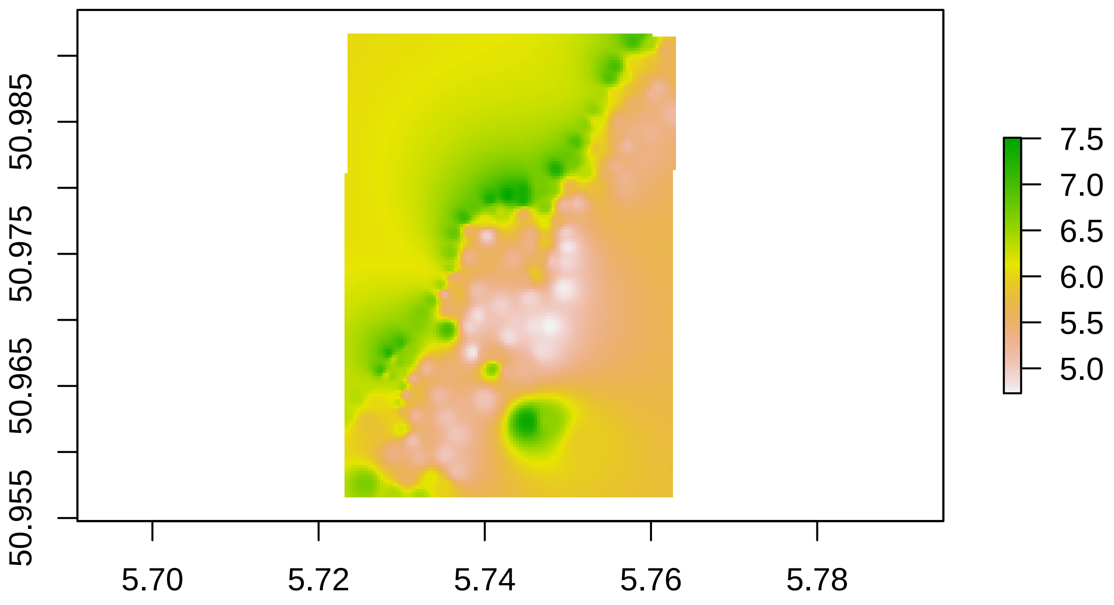
Due to the definition of the plot command, we can even plot these together.

```r
plot(meuse_pts)
plot(meuse_riv, add = TRUE)
```



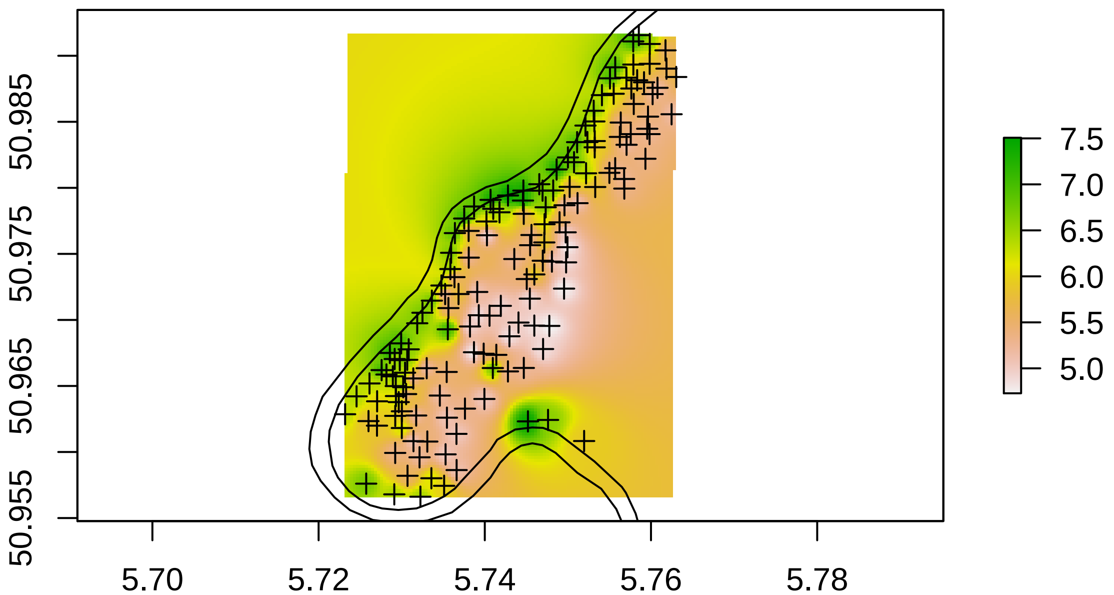
The behaviour of the plot command is radically different for `meuse_grd`, which is an object of class `RasterLayer`. The output from `plot()` called on `meuse_grd` is an image, with each cell coloured by `log(zinc)` concentrations in the area around the Meuse river.

```
plot(meuse_grd)
```



Again, we can plot these layers on top of each other.

```
plot(meuse_grd)
plot(meuse_pts, add = TRUE)
plot(meuse_riv, add = TRUE)
```



One more point to consider is that we can convert R objects between classes. This is called 'coercion'. For example, we can convert between an object of class `RasterLayer` and an object of class `SpatialPixelsDataFrame`. The code block below demonstrates this:

```
meuse_spx <- as(
  meuse_grd, # Object to coerce
  "SpatialPixelsDataFrame" # Target of coercion
)
```

We can then further reduce this to an object of class `data.frame`. This is not usually done but can be useful for some mapping applications.

```
meuse_gdf <- as.data.frame(
  meuse_spx
)

head(meuse_gdf)
```

```
##   meuse_grd        x        y
## 1  6.043002 5.723679 50.99156
## 2  6.044538 5.724035 50.99156
## 3  6.046091 5.724391 50.99156
## 4  6.047660 5.724747 50.99156
## 5  6.049244 5.725103 50.99156
## 6  6.050843 5.725459 50.99156
```

Some basic operations on spatial data

In this section, we will introduce some of the common operations that people perform on spatial data. These are

- Projecting coordinates
- Masking rasters
- Writing spatial data to file

Projections

Projections are the bread and butter of spatial data operations. Projections are used to transmit spatial data from one coordinate system to another. There are two broad types of coordinate system (also called coordinate reference systems, CRS):

1. Projected (or planar)
2. Geographic

The main difference is that geographic coordinates are expressed in angular units (longitude, latitude) that locate you on the globe; projected coordinates are expressed in linear units (e.g. m, km) and are basically the result of a curved earth mapped on to a flat piece of paper.

The data we currently have is in geographic coordinates. Specifically, it is in a coordinate system called EPSG (European Petroleum Survey Group) 4326, which is based on the global WGS84 ellipsoid and datum. We can query this for our Spatial* layers by asking for the `proj4string`, which is a common standard for representing projection information.

```
proj4string(meuse_pts)
```

```
## [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

For rasters, we use the command `crs` instead. Note however that the projection information is still given as a `proj4string`; it's just the way of accessing this information that is different.

```
crs(meuse_grd)
```

```
## CRS arguments:
##  +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

Sometimes it is important to project spatial data into or out of geographic coordinates. This is particularly true when doing distance calculations, which are much more complicated in elliptical space. To project Spatial* objects, we use `spTransform` from the package `sp`. The code block below demonstrates this process.

```
# Before projection
head(meuse_pts@coords)
```

```
##      coords.x1 coords.x2
## [1,]  5.758536  50.99156
## [2,]  5.757863  50.99109
## [3,]  5.759855  50.99089
## [4,]  5.761746  50.99041
## [5,]  5.761863  50.98903
## [6,]  5.763040  50.98839
```

```
# Projecting
meuse_pts_proj <- spTransform(
  meuse_pts,
  CRS("+init=epsg:28992") # Some Norwegian system
)
```

```
# After projection
head(meuse_pts_proj@coords)
```

```
##      coords.x1 coords.x2
## [1,]    181072    333611
## [2,]    181025    333558
## [3,]    181165    333537
## [4,]    181298    333484
## [5,]    181307    333330
## [6,]    181390    333260
```

A different function must be used for raster data, since its gridded nature prevents piecewise projection. We have to use `projectRaster` from the package `raster`. The code block below demonstrates the use of this function.

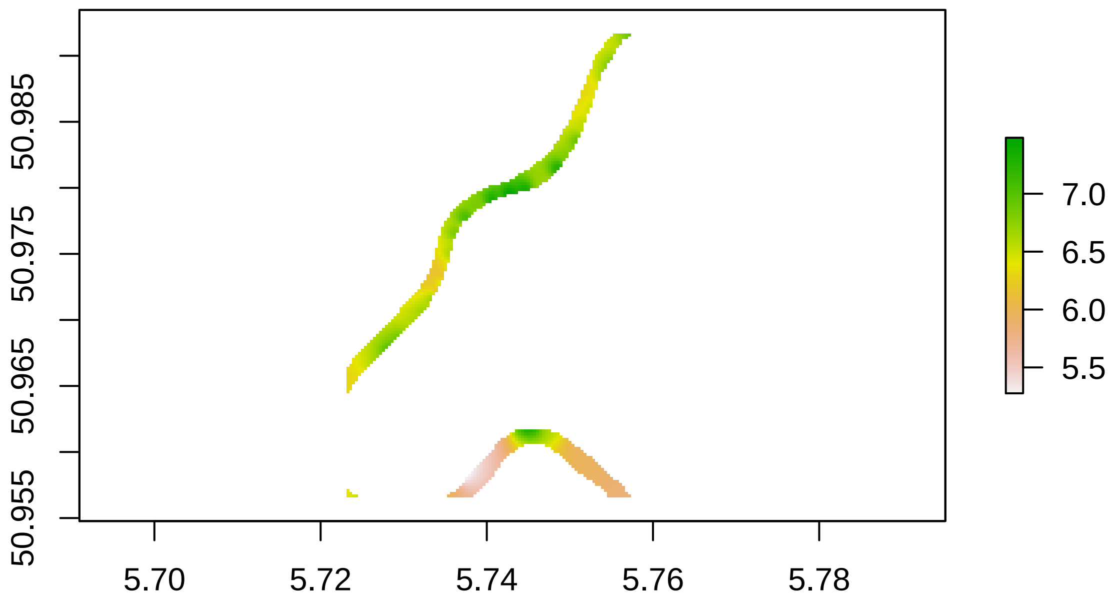
```
meuse_grd_proj <- projectRaster(
  meuse_grd,
  crs = CRS("+init=epsg:28992")
)
```

Masking rasters

Masking a raster is the process of removing the cells in a raster which lie within or outside of a certain area, which is usually defined by a polygon. In R, this is a simple process relying on the `mask` function from the `raster` package. We demonstrate below.

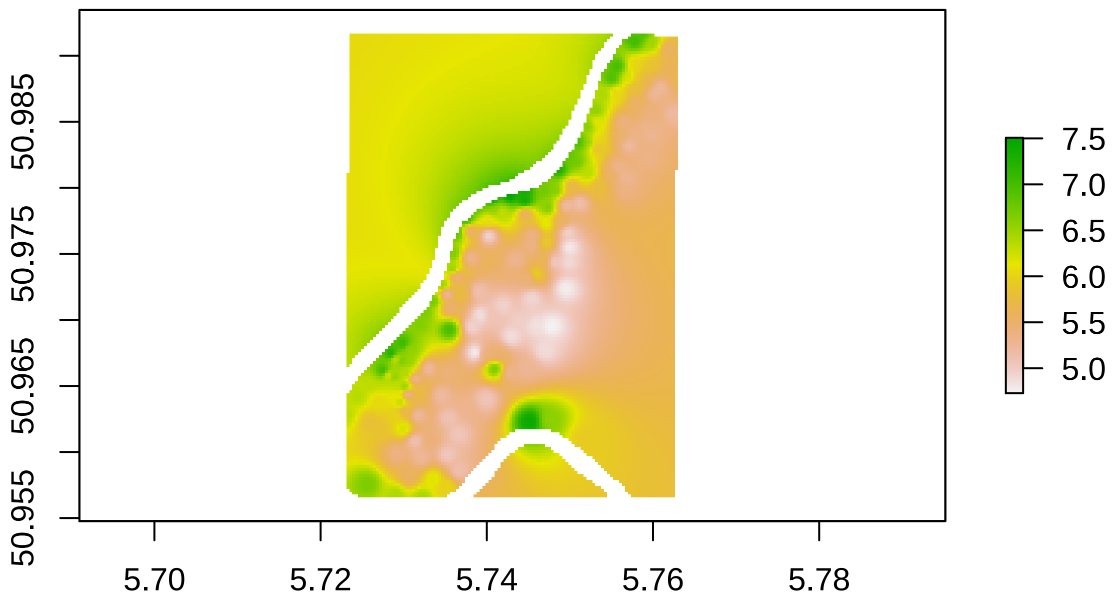
```
meuse_grd_mask <- mask(
  meuse_grd, # The raster to mask
  meuse_riv  # The mask
)

plot(meuse_grd_mask)
```



```
meuse_grd_mask_inverse <- mask(
  meuse_grd,
  meuse_riv,
  inverse = TRUE
)

plot(meuse_grd_mask_inverse)
```



Writing files to disk

Data that you have stored in your R environment can be written to your computer as files. If you want to produce a shapefile, use `writeOGR` from the `rgdal` package. If you want a raster, use `writeRaster` from the `raster` package.

```r
# For shapefiles
writeOGR(
  obj = meuse_pts_proj, # The object you want to write to file
  dsn = "meuse_prj.shp", # File path, with extension
  layer = "meuse_prj", # The file name, no extension
  driver = "ESRI Shapefile" # The file encoding
)

# For rasters
writeRaster(
  x = meuse_grd_mask_inverse, # The object to write
  filename = "meuse_mgrd.tif" # The file path, with extension
)
```

Mapping lead concentrations in soil

Now we arrive at our major ‘case study’. The data we have been working with is actually a soil science study done in the floodplain of the river Meuse, in the Netherlands. The study looked at concentrations of heavy metals deposited by the river in flood, due to the region’s history of intense mining.

The `meuse_pts` dataset are the locations where the scientists measured concentrations of heavy metals in the soil, and the associated data are the concentrations they found. But note that these are just point observations. For management purposes, it is often more useful to have a continuous surface, like in `meuse_grd`, to understand the overall distribution of soil contaminants. In this section, we will go through a basic process for generating and mapping such a surface.

An outline of the methodology

Here, we outline the workflow for this problem in spatial prediction. To make predictions in space, we have to

1. Make sure the data we use for prediction is in linear units (important for the distance-based calculations we have to make).
2. Define a set of locations at which we want to predict the soil concentration of lead.
3. Predict the value of lead concentration at these new locations.
4. Make sure our predictions make physical sense.

Projecting our data to linear units

The data that we have are the observed points in the `meuse_pts` dataset. We have already projected these points to linear units but we'll do it again here.

```r
meuse_pts_proj <- spTransform(
  meuse_pts,
  CRS("+init=epsg:28992")
)
```

Defining a prediction grid

We have to define a grid of prediction points in our study area. This involves first finding the extent of our study area, which we do thus:

```r
meuse_ext <- extent(meuse_pts_proj)
```

Then using the raster package we can create a grid from scratch.

```r
meuse_grd <- raster(
  ext = meuse_ext, # define the study area
  res = 25, # The resolution of the grid, in metres
  crs = proj4string(meuse_pts_proj) # The coordinate system for the grid
)
meuse_grd[] <- 0 # Fill the grid with nonsense values
```

DUe to the demands of the function we will be using for spatial prediction, we have to coerce this to a `SpatialPixelsDataFrame` from its current state as a `RasterLayer` object.

```r
meuse_spx <- as(
  meuse_grd,
  "SpatialPixelsDataFrame"
)
```

Making predictions

There are several ways to make predicitons in space. We are going to use a relatively simple method called 'inverse distance-weighted interpolation'. This is implemented using the `idw` function from the `gstat` package.

This method infers the value of lead concentrations on the grid we defined earlier based on the proximity of each grid cell to its surrounding observed points. Closer observed points have more influence on the value of the inferred lead concentration at that point on the grid.

```
meuse_grd[] <- idw(
  formula = log(lead) ~ 1, # A prediction formula,
  locations = meuse_pts_proj, # The observed data
  newdata = meuse_spx, # The prediction grid
  nmax = 25, # Number of nearest neighbours to use for interpolation
  idp = 3 # The inverse distance weighting power, controls smoothness
)$var1.pred
```

```
## [inverse distance weighted interpolation]
```

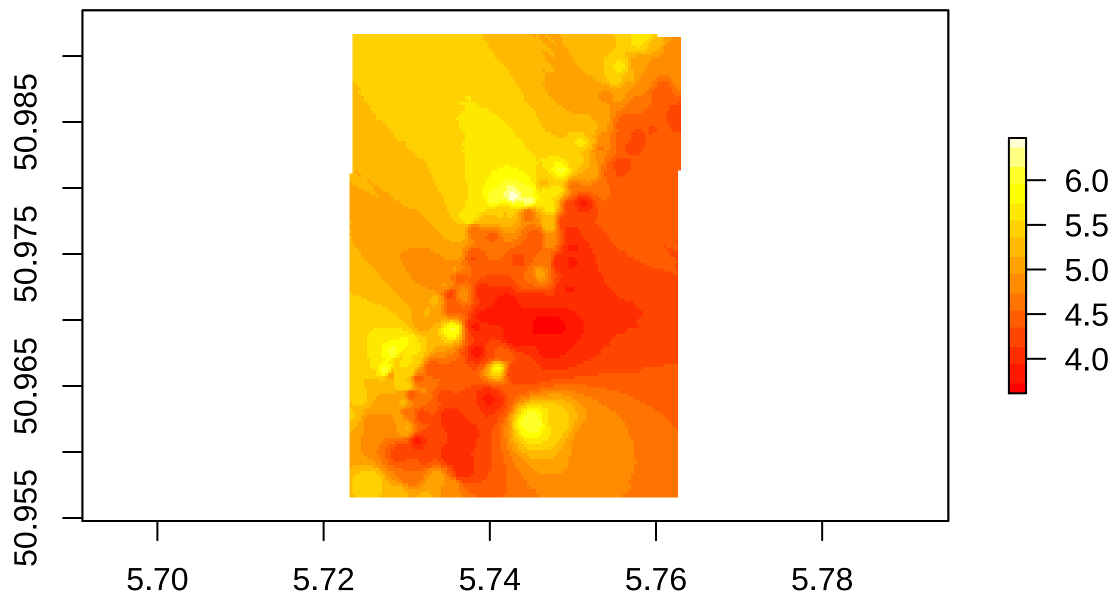
Making sure the predictions make physical sense

Now that we have the predicted values on the grid, we first want to return the grid to geographic coordinates. For this, we have to project the predicted raster.

```
meuse_grd_geo <- projectRaster(
  meuse_grd,
  crs = proj4string(meuse_pts)
)
```

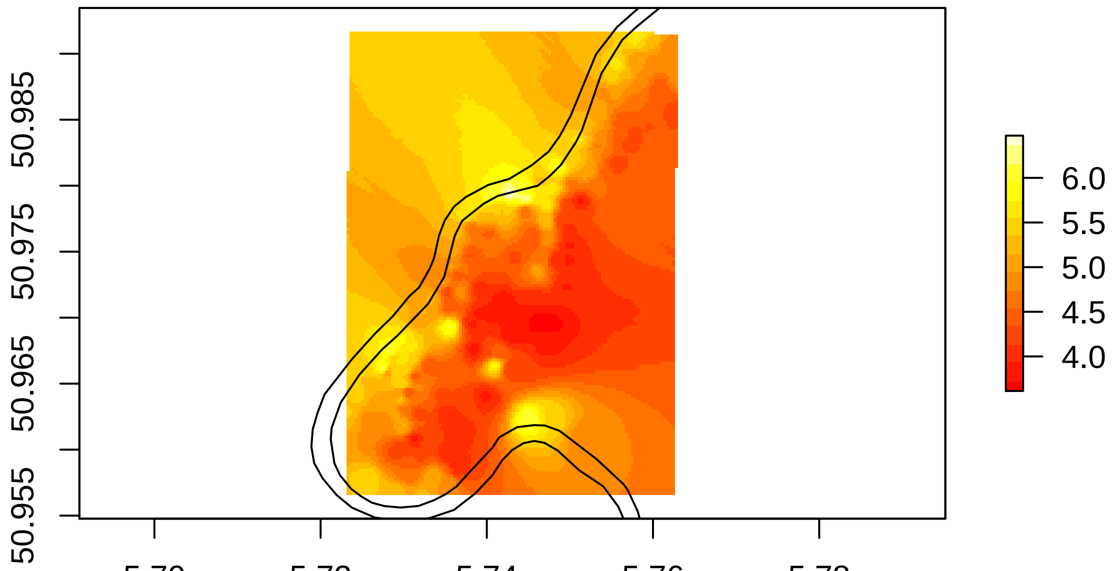
We can then have a look at the log-lead concentrations inferred at every point on the grid.

```
plot(meuse_grd_geo, col = heat.colors(15))
```



One thing to note is that, since these are soil concentrations, it does not make sense to predict values inside the river.

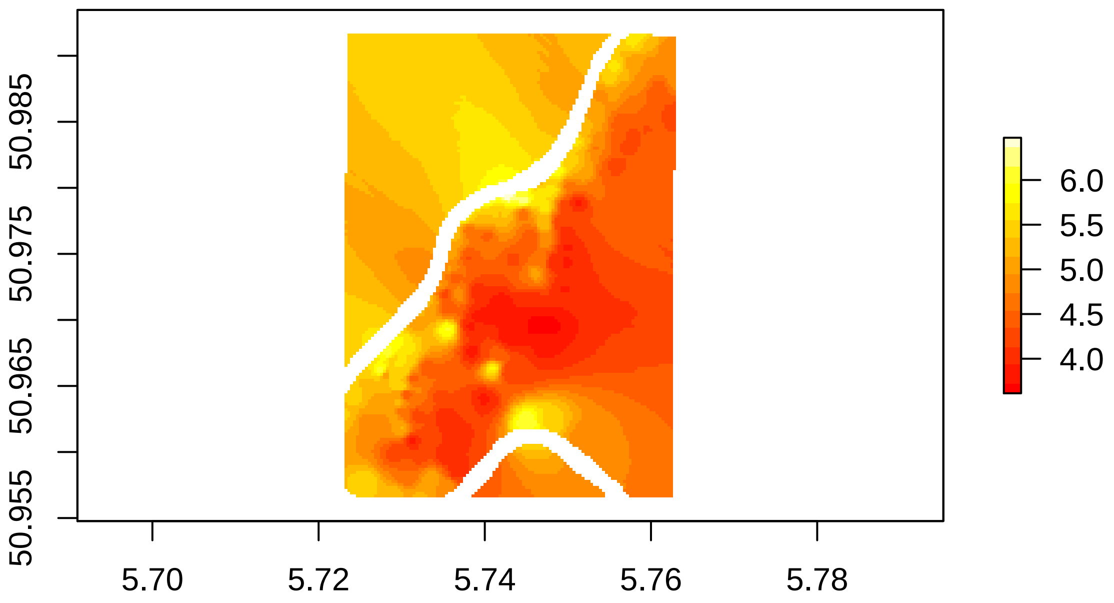
```r
plot(meuse_grd_geo, col = heat.colors(15))
plot(meuse_riv, add = TRUE)
```



We have to mask out the river.

```r
meuse_grd_geo_mask <- mask(
  meuse_grd_geo,
  meuse_riv,
  inverse = TRUE
)

plot(meuse_grd_geo_mask, col = heat.colors(15))
```



Now we have a set of predictions that make physical sense. You can affect the predictions by playing around with the `nmax` and `idp` parameters inside `idw`. Smaller `nmax` in particular will produce visually striking but less meaningful results. Try it out!

Bonus: more advanced visualisation tools

In this section, we will plot our prediction raster over a satellite image from Google Earth. In short, our workflow is

1. Load the `ggmap` package
2. Download a basemap in our study area
3. Convert our data to `data.frame` format
4. Plot in `ggmap`

The first step is easy.

```r
library(ggmap)
```

```
## Loading required package: ggplot2
```

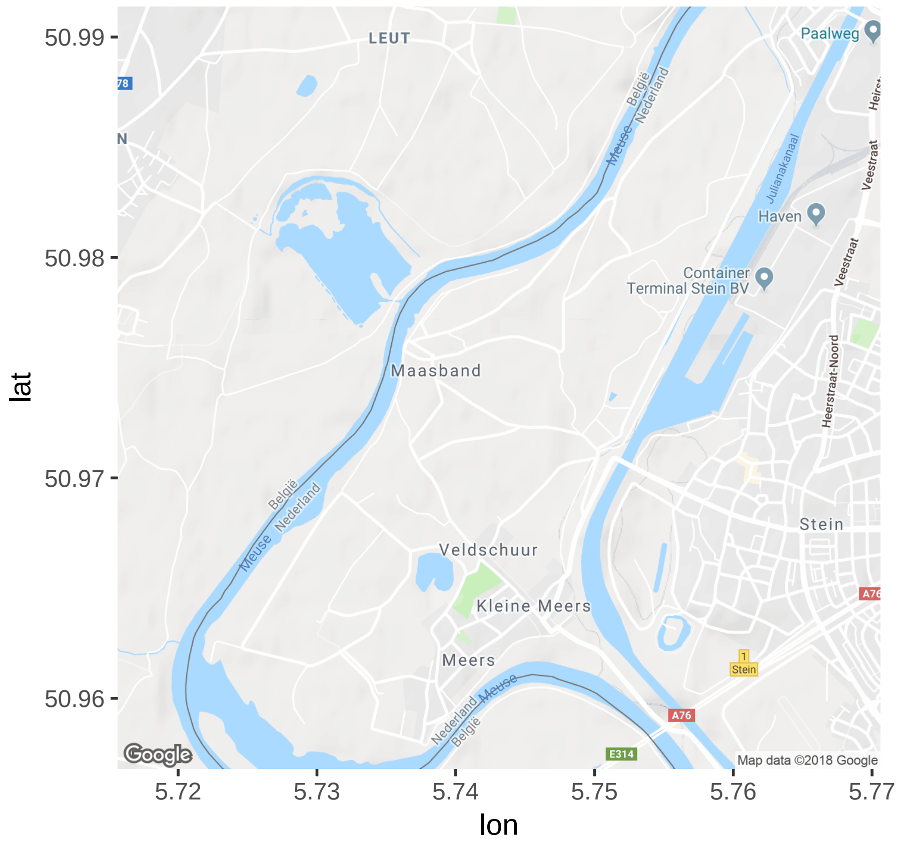
The second step needs to be done in two parts. First, we need to get a coordinate to use as Google Earth's reference for a download. We can simply find the centre of our study area.

```r
meuse_bbx <- bbox(meuse_grd_geo) # Extract bounding box
meuse_ctr <- rowMeans(meuse_bbx) # Find centre of box
```

Then we need to download the base map using the `get_map` command. We need to confirm we got the right area by plotting it.

```
meuse_map <- get_map(
  location = meuse_ctr, # centre of study area
  zoom = 14
)

ggmap(meuse_map)
```



We can now add our other map elements, once they are appropriately processed. The processing we need to do is to coerce our `RasterLayer` to a `data.frame` object. We must first convert it to a `SpatialPixelsDataFrame` object because this will ensure the coordinates of each cell in the raster are preserved in the `data.frame` format.

```
meuse_spx_geo <- as(
  meuse_grd_geo_mask,
  "SpatialPixelsDataFrame"
)
meuse_gdf <- as.data.frame(meuse_spx_geo)
head(meuse_gdf)
```

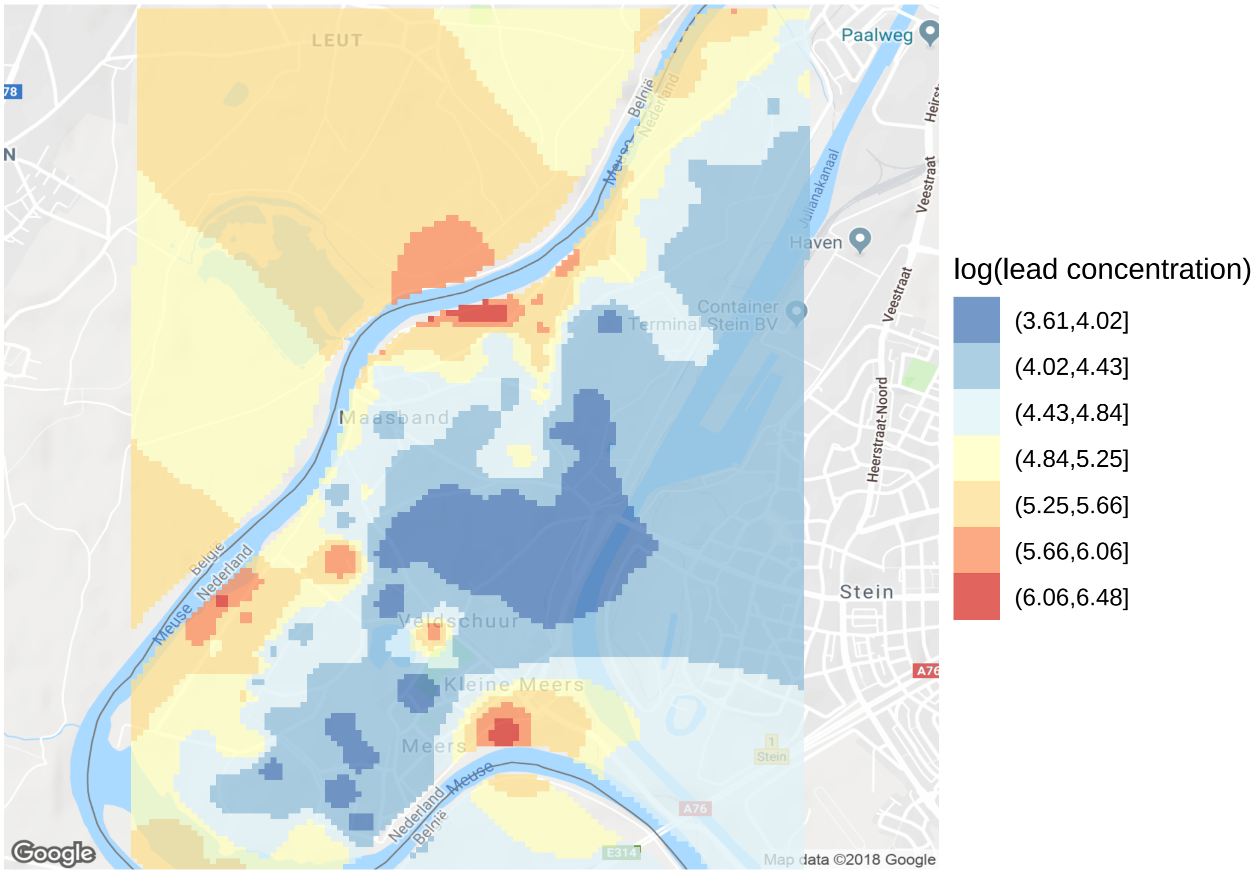
```
##      layer        x        y
## 1 5.431805 5.723679 50.99156
## 2 5.433145 5.724035 50.99156
## 3 5.434505 5.724391 50.99156
## 4 5.435306 5.724747 50.99156
## 5 5.435149 5.725103 50.99156
## 6 5.443359 5.725459 50.99156
```

The other thing we want to do is to discretise the values in the raster, just for display purposes.

```r
meuse_gdf$discrete <- cut(meuse_gdf$layer, 7)
```

Now we can begin putting all these elements together into a plot.

```r
ggmap(meuse_map) +
  ## The remainder of the code is constructed like normal ggplot2 code
  geom_raster(
    data = meuse_gdf,
    aes(
      x = x,
      y = y,
      fill = discrete
    ),
    alpha = 0.75
  ) +
  scale_fill_brewer(palette = "RdYlBu", direction = -1) +
  theme_void() +
  coord_cartesian() +
  labs(fill = "log(lead concentration)")
```


LEUT
Paalweg
Julianakanaal
Haven
Container Terminal Stein BV
Maasband
Veldschuur
Kleine Meers
Meers
Stein
Maas
België
Nederland
Meuse
A76
E314
Google
Map data ©2018 Google
log(lead concentration)
(3.61,4.02]
(4.02,4.43]
(4.43,4.84]
(4.84,5.25]
(5.25,5.66]
(5.66,6.06]
(6.06,6.48]