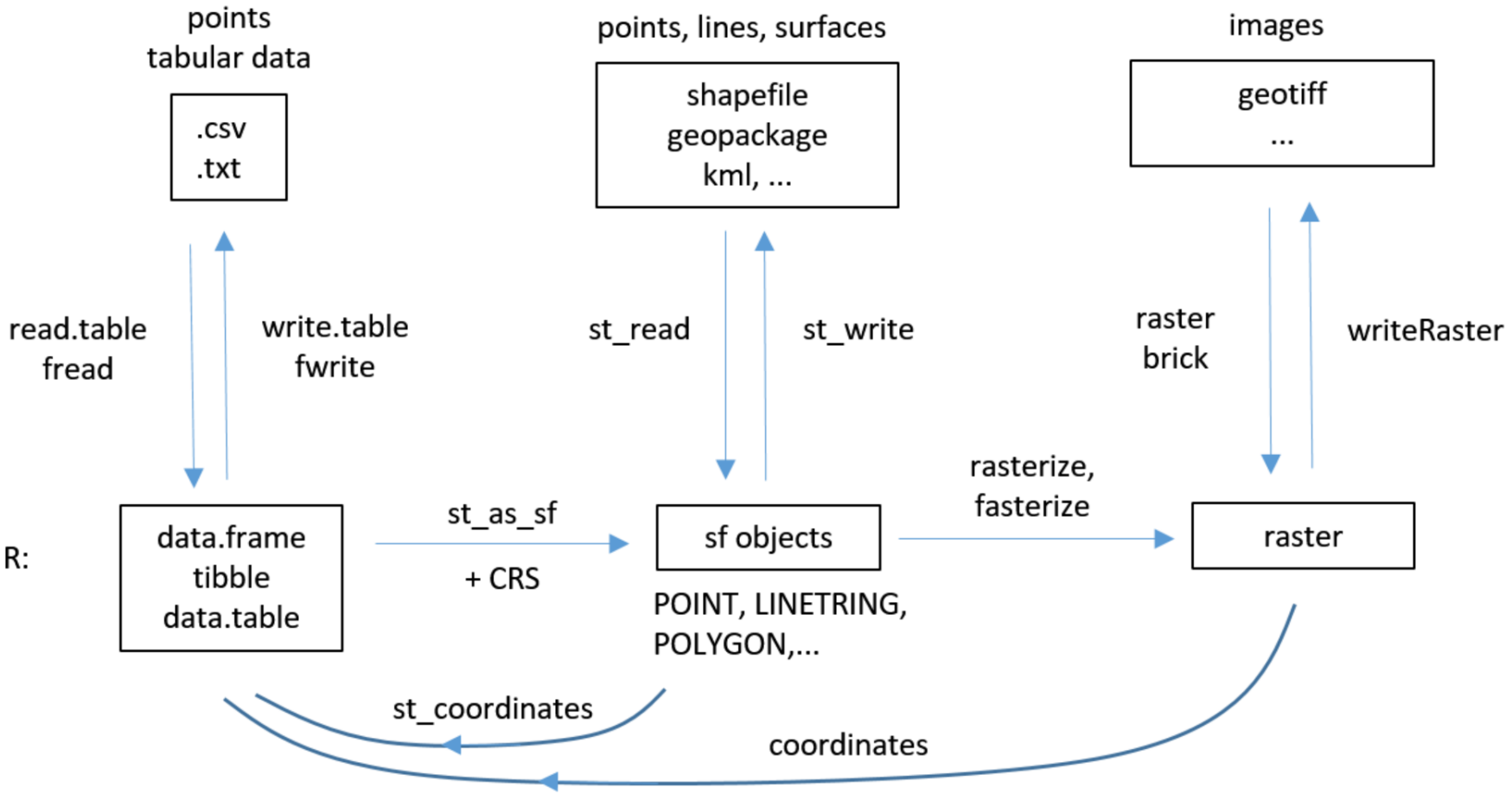
Geographic Data Sets with R

raster and vector data sets

Manuel Campagnolo

Instituto Superior de Agronomia, Universidade de Lisboa

Processing geographic data with R: data formats



R packages for geographic data

The following site updates regularly the many available packages for geographic and spatial processing:

> *The Comprehensive R Archive Network's task view "Analysis of Spatial Data", by Roger Bivand.*

The major packages to read and create spatial objects in R are `raster` and `sf`. In addition we will use `mapview` for maps and a few other packages for specific tasks.

```
library(raster)
library(sf)
library(mapview)
library(RANN) # fast nearest neighbors
library(interp) # linear spatial interpolation
library(spdep) #  spatial data analysis, includes knn2nb
library(tidyverse) # packages for data science, includes %>%
```

R packages for geographic data (cont')

Some spatial analysis packages still require objects of the older vector class `sp`, so when necessary, `sf` objects are converted into `sp` objects using general purpose function `as` or `as_Spatial` from package `sf`.

```
nc <- st_read(system.file("shape/nc.shp", package="sf"), quiet
    = TRUE)
nc_sp <- as(nc, Class = "Spatial") # convert from sf to sp
nc_sp <- as_Spatial(nc) # convert from sf to sp
nc_sf <- st_as_sf(nc_sp, "sf") # convert from sp to sf
```

Some differences between `sf` and `sp`:

1. While `sf` spatial objects have a *tidy* structure (a simple table), `sp` objects have a more complex structure made of slots (`@`) and lists;
2. Reading/writing large spatial data sets is much faster with `sf` than with `sp`;
3. Furthermore, `sf` complies with the OGC (Open Geospatial Consortium) Simple Feature standard while `sp` does not.

Vector geographic data sets

Vector data sets: a simple example with point data

The Aragonez data set will be explored in the remaining of the course. It contains the yield (kg of grapes) measured at 1019 locations in a vineyard in Alentejo. Distances between neighbors are approximately 2.75 m to 3.75 m.

```
1 Aragonez<-read.table(file.path(getwd(),"datasets","Aragonez.txt
    "),header=TRUE)
2 head(Aragonez,3)
```

```
  genotype block col row colm  rowm yield       lon      lat
1    RZ717    B1   4   2    0 93.75 2.417 -7.516431 38.44193
2   RZ1158    B1   4   9    0 67.50 2.724 -7.516291 38.44172
3   RZ1325    B1   4   6    0 78.75 2.647 -7.516351 38.44181
```

Columns `lon` and `lat` are longitudes and latitudes at the 1019 locations.

The coordinate reference systems for those coordinates is known as WGS84 and has EPSG code 4326 (EPSG stands for European Petroleum Survey Group).

Vector data sets: the Aragonez point data set

The Aragonez table is easily converted into a POINT geometry `sf` object with `sf::st_as_sf`.

```
1 AragonezSF<-st_as_sf(Aragonez, coords=c("lon","lat"),crs=4326)
```

```
Simple feature collection with 1019 features and 7 fields
geometry type:  POINT
dimension:      XY
bbox:           xmin: -7.516431 ymin: 38.44118 xmax: -7.515039 ymax: 38.44
epsg (SRID):    4326
proj4string:    +proj=longlat +datum=WGS84 +no_defs
First 10 features:
   genotype block col row colm  rowm yield                    geometry
1     RZ717    B1   4   2    0 93.75 2.417 POINT (-7.516431 38.44193)
2    RZ1158    B1   4   9    0 67.50 2.724 POINT (-7.516291 38.44172)
3    RZ1325    B1   4   6    0 78.75 2.647 POINT (-7.516351 38.44181)
```

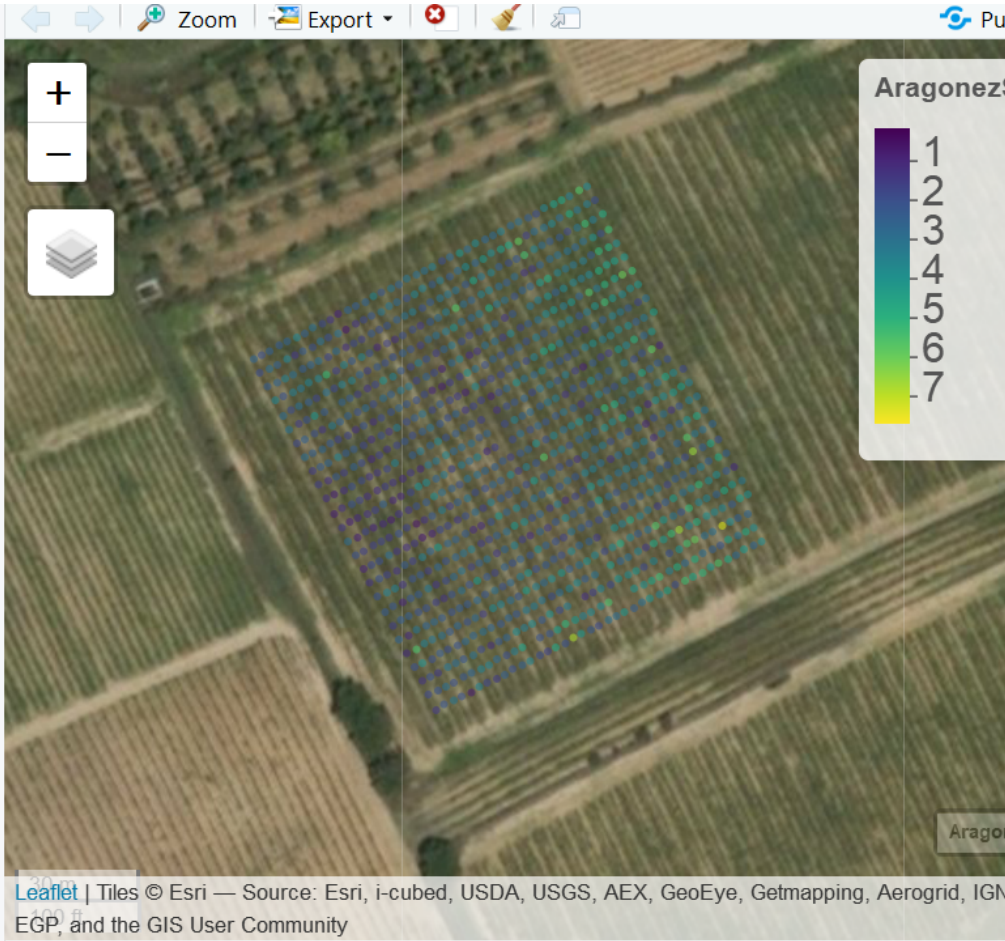
`AragonezSF` is a table (`data.frame`) with a special column named `geometry`. This column is an object of class `sfc`, which stands for *simple feature geometry list-column*.

Vector data sets: mapview

R package `mapview` allows interactive visualizations of spatial data with or without background maps. The basic visualization function is called `mapview`.

```
1 mapviewOptions(basemaps="
      Esri.WorldImagery") #
      optional
2 mapview(AragonezSF, cex=2,
      zcol="yield",lwd=0)
```

`zcol` indicates the column to be rendered.



Vector data sets: a simple example with point data

Since geographic coordinates (longitude and latitude) are not adequate to measure distances, angles and areas, it is recommended to work over a cartographic CRS suitable for the study area.

For Continental Portugal, the official CRS is named ETRS89-TM-PT06 and has EPSG code 3763. Function `sf::st_transform` does the reprojection.

```
Aragonez3763<-st_transform(AragonezSF, crs=3763)
```

Finally, `Aragonez3763` can be exported as *shapefile*, *geopackage*, *kml* or other vector format with `st_write`.

Available drivers can be listed with `st_drivers()`.

```
st_write(Aragonez3763, "Aragonez3763.shp")
```

Determining neighbors in spatial point data sets

Determining neighbors in spatial point data sets

The concept of *neighborhood* is crucial to analyse spatial data and will be at the heart of the statistical techniques discussed in the course. Essentially, one wants to determine, for each data feature (e.g. a POINT in the Aragonez data set), which are its neighbors.

Package `spdep` to be introduced later in the course, gives many options to easily define neighborhoods for spatial data. Here, we explore `sf` functions and the structure of `sf` spatial objects.

```
D <- st_distance(Aragonez3763) # matrix N*N
nn<-col(D)  #  assume that all features are neighbors
nn[as.numeric(D)>5]<-NA # NA is assigned to non neighbors
head(nn[,1:10],3)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   NA   NA   NA   NA    6   NA   NA   NA    NA
[2,]   NA    2   NA    4   NA   NA   NA    8   NA    NA
[3,]   NA   NA    3   NA   NA   NA   NA   NA   NA    10
```

Matrix `nn` above indicates that feature 1 is a neighbor of feature 1 and feature 6, feature 2 is a neighbor of features 2, 4 and 8, and so on.

Determining neighbors in spatial point data sets: `knn` object

To be able to use `nn` (of class `matrix`) for spatial analysis with the `spdep` package, it needs to be first converted into an object of class `knn`, which requires a list with the following components:

1. the matrix `nn`,
2. the number of features `np`,
3. the number `k` of columns of `nn`,
4. `dimension=2` since there are two coordinates, and
5. the matrix of coordinates `x`.

```
1 xy<-st_coordinates(Aragonez3763)
2 NN<-structure(list(nn=nn, np=nrow(nn), k=ncol(nn), dimension=2,
      x=xy), class="knn")
```

Function `sf::st_coordinates` returns the matrix of coordinates of an `sf` spatial object.

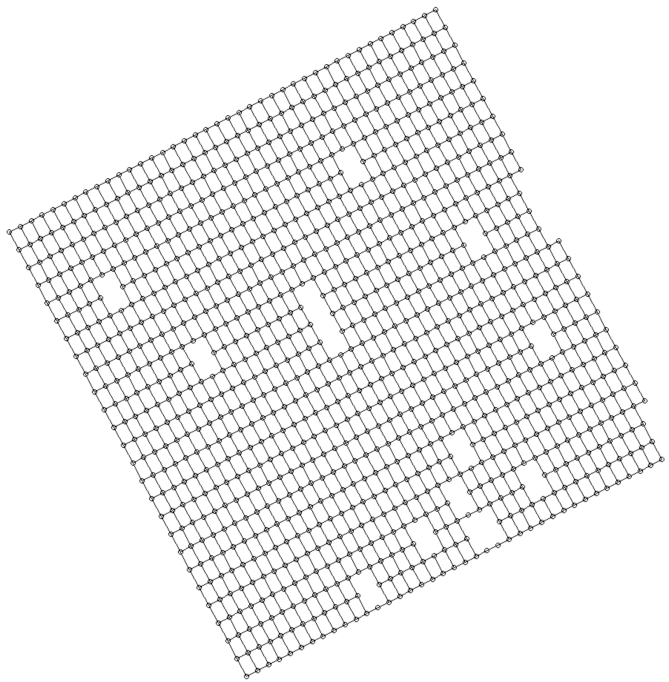
Determining neighbors in spatial point data sets: `knn` object

In alternative to `sf::st_distance`, one can use `sf::st_is_within_distance` which returns a logical matrix, and proceed similarly to define `nn`.

```
D <- st_is_within_distance(Aragonez3763,dist=4,sparse=FALSE) # logical N*N
nn<-col(D)#  assume that all features are neighbors
nn[!D]<-NA # NA is assigned to non neighbors
```

Function `knn2nb` will be described later in the course.

```
xy<-st_coordinates(Aragonez3763)
NN<-structure(list(nn=nn, np=nrow(nn), k=ncol(nn), dimension=2, x=xy), class="knn")
plot(spdep::knn2nb(NN),coords=NN$x)
```



Determining neighbors in spatial point data sets: `nn2` function

For large data sets the construction above is not convenient since it requires a very large matrix `D`. Alternatively, one should use function `RANN::nn2` to create a more compact matrix `nn`.

```
xy <- st_coordinates(Aragonez3763)
nn<-nn2( xy , k=30, searchtype="radius",radius=4)$nn.idx
nn[nn==0]<-NA
head(nn[,1:10],3)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   28    6   NA   NA   NA   NA   NA   NA    NA
[2,]    2   29    4    8   NA   NA   NA   NA   NA    NA
[3,]    3   34   10   11   NA   NA   NA   NA   NA    NA
```

The resulting matrix indicates that the neighbors of feature 1 are features 1, 28, and 6, the neighbors of feature 2 are features 2, 29, 4 and 8, and so on.

Determining neighbors in spatial point data sets: `nn2` function

If we rely on function `RANN::nn2`, then we explore the fact that the output of `nn2` is a list with two components: `$nn.idx` is the matriz of indices of neighbors, and `$nn.dists` is the corresponding matrix of distances.

```
nn<-nn2( xy , k=30)$nn.idx  # matrix 1019*30 of indices
d<-nn2( xy , k=30)$nn.dists # matrix 1019*30 of distances
```

Suppose that only groups of plants which either belong to the same vineyard column and are at most 4 meters apart, or belong to distinct vineyard columns but are at most 3 meters apart, should be neighbors.

```
idxcol<-Aragonez3763$col
nn[(idxcol[nn]==idxcol & d >4) | (idxcol[nn]!=idxcol & d>3)]<-
    NA
```

Here, the attribute of interest is the vineyard column number for each feature (vector `idxcol`) but this construction can be easily adapted to any other data atributes. It is used to restrict the matrix of distances to just the desired entries.

Vector data sets: multiple geometries

Objects `sf`, `sfc` and geometries

A `sf` object has a table structure with a special column called *geometry*. The main simple feature geometry types are POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON and GEOMETRYCOLLECTION as the mixed type.

The geometry of features in a sp object can be extracted with function `st_geometry`, which returns a list of the geometries of all features (the list is an object of class `sfc`, for simple feature geometry list-column).

```
1 st_geometry(Aragonez3763)
```

```
Geometry set for 1019 features
geometry type:  POINT
dimension:      XY
bbox:           xmin: 53837.87 ymin: -136045.6 xmax: 53959.69 ymax: -13592
epsg (SRID):    3763
proj4string:    +proj=tmerc +lat_0=39.66825833333333 +lon_0=-8.1331083333
First 5 geometries:
POINT (53837.87 -135962.6)
POINT (53850.27 -135985.8)
```

Manipulating the geometry of a `sf` spatial object

Let's consider the problem of determining a buffer around the *Aragonez* points.

The approach below has three steps.

1. Aggregate all points into a single feature of MULTIPOLYGON geometry; the attributes are lost, and `st_union` returns an `sfc` object;
2. Determine the convex hull of that feature; he convex hull has POLYGON geometry;
3. Derive the buffer, which also has POLYGON geometry.

```
1 AragonezMP<-st_union(Aragonez3763) # MULTIPOINT sfc object (
    with 1019 points)
2 AragonezCHull<-st_convex_hull(AragonezMP) # class sfc with one
    POLYGON
3 AragonezBuffer<- st_buffer(AragonezCHull, dist=3) # class sfc
    with one POLYGON
```

Manipulating the geometry of a `sf` spatial object

We associate to each group of plants its *Voronoi polygon*. Function `sf::st_voronoi` is used to create the Voronoi polygons for the 1019 points in `AragonezMP`.

To make the code easier to read, the next instruction uses `magrittr`'s pipe operator (package `magrittr` is part of the set of pakages called `tidyverse`). The function after `%>%` uses as its first argument the object before `%>%`

```
1 AragonezVoronoi<-AragonezMP %>%
2   st_voronoi() %>% # create Voronoi polygons
3   st_cast()        # cast to  POLYGON
```

```
Geometry set for 1019 features
geometry type:  POLYGON
dimension:      XY
bbox:           xmin: 53713.36 ymin: -136170.1 xmax: 54084.2 ymax: -135796
epsg (SRID):    3763
proj4string:    +proj=tmerc +lat_0=39.66825833333333 +lon_0=-8.1331083333
First 5 geometries:
POLYGON ((53713.36 -136031.2, 53713.36 -135796....
POLYGON ((53752.45 -135796.5, 53755.05 -135796....
```

Manipulating the geometry of a `sf` spatial object

Finally, we "clip" `AragonezVoronoi` with `AragonezBuffer` and we recover the original attributes from `Aragonez3763`.

```
1 AragonezVoronoiBuffer<-st_intersection(AragonezVoronoi,
      AragonezBuffer)
2 Aragonez3763Vor<-Aragonez3763 # copy sf
3 st_geometry(Aragonez3763Vor)<-AragonezVoronoiBuffer # replace
      geometry
```

```
1 mapview(Aragonez3763Vor,zcol
      ="yield",alpha.regions
      =0.7, lwd=0.3)
```

Manipulating the geometry of a `sf` spatial object

Similarly to what was done earlier for points, one can determine neighborhood relations for the areal representation of the Aragonez data set.

```
tol<-1
D <- st_is_within_distance(Aragonez3763Vor,dist=tol,sparse=
    FALSE)# matrix N*N
nn<-col(D)  #  assume that all features are neighbors
nn[!D]<-NA  # non neighbors are NA
head(nn[,1:10],3)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1    2    3    4   NA   NA   NA   NA   NA    NA
[2,]    1    2    3    4    5   NA   NA   NA   NA    NA
[3,]    1    2    3    4   NA    6    7   NA   NA    NA
```

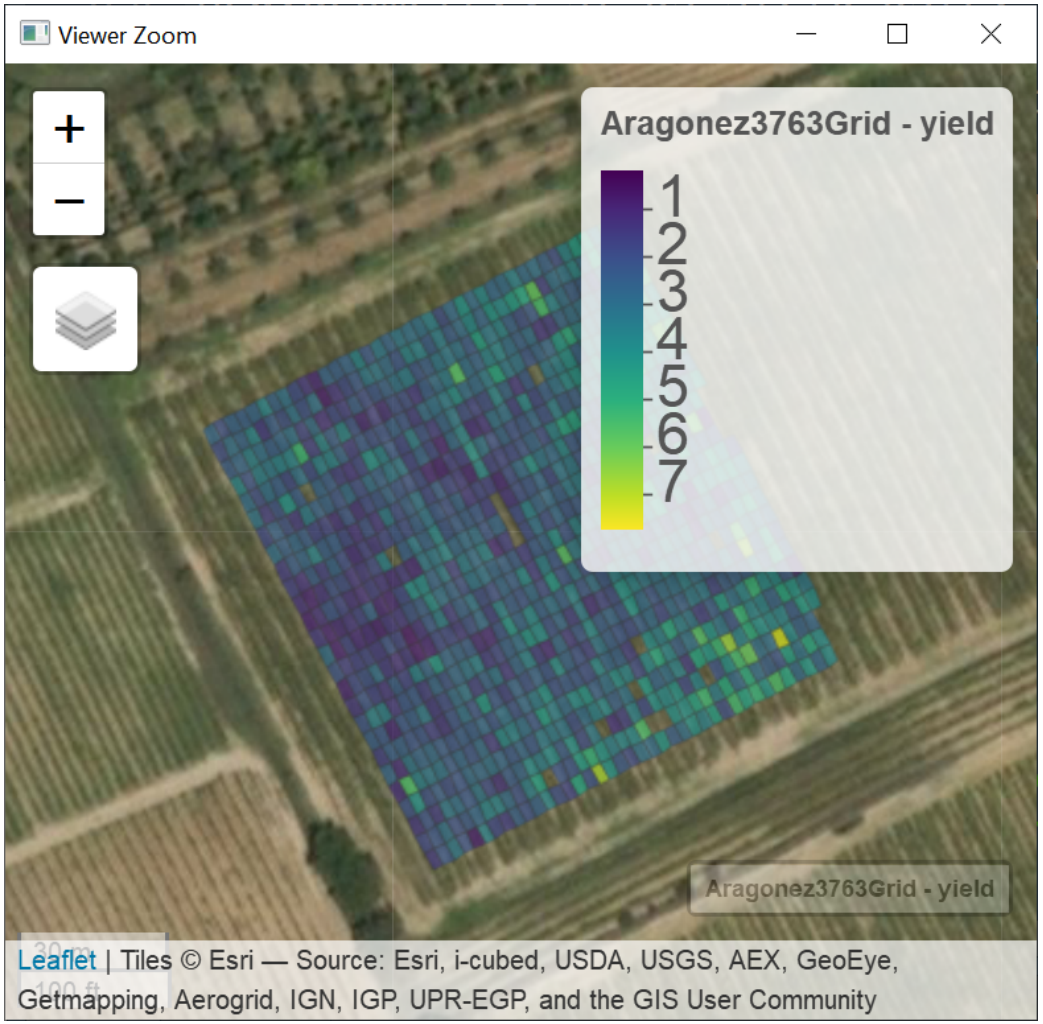
Features on the edges of the field have fewer neighbors but most features have 8 neighbors (besides themselves), as can be checked by counting the number of neighbors of each feature.

```
apply(nn,1,function(x) sum(!is.na(x))) %>% summary() # 4 to 11
```

Exercise: create a regular grid for the Aragonez data set

The goal is to assign to each group of plants a rectangle aligned with the rows of the vineyard. Grid cells should be void for missing data.

One possible solution can be found in sections 3.4 and 3.5 of the book.



Raster geographic data sets

Read images with `raster` and `brick` from package `raster`

In the example below the data set is a multilayer image with three Landsat 8 surface reflectance spectral bands.

```
1 b<-brick(file.path(getwd(),"datasets","LC82030332014151LGN00_sr
      _bands345.tif"))
2 raster::inMemory(b) # FALSE
3 names(b)<-c("band3","band4","band5")
4 print(b)
```

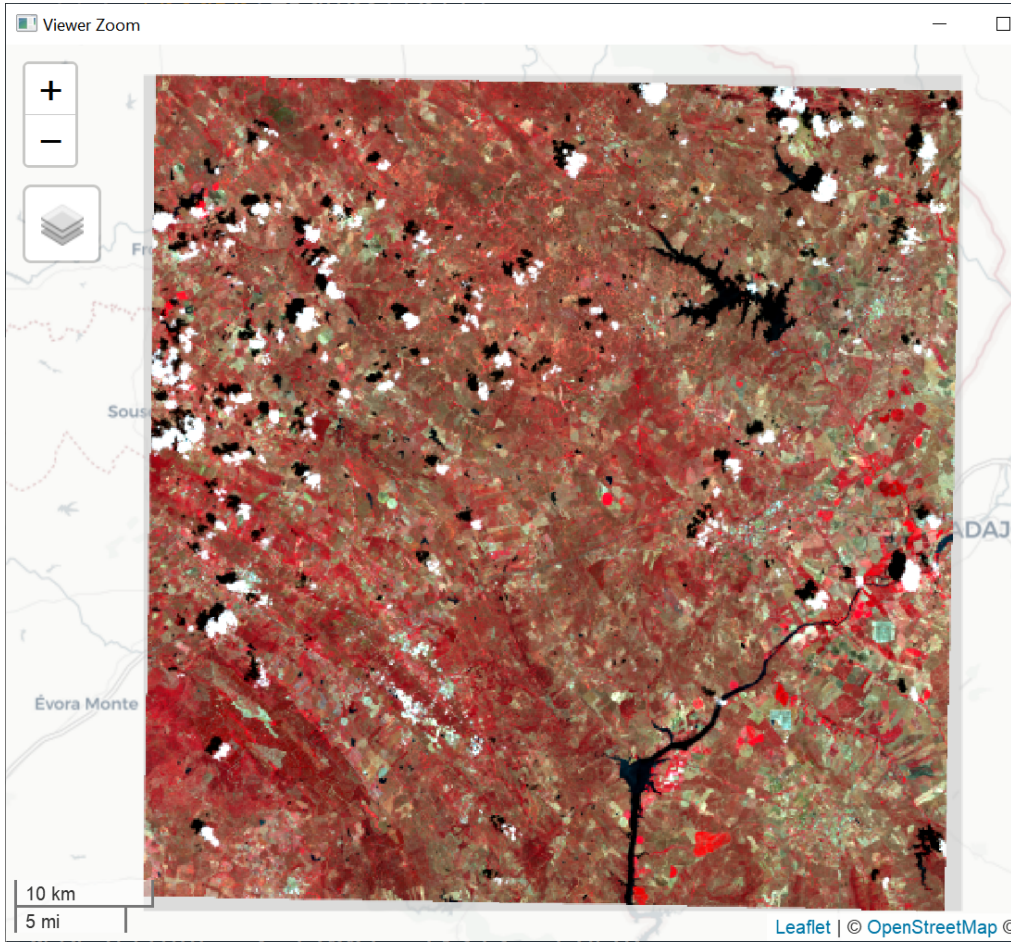
```
class       : RasterBrick
dimensions  : 7791, 7651, 59608941, 3  (nrow, ncol, ncell, nlayers)
resolution  : 30, 30  (x, y)
extent      : 529785, 759315, 4190085, 4423815  (xmin, xmax, ymin, ymax)
coord. ref. : +proj=utm +zone=29 +datum=WGS84 +units=m +no_defs +ellps=WGS
data source : H:\GeographicalData-book-section\datasets\LC82030332014151LG
names       : band3, band4, band5
min values  :   -20,   -88,  -205
max values  : 12661, 13515, 12429
```

Crop and plot images with `mapview`

Function `raster::extent` returns a `extent` object which is altered with `ext/4`. Function `raster::crop` crops the brick of rasters, but it does not load it in memory.

While `mapview` plots one band at a time, `viewRGB` creates a color composite.

```
1 ext<-extent(b) # raster
      extent
2 myb<-crop(b,ext/4) # crop to
      a smaller extent
3 inMemory(myb) # FALSE
4 mapviewOptions(basemaps="
      CartoDB.Positron")
5 mapview::viewRGB(myb,r = 3,
      g = 2, b = 1)
```



Access and manipulate layers of a `raster` or a `brick`

Landsat 8 surface reflectance values are not supposed to be smaller than 0 or larger than 10000, so we assign `NA` to those pixels, before computing the vegetation index `ndvi`.

```
myb[myb<=0 | myb>10000]<-NA # set non valid values
ndvi<-(myb[[3]]-myb[[2]])/(myb[[3]]+myb[[2]]) # access
    individual bands
```

Notice that when pixel values are accessed for calculations, those values have to be loaded in memory.

```
inMemory(myb) #TRUE
inMemory(ndvi) #TRUE
```

If one needs to read a very large file, `raster` can be used to create the connection, and then data can be loaded by blocks of rows with `raster::getValues` and processed one block at the time.

Reproject raster to a new CRS

The CRS of a `raster` spatial object is of class `CRS`.

```
1 crs(b)
```

```
CRS arguments:
 +proj=utm +zone=29 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0
```

One can extract the CRS as a `PROJ string`, which is compatible with `sf`:

```
1 as.character(crs(b))
```

```
[1] "+proj=utm +zone=29 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs
```

To specify the CRS using the EPSG code:

```
1 utm.29<-"+init=epsg:32629" # string that can be interpreted as
      a CRS
```

Reproject a `raster` data set:

```
1 ndvi4326<-projectRaster(from=ndvi,crs="+init=epsg:4326")
```

Access pixel coordinates and values

Pixel values can be extracted (and loaded into memory) with function `raster::values` and coordinates can be extracted with `raster::coordinates`.

Function `raster::rasterToPoints` extract both and removes pixels with `NA` values. All those functions return vectors or matrices.

```
head(values(ndvi),3) # vector
head(values(myb),3) # matrix
head(coordinates(myb),3) # 2-column matrix
head(rasterToPoints(myb),3) # matrix with no NA
```

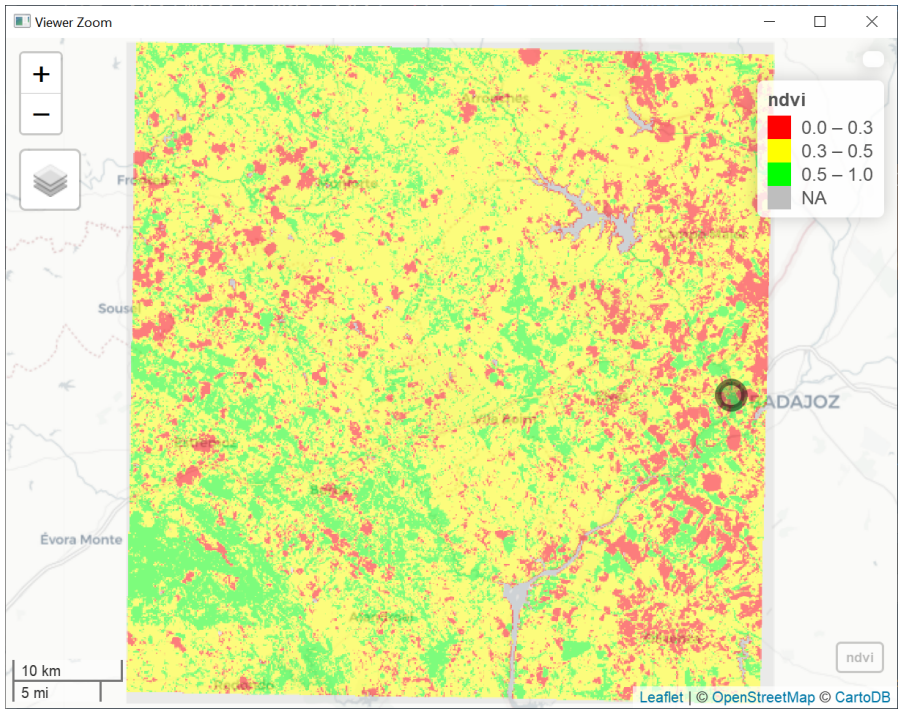
```
           x       y band3 band4 band5
[1,] 615870 4336140   709   972  2250
[2,] 615900 4336140   708   957  2250
[3,] 615930 4336140   732   988  2371
```

Exercise: determine the location where NDVI is highest.

1. Firstly, determine `lon` and `lat` where `ndvi4326` is highest.
2. Secondly, create map, with the location that was found. The map legend should have three classes of NDVI values.

Multiple maps can be superimposed with `mapview(...)+mapview(...)` or with `addFeatures`.

```
1 lonlat<-st_as_sf(data.frame(
      lon=lon,lat=lat),coords=c
      ("lon","lat"), crs=4326)
2 mycolors<-c("red","yellow","
      green")
3 mapviewOptions(basemaps="
      CartoDB.Positron")
4 m<-mapview(ndvi, col.regions
      =mycolors,at=c(0,.3,.5,1)
      , alpha.regions=0.5)
5 m<-addFeatures(m,lonlat,
      color="black", radius=10)
6 m
```



Working example: Las Rosas

The *Las Rosas* spatial data set

Goal: read data from a corn field experiment in Las Rosas, Argentina, and combine the data with relief data extracted from a raster digital elevation model. The *Las Rosas* data set will be explorer further in the remaining of the course.

The original data set is from (Anselin *et al.*, 2001), with columns YIELD (kg/ha), N (amount of fertilizer in kg/ha), and longitudes and latitudes. Distances among points are *approx*71 cm.

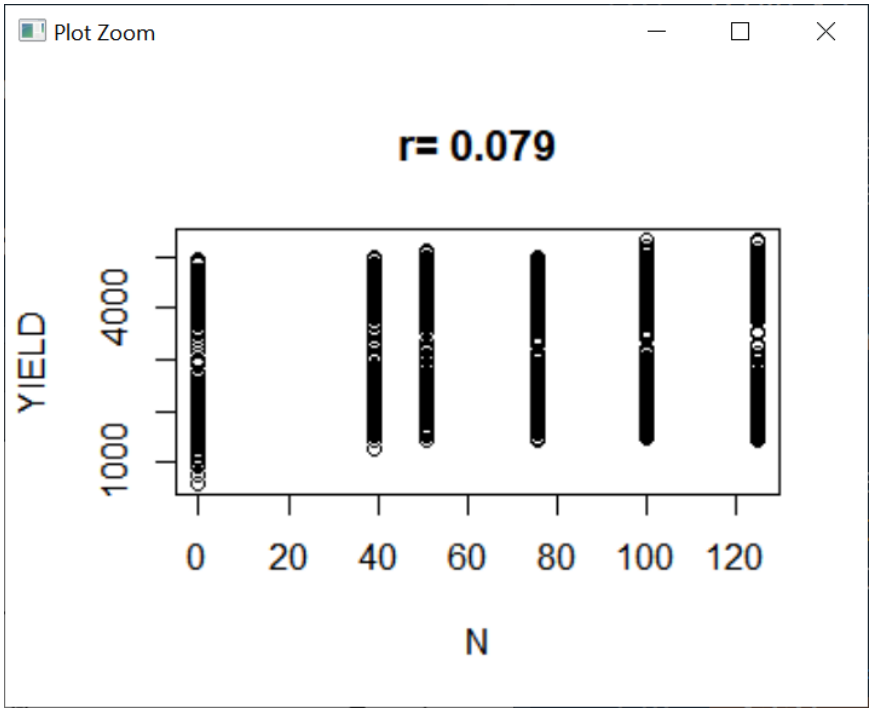
```
1 X<-read.table(file.path(getwd(),"datasets","rosas2001predN-kg-
      ha.txt"),header=TRUE)
```

```
> head(X)
     YIELD     N LONGITUDE  LATITUDE
1 4224.759 124.6 -63.84857 -33.04995
2 4308.220 124.6 -63.84850 -33.04998
3 4300.509 124.6 -63.84843 -33.05000
```

The *Las Rosas* data set

Surprisingly, variables `YIELD` and `N` have very low correlation, which means that that is no hope of modeling the yield using the amount of fertilizer as a the only predictor variable.

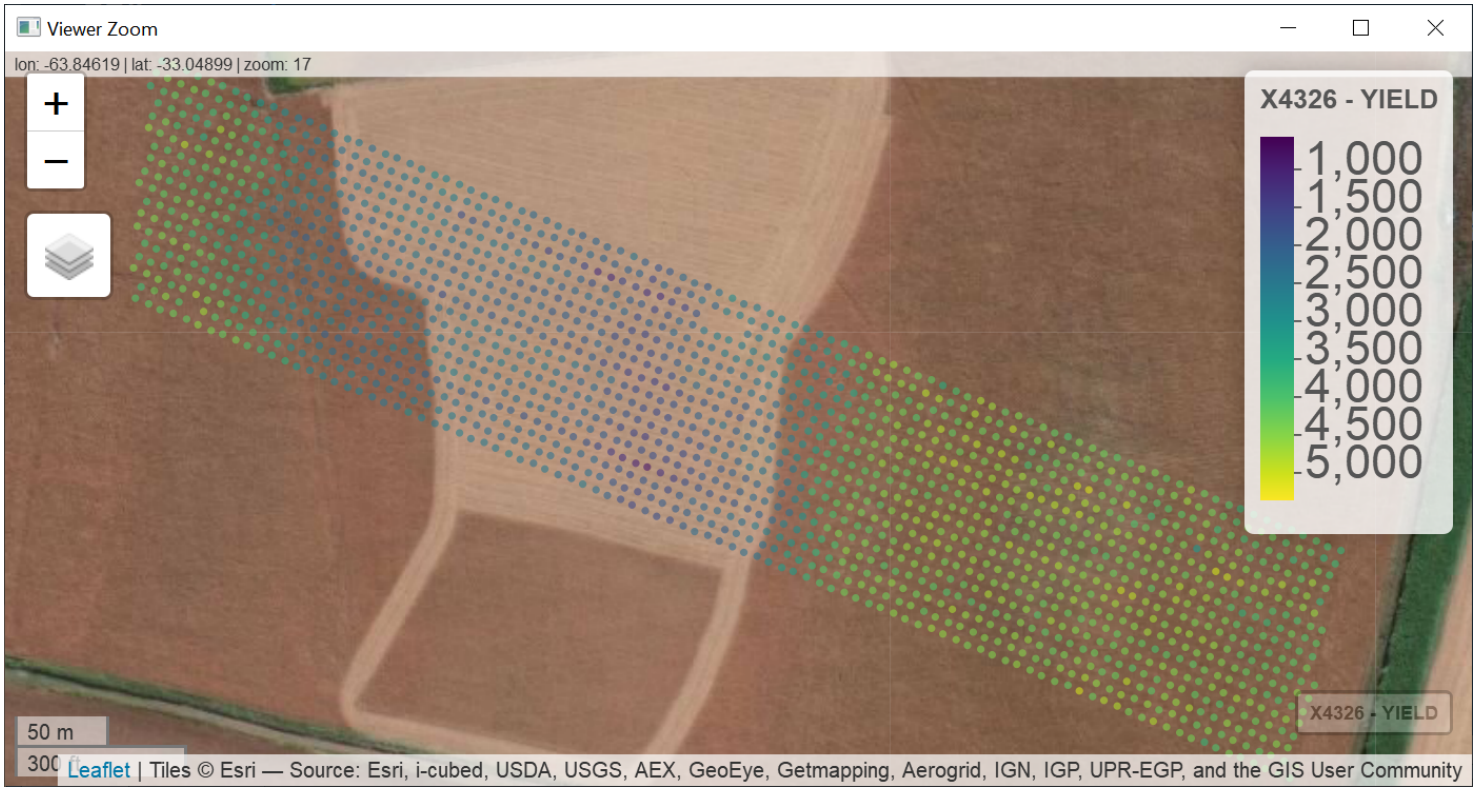
```
1 title<-paste("r=",round(cor(X$YIELD,X$N),3))
2 plot(YIELD~N, data=X, main=title)
```



The *Las Rosas* data set

If we map the data over high resolution imagery, we can suspect that the terrain is not flat, and believe that the relief could be relevant to model the yield.

```
X4326<-st_as_sf(X, coords=c("LONGITUDE","LATITUDE"),crs=4326)
mapviewOptions(basemaps="Esri.WorldImagery")
mapview(X4326,zcol="YIELD",legend=TRUE, cex=1.5, lwd=0)
```



The *Las Rosas* data set: adding elevation data

Several global elevation raster data sets are freely available. The most commonly used is the SRTM digital elevation model which can simply be read from the following URL (it contains the 3 arc-second version ≈90 m resolution). The `if` condition tests if the file `S34W064.hgt` already exists in the user's working directory.

```
if (!("S34W064.hgt" %in% list.files(path=file.path(getwd(),"
    datasets")))) {
urlzip<-"http://dds.cr.usgs.gov/srtm/version2_1/SRTM3/South_
    America/S34W064.hgt.zip"
download.file(url=urlzip,destfile=file.path("datasets","S34W064
    .hgt.zip"),mode="wb")
unzip(zipfile=file.path("datasets","S34W064.hgt.zip")) }
srtm<-raster(file.path(getwd(),"datasets","S34W064.hgt"))
```

A finer resolution version (1 arc-second) can be downloaded, for instance, from Earth Explorer.

```
srtm1<-raster(file.path("datasets","s34_w064_1arc_v3.tif"))
```

The *Las Rosas* data set: adding elevation data

Now we have two sources of data:

1. The original Las Rosas data set with 1704 points;
2. The SRTM data set in raster format for a $1^o \times 1^o$ tile that covers in particular de Las Rosas study area.

The final goal is to combine both data sets and end up with a new `data.frame` with the same 1704 rows, but with additional columns that are going to be derived from the digital elevation model:

```
     YIELD     N     elev      slope   slopeX     accu   aspect    hshade
1 4224.759 124.6 271.5557 0.02204964 13.36769 72.51755 4.460748 0.8636355
2 4308.220 124.6 271.6656 0.02381567 14.52639 70.42829 4.495187 0.8636663
3 4300.509 124.6 271.7812 0.02555609 15.66815 68.31455 4.528715 0.8636961
```

As a local cartographic CRS, we consider the UTM zone for that region of Argentina.

```
1 utm20s<-"+proj=utm +zone=20 +south +ellps=WGS84 +datum=WGS84 +
      units=m +no_defs"
```

The *Las Rosas* data set: cropping elevation data

To reduce the amount of data, let's first crop `srtm` to a vicinity of, say, 500 m of the Las Rosas experiment. We have seen we can crop a raster with `raster::crop` we just need to determine the extent for the crop in longitude and latitude.

```
ext<-X4326 %>%
  st_transform(crs=utm20s) %>%  # reproject to x/y
  st_union() %>%                # create MULTIPOINT object
  st_convex_hull() %>%          # create POLYGON convex hull
  st_buffer(500) %>%            # create POLYGON buffer
  st_transform(crs=4326) %>%    # reproject back to lon/lat
  st_bbox() %>%                 # determine extension
  as.vector()                   # vector xmin ymin xmax ymax
```

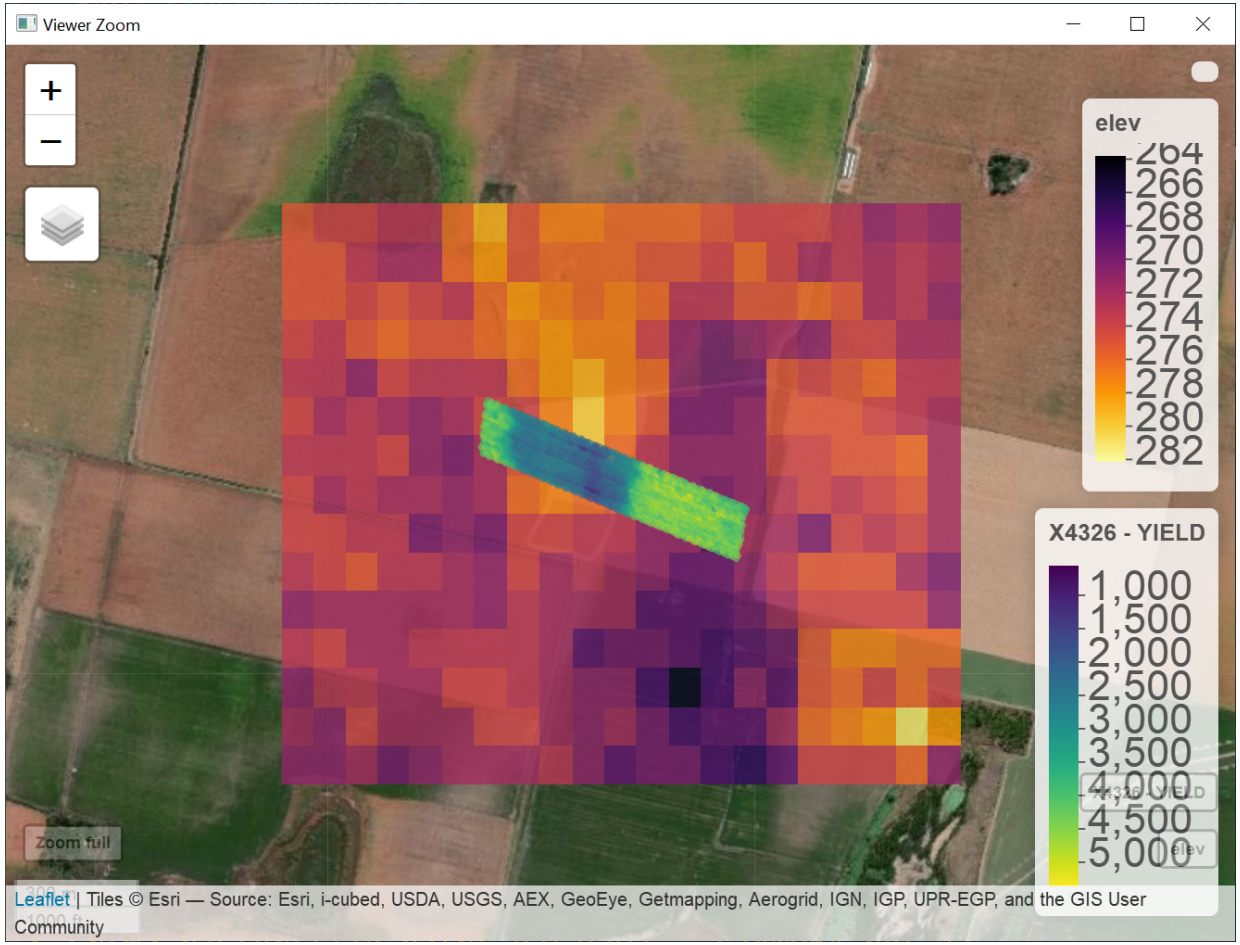
```
[1] -63.85393 -33.05674 -63.83639 -33.04432 # ext
```

Then, the cropped `srtm` is obtained as follows. Since `raster::crop` requires the extent input as *xmin, xmax, ymin, ymax* we have to re-order vector `ext` accordingly.

```
elev<-crop(srtm,y=ext[c(1,3,2,4)])
```

The *Las Rosas* data set: cropping elevation data

```
1 mapview(elev,legend=TRUE)+mapview(X4326, zcol="YIELD",cex=2,lwd
      =0)
```



The *Las Rosas* data set: deriving slope and other relief variables

From `elev`, new relief variables can be derived. In particular, we use function `raster::terrain` to compute the slope and the aspect from `elev`. All outputs are of class `raster`.

Function `raster::focal` allow us to apply an arbitrary linear filter and define its kernel.

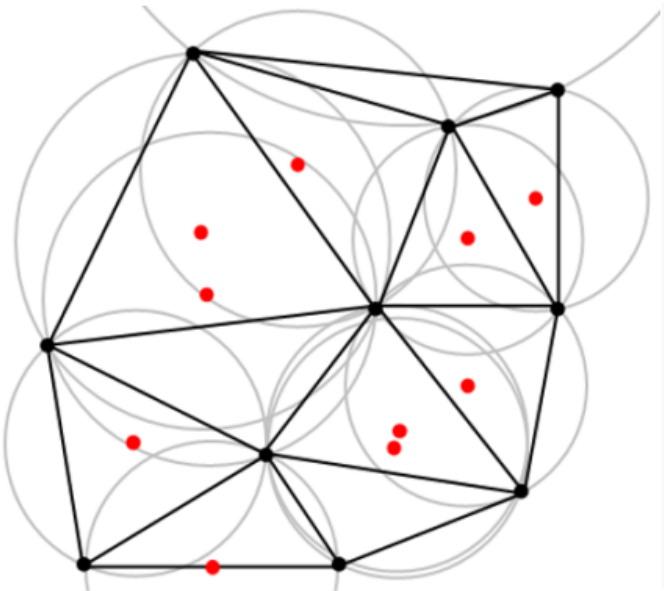
Yield could also be related to the amount of solar radiation that each location gets. This can be measured by the cosine of the incidence angle of the radiation (0 if the radiation is normal to the surface), which is estimated by function `raster::hillShade` given some position for the sun over the horizon.

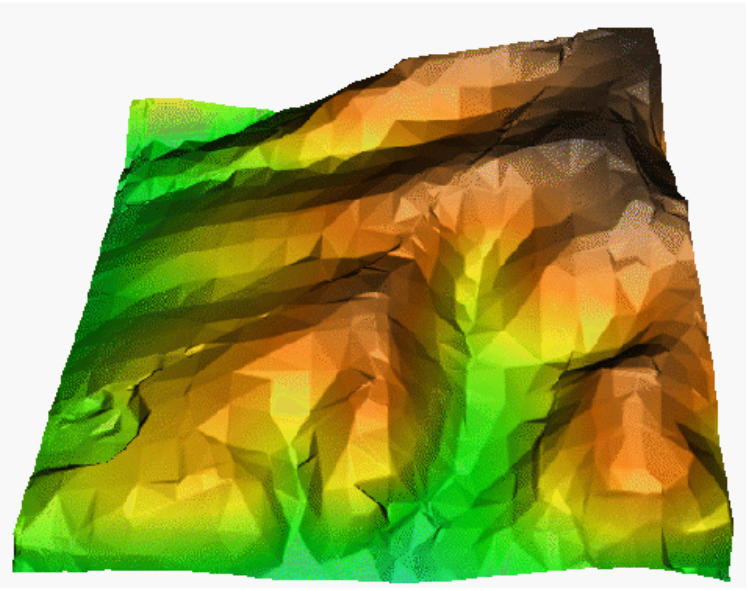
```
slope<-terrain(elev,opt="slope", unit="radians")
aspect<-terrain(elev,opt="aspect",unit="radians")
slopeX<-focal(elev,w=matrix(c(-1,-2,-1,0,0,0,1,2,1),ncol=3))
accu<-focal(slopeX,w=matrix(c(-1,-2,-1,0,0,0,1,2,1),ncol=3))
hshade<-hillShade(slope=slope,aspect=aspect,angle=60,direction
    =0) # direction is the azimuth with respect to North
```

The *Las Rosas* data set: resampling slope and othervariables

We need to combine both sources of data – `X4326` and all raster data sets – into a single table to be able to applied statistical modeling tools to the data. Since the spatial resolution of the Las Rosas point experiment is much finer than the raster data sets, we use it as a the reference and we spatially interpolate the raster data to estimate relief variables values at each one of the 1704 location of `X4326`.

We consider the pixel centers, and their values an apply a linear interpolation to obtain estimates at arbitrary locations.





The *Las Rosas* data set: resampling slope and other variables

That are many R packages that provide interpolation functions. Here we use `interp::interp`. Since we have to apply it multiple times, let us define a function to facilitate our task:

Function `raster::rasterToPoints` returns the centers of the pixels and their values. Function `interp` performs the interpolation and compute the interpolated values at arbitrary locations, defined by `x0` and `y0`. The new function `interpolate` has inputs `r` (the raster to be interpolated), and vectors `longs` and lats of arbitrary points where the interpolated values must be computed.

Here we consider that the CRS of `r` is WGS84, which is not a concern since the study area is very small.

```
1 interpolate<-function(r,longs,lats)
2 {
3   rtp<-rasterToPoints(r)
4   interp(x=rtp[,1],y=rtp[,2],z=rtp[,3], xo=longs,yo=lats,output
        = "points",duplicate="mean")$z
5 }
```

The complete *Las Rosas* data set

To conclude the task we just have to apply `interpolation` to each raster and colect the interpolated values as new columns of `X4326`.

```
longs<-st_coordinates(X4326)[,1]
lats<-st_coordinates(X4326)[,2]
X4326$elev<-interpolate(r=elev,longs,lats)
X4326$slope<-interpolate(r=slope,longs,lats)
X4326$slopeX<-interpolate(r=slopeX,longs,lats)
X4326$accu<-interpolate(r=accu,longs,lats)
X4326$aspect<-interpolate(r=aspect,longs,lats)
X4326$hshade<-interpolate(r=hshade,longs,lats)
```

A little more housekeeping might be necessary to obtain either a `sf` spatial object with cartographic coordinates or a simple table for posterior statistical analysis.

```
Xsfutm<-st_transform(X4326,crs=utm20s) # sf object
X<-X4326 # make copy of sf object
st_geometry(X)<-NULL # remove geometry column
cor(X) # works but cor(X4326) does not.
```

The complete *Las Rosas* data set

Package `mapview` includes a function `sync` that allows to navigate simultaneously over more than one image.

```
m1<-mapview(X4326,zcol="YIELD",cex=0.7,lwd=0,legend=TRUE)
m2<-mapview(X4326,zcol="N",cex=0.7,lwd=0,legend=TRUE)
m3<-mapview(X4326,zcol="elev",cex=0.7,lwd=0,legend=TRUE)
m4<-mapview(X4326,zcol="slope",cex=0.7,lwd=0,legend=TRUE)
sync(m1,m2,m3,m4)
```