SF & Terra

Announcements

- Final Project Pre-Proposal Due today!
  - Again, no late penalties will be applied but this must be submitted before the Final Pre-Proposal due date, and give time for us to meet.
  - Not looking for anything crazy, and if you don't know what analysis you want to do its ok :)
- Final Project Meeting Calendar fixed
- Last class was only a few days ago, so we will have more time to do Labs 2 and 3, with next week's lab being very short to account for this.
- Don't wait to get started on these as they can be long depending on your experience with R.

What are SF & Terra?

- SF & Terra are R integrations of reading, writing, and manipulating vector and raster data respectively.
- SF (Simple Features) : Vector
- Terra : Raster

Package History Background

- Packages and standards are always changing, which is why we need to stay up to date on what is new and in use!
- SF recently overtook the SP package, which is now depreciated
- Terra recently overtook the Raster package, which is now depreciated
- Some packages still depend on SP and Raster, and have not been updated to SF and Terra.
- Keep this in mind for your stack exchange answer interpretations.

1. Vectors in R w/ SF

Simple Features (SF) - Cheat Sheet

- st_read(): read simple features from a file.
- st_write(): write simple features to a file.
- st_as_sf(): converts objects to simple features objects.
- st_transform(): transform / reproject CRS.
- st_crs(): retrieves the CRS of an sf object.
- Geometric Binaries
  - st_intersection(): clip
  - st_union()
  - st_difference(): erase
  - st_buffer()
  - st_area()
  - st_length()

Anatomy of a Shapefile

- **.shp** - The main file that stores the geometry of all features (points, lines, polygons).
- **.shx** - The index file links the attribute data in the .dbf file to the spatial data in the .shp file.
- **.dbf** - The dBASE table that stores attribute data for each shape.
- **.prj** - The projection file that stores the coordinate system and projection information.
- **.sbn** and **.sbx** - These are spatial index files that improve the performance of spatial queries. Not all GIS software will create or use these files.
- **.xml** - An optional metadata file.

Reading Data - Shapefiles

- st_read()
- Note: You only need the .shp which will point sf to the other dependency files
- Example:
  - **polygon = st_read(here::here('data', 'polygon.shp'))**

Reading Data - Shapefile Example

```
nc = st_read(here::here('data', 'nc.shp'))
```

```
Reading layer `nc' from data source
  `C:\Users\James\
  using driver `ESRI Shapefile'
Simple feature collection with 100 features and 14 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
Geodetic CRS:  NAD27
```

Reading Data - Converting Data Frames to SF

- st_as_sf(dataframe, coords = c("Longitude","Latitude"), crs = __)
- USE EPSG CODES FOR CRS ALWAYS (epsg.io)
- Example:
    -

```r
# Example dataframe
df <- data.frame(
  id = 1:3,
  lat = c(40.7128, 34.0522, 41.8781),
  long = c(-74.0060, -118.2437, -87.6298)
)

head(df)
```

```
> head(df)
  id     lat      long
1  1 40.7128  -74.0060
2  2 34.0522 -118.2437
3  3 41.8781  -87.6298
```

```r
# Convert to SF Object
sf <- st_as_sf(df,
               coords = c("long", "lat"), # Specify long and lat columns
               crs = 4326) # Specify projection (WGS84 EPSG)
```

Reading Data - Geodatabases

1. Inspect Geodatabase
   - > gdb_path = here::here(‘data’, ‘default.gdb’)
   - > layers = **st_layers(gdb_path)**
   - > print(layers)
2. Specify Layer in st_read()
   - > layer = layers[[1]][#]   # subset for your layer of interest
   - > data = st_read(gdb_path, **layer = layer**)

Use GDB as input   Specify Layer w/ layer parameter

Reading Data - Limited Geodatabases Example

- If you simply use the gdb location as the st_read() input, it will default to using the first layer in the gdb and ignore the rest.

```
> gdb = st_read(soils_gdb_path)
Multiple layers are present in data source C:\Users\James\Documents\Academic\SDAR\data
\Soils_MassGIS_GDB\Soils_MassGIS.gdb, reading layer `SOILS_MUPOLYGON_TOP20'.
Use `st_layers' to list all layer names and their type in a data source.
Set the `layer' argument in `st_read' to read a particular layer.
Reading layer `SOILS_MUPOLYGON_TOP20' from data source
  `C:\Users\James\Documents\Academic\SDAR\data\Soils_MassGIS_GDB\Soils_MassGIS.gdb'
  using driver `OpenFileGDB'
Simple feature collection with 249771 features and 44 fields
```

Reading Data - Geodatabases Example

1. Retrieve layers from gdb

```
> soils_gdb_path = here::here('data',
+                              'Soils_MassGIS_GDB',
+                              'Soils_MassGIS.gdb')
> soils_gdb_layers = st_layers(soils_gdb_path)
> soils_gdb_layers
Driver: OpenFileGDB
Available layers:
                layer_name     geometry_type features fields
1    SOILS_MUPOLYGON_TOP20     Multi Polygon   249771     44
2 SOILS_POLY_PRIMEFARMLAND     Multi Polygon    94535      7
3       SOILS_SPECFEAT_ARC Multi Line String     4516      7
4        SOILS_SPECFEAT_PT             Point    35542      6
5  SOILS_SURVEY_AREAS_POLY     Multi Polygon       19     10
```

Reading Data - Geodatabases Example

2. Subset layer output list to layer of interest

```
> # Note: st_layers() returns a list
> soils_gdb_layers[[1]]
[1] "SOILS_MUPOLYGON_TOP20"    "SOILS_POLY_PRIMEFARMLAND" "SOILS_SPECFEAT_ARC"
[4] "SOILS_SPECFEAT_PT"        "SOILS_SURVEY_AREAS_POLY"
> soils_gdb_layers[[1]][1]
[1] "SOILS_MUPOLYGON_TOP20"
> soils_gdb_layers[[1]][2]
[1] "SOILS_POLY_PRIMEFARMLAND"
```

Reading Data - Geodatabases Example

3. Utilize st_layers() to choose layer of interest from gdb

```
> layer = soils_gdb_layers[[1]][2]
> soils_primefarm = st_read(soils_gdb_path, layer = layer)
Reading layer `SOILS_POLY_PRIMEFARMLAND' from data source
  `C:\Users\James\Documents\Academic\SDAR\data\Soils_MassGIS_GDB\Soils_MassGIS.gdb'
  using driver `OpenFileGDB'
Simple feature collection with 94535 features and 7 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: 35513.3 ymin: 777671.1 xmax: 330291.3 ymax: 959747.1
Projected CRS: NAD83 / Massachusetts Mainland
```

- > layer = layers[[1]][#]   # subset for your layer of interest

Simple Feature Objects

- sf dataframes store spatial metadata like the CRS and the geometry
  - **head(sf1, 3)** will show you the metadata

Simple Feature Objects - Inspecting Data Example

```
> head(nc, 3)
Simple feature collection with 3 features and 14 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -81.74107 ymin: 36.23388 xmax: -80.43531 ymax: 36.58965
Geodetic CRS:  NAD27
   AREA PERIMETER CNTY_ CNTY_ID      NAME  FIPS FIPSNO CRESS_ID BIR74 SID74
1 0.114     1.442  1825    1825      Ashe 37009  37009        5  1091     1
2 0.061     1.231  1827    1827 Alleghany 37005  37005        3   487     0
3 0.143     1.630  1828    1828     Surry 37171  37171       86  3188     5
  NWBIR74 BIR79 SID79 NWBIR79                       geometry
1      10  1364     0      19 MULTIPOLYGON (((-81.47276 3...
2      10   542     3      12 MULTIPOLYGON (((-81.23989 3...
3     208  3616     6     260 MULTIPOLYGON (((-80.45634 3...
```

Simple Feature Objects

- sf dataframes store spatial metadata like the CRS and the geometry
    - **head(sf1, 3)** will show you the metadata
- Geometry is stored in a list column!
- The geometry list column is a SFC class = **Simple Features Column**
    - **st_geometry**(sf1): Allows you to extract geometry to its own vector

Simple Feature Objects - SF Geometry Example

nc

	NWBIR79	geometry
0	19	MULTIPOLYGON (((-81.47276 3...
3	12	MULTIPOLYGON (((-81.23989 3...
6	260	MULTIPOLYGON (((-80.45634 3...
2	145	MULTIPOLYGON (((-76.00897 3...

- The geometry column contains a list format of coordinates for each element.
- Remember that lines and polygons are made of points, so these geometry lists contain point info.

Name	Type	Value
nc[[15]][[1]]	list [1] (S3: XY, MULTIPOLYGON, s	List of length 1
[[1]]	list [1]	List of length 1
[[1]]	double [27 x 2]	-81.5 -81.5 -81.6 -81.6 -81.7 -81.7 ...

Simple Feature Objects - SF Geometry Tips

- The geometry column will be retained through subsets
  - Keep this in mind if your input requires no geometry column
  - You can utilize the **st_drop_geometry()** function to remove geometry

CRS & Projections

- R cannot project ‘on the fly’ like ArcPro
  - Visually two layers may look properly projected, but to the ArcPro tools they are not!
  - ArcPro tools cannot project on the fly, so CRS/Projections must be manually consistent
- st_crs() : get CRS/Projection
- st_transform() : project & reproject sf objects
- Example: We can ensure matching projectings with these two functions
  - > crs_sf1 = **st_crs(sf1)**
  - > sf2_transformed = **st_transform(sf2, crs = crs_sf1)**
  - > st_crs(sf1) == st_crs(sf2)
  - > TRUE

Projections - Inspecting CRS Example

```
> st_crs(nc)
Coordinate Reference System:
  User input: NAD27
  wkt:
GEOGCRS["NAD27",
    DATUM["North American Datum 1927",
        ELLIPSOID["Clarke 1866",6378206.4,294.978698213898,
            LENGTHUNIT["metre",1]]],
    PRIMEM["Greenwich",0,
        ANGLEUNIT["degree",0.0174532925199433]],
    CS[ellipsoidal,2],
        AXIS["latitude",north,
            ORDER[1],
            ANGLEUNIT["degree",0.0174532925199433]],
        AXIS["longitude",east,
            ORDER[2],
            ANGLEUNIT["degree",0.0174532925199433]],
    ID["EPSG",4267]]
```

Projections - Transforming with EPSG Codes

```
> st_transform(nc, crs = "EPSG:4326")
Simple feature collection with 100 features and 14 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -84.32377 ymin: 33.88212 xmax: -75.45662 ymax: 36.58973
Geodetic CRS:  WGS 84
First 10 features:
    AREA PERIMETER CNTY_ CNTY_ID        NAME  FIPS FIPSNO CRESS_ID BIR74 SID74
1  0.114     1.442  1825    1825        Ashe 37009  37009        5  1091     1
2  0.061     1.231  1827    1827   Alleghany 37005  37005        3   487     0
3  0.143     1.630  1828    1828       Surry 37171  37171       86  3188     5
4  0.070     2.968  1831    1831   Currituck 37053  37053       27   508     1
5  0.153     2.206  1832    1832 Northampton 37131  37131       66  1421     9
6  0.097     1.670  1833    1833    Hertford 37091  37091       46  1452     7
7  0.062     1.547  1834    1834      Camden 37029  37029       15   286     0
8  0.091     1.284  1835    1835       Gates 37073  37073       37   420     0
9  0.118     1.421  1836    1836      Warren 37185  37185       93   968     4
10 0.124     1.428  1837    1837      Stokes 37169  37169       85  1612     1
```

Geometric Binaries - Cheat Sheet

st_intersection(sf1, sf2) : clip

st_difference(sf1, sf2) : erase

st_union(sf1) or st_union(sf1, sf2)

st_buffer(sf1, buf_dist)

- Distance unit is based on CRS

Geometric Operations - Cheat Sheet

st_join(sf1, sf2) or st_merge(sf1, sf2)

st_centroid(sf1) : will find centroids for your

st_area(sf1)

st_length(sf1)

st_distance(sf[1,], sf1[2,])

Simple Feature Manipulation - Geometric Integrity

- Features may not always be valid by the standards of SF
  - Self-Intersecting Polygons
  - Non-Closed Polygons
  - Duplicate Points
  - Incorrect Ring Ordering
  - Holes Outside of Polygons
  - Overlapping or Duplicate Rings
  - Zero-Area Geometries
  - Invalid Geometric Constructions
- st_is_valid(sf)   # Determines validity of features in SF
  - Will return TRUE if feature is valid
  - Will return FALSE if feature is invalid
- st_make_valid(sf) # Fixes invalid features in SF

Simple Feature Manipulation - st_cast()

- st_cast()
- This function is particularly useful when you want to change the type of geometry of an sf object, for example, converting a multipolygon to individual polygons, or a multiline to individual lines.
    - Some functions require a specific sf geometry type, so this is helpful to wrangle data!
- Example:
    - **> sf_polygon <- st_cast(sf_multipolygon, "POLYGON")**

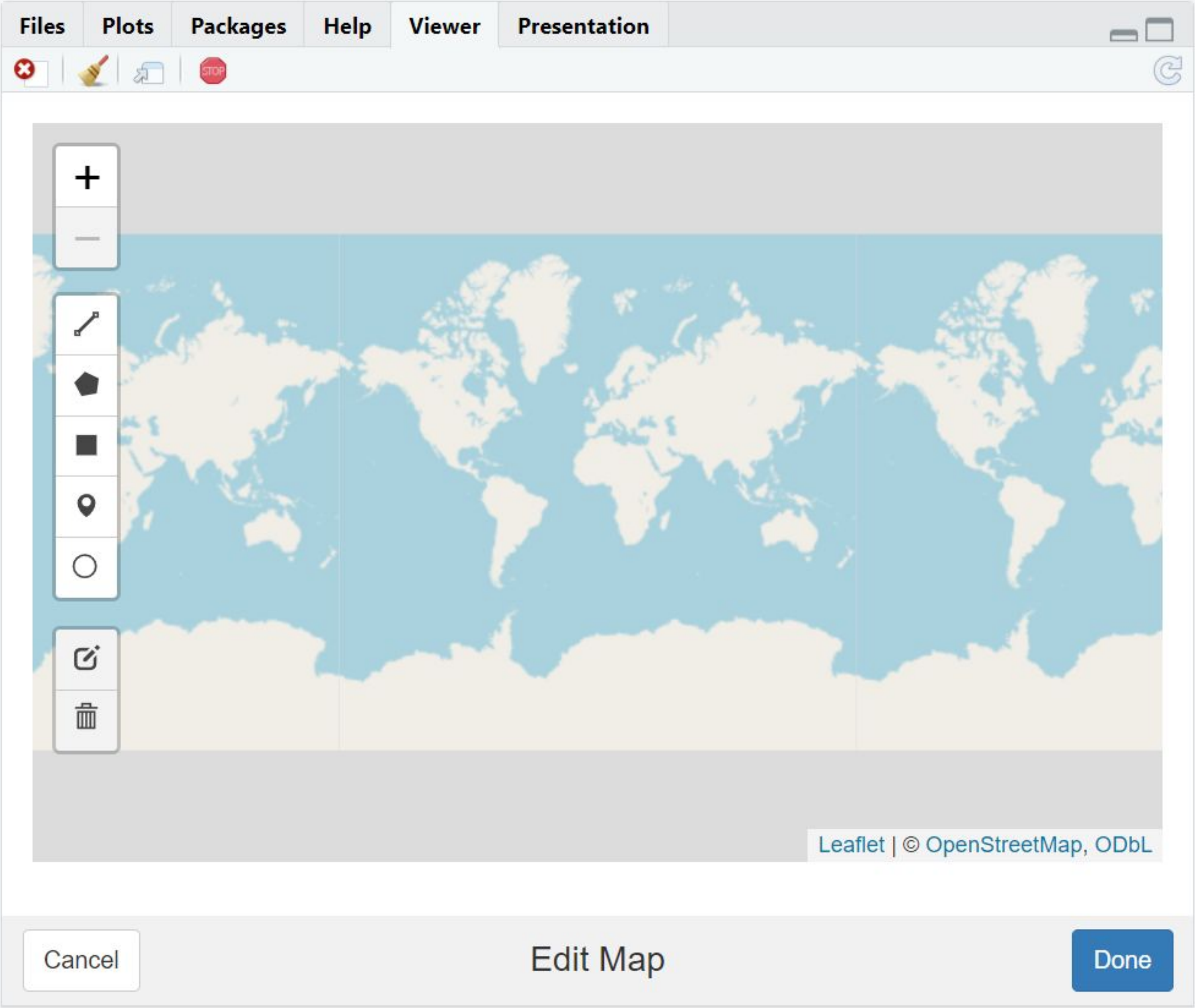
Writing SF Data

- st_write()
- Example:
    - > st_write(points, “points.shp”)
- Write points to .csv w/ coordinates
    - > st_write(points, “points.csv”,
    - layer_options = “GEOMETRY=AS_XY”)

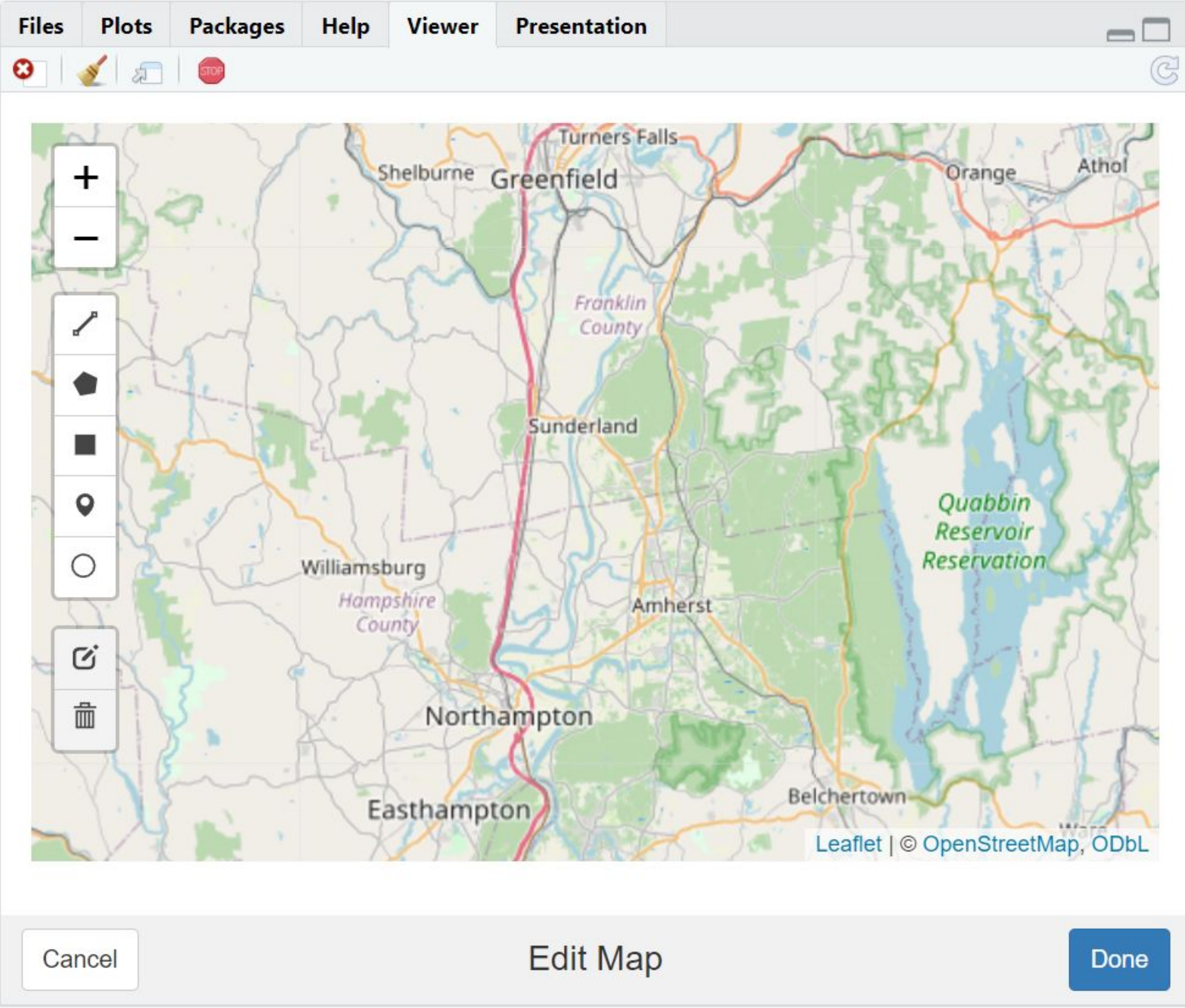
2. Leaflet & mapedit

Creating Vectors in R

- Leaflet is a free and open-source mapping project to make interactive maps
- mapedit is a package which leverages leaflet to create a mapping UI within R to create points, lines, and polygons.
- These are useful if you don’t want to open a new QGIS/ArcPro session just to create a vector.

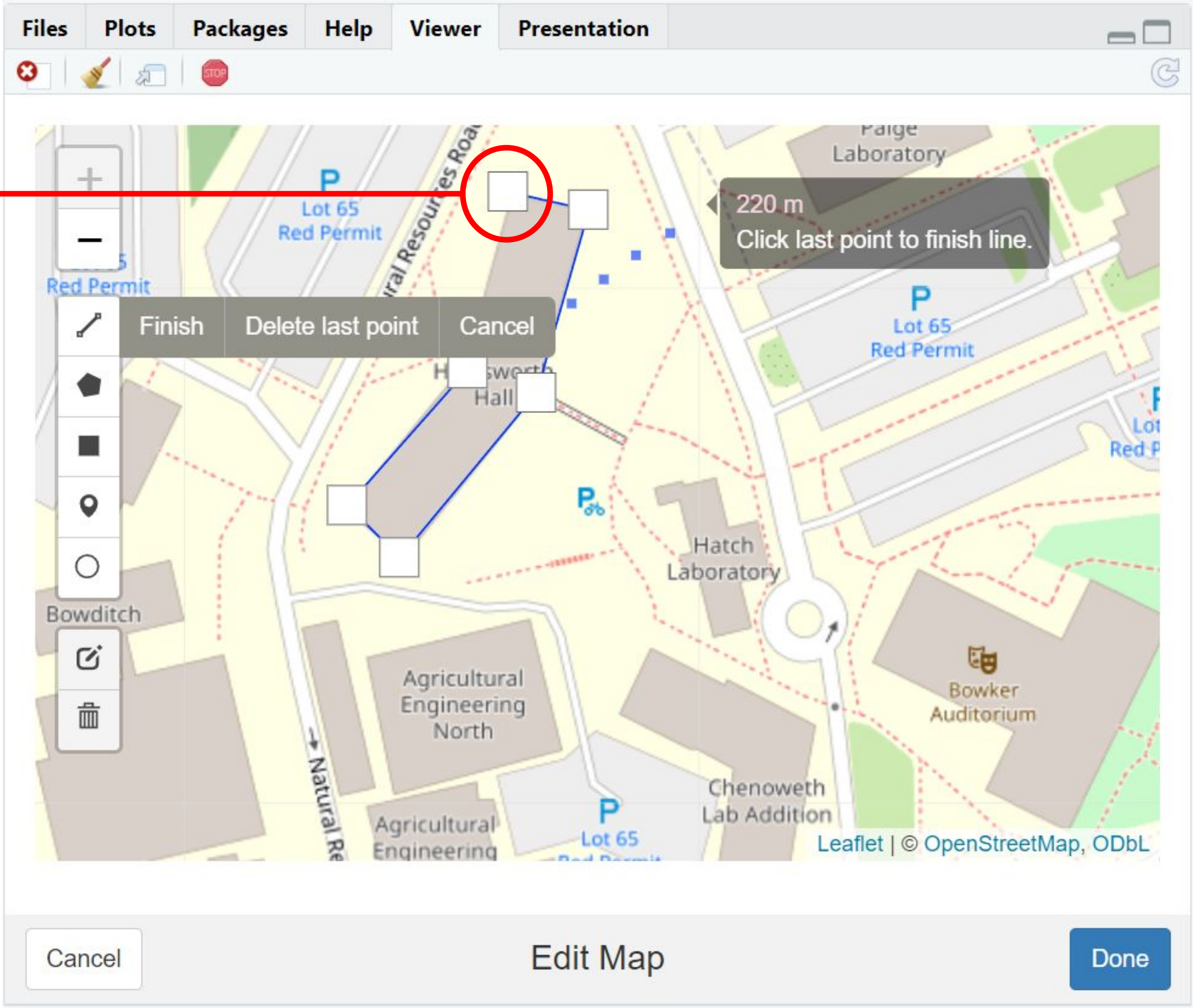

Files
Plots
Packages
Help
Viewer
Presentation
Leaflet | © OpenStreetMap, ODbL
Cancel
Edit Map
Done

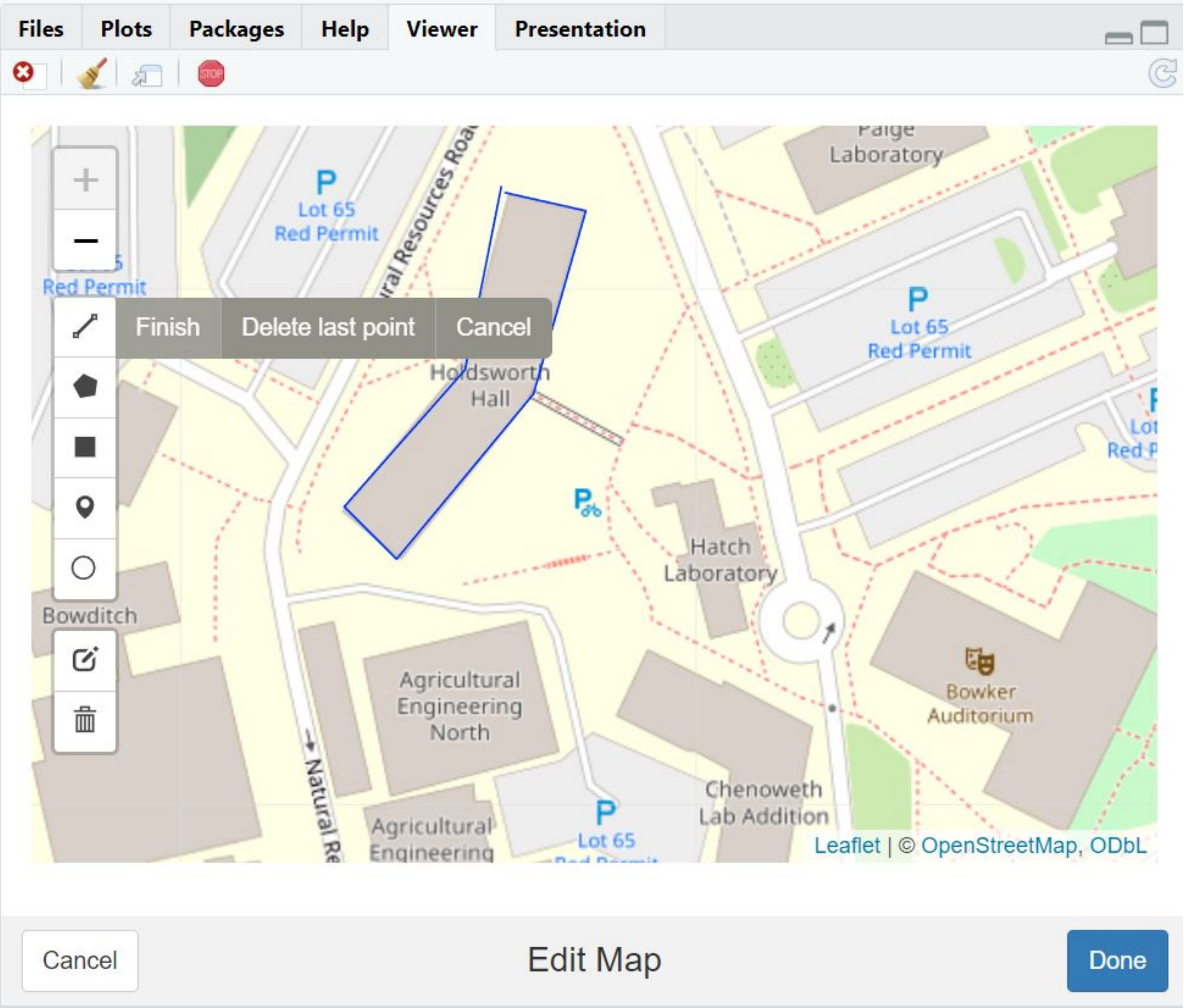
Line / Polyline
Polygon
Square
Polygon
Points
Circle

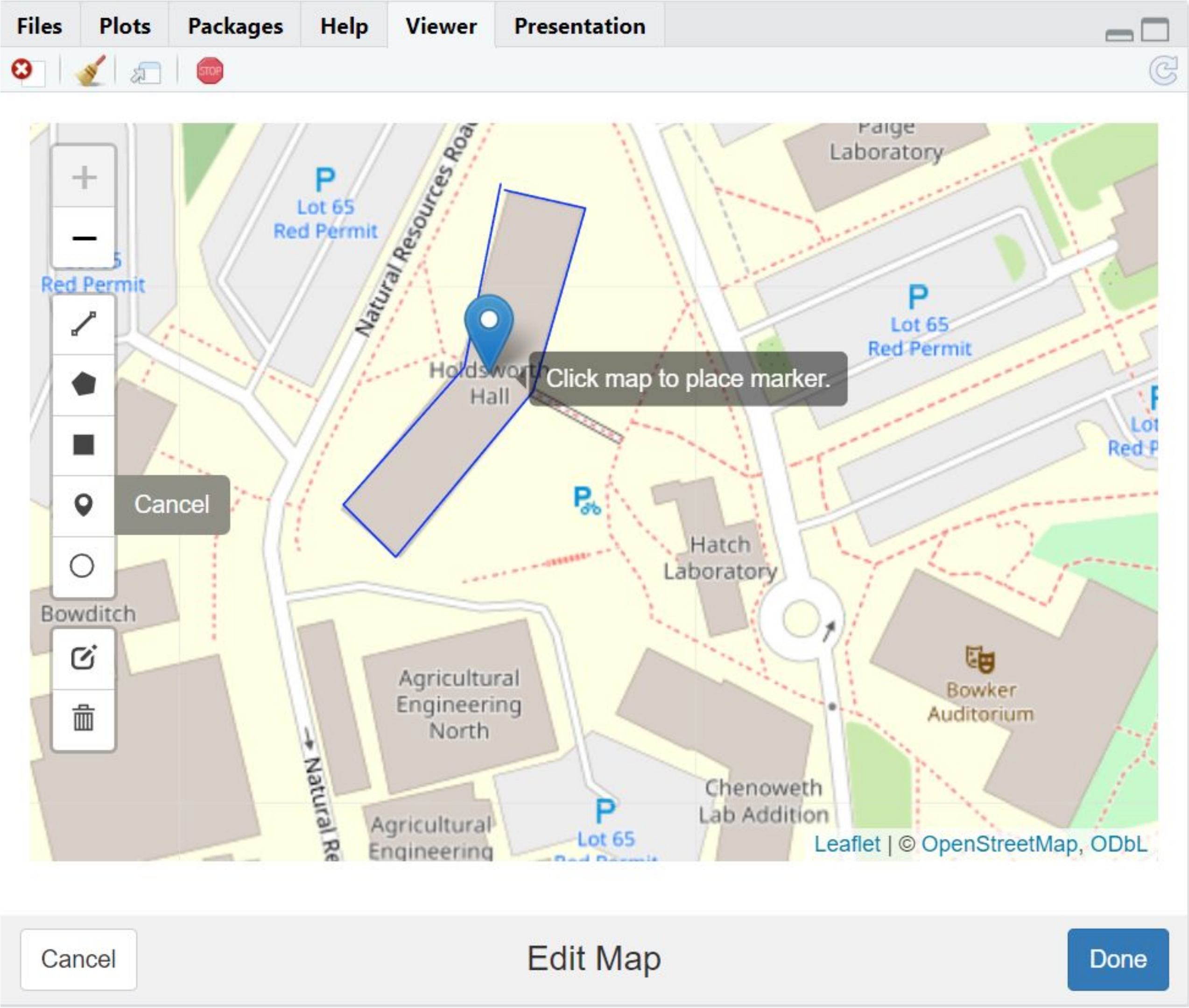


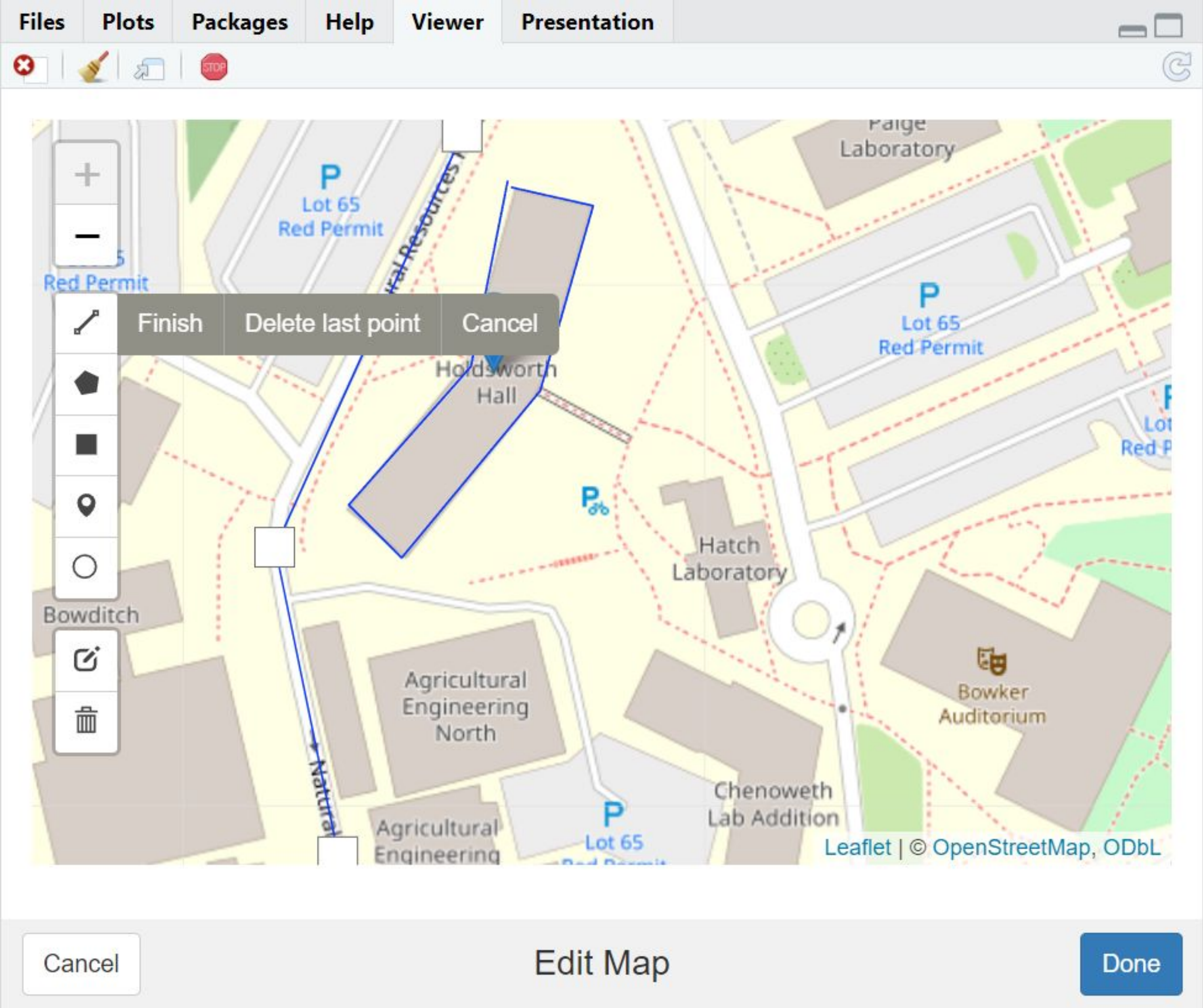
First point in the polygon.

Click to finish poly.




Files
Plots
Packages
Help
Viewer
Presentation
Paige Laboratory
P
Lot 65
Red Permit
Red Permit
Finish
Delete last point
Cancel
Holdsworth Hall
P
Lot 65
Red Permit
Hatch Laboratory
Bowditch
Agricultural Engineering North
Agricultural Engineering
P
Lot 65
Chenoweth Lab Addition
Bowker Auditorium
Leaflet | © OpenStreetMap, ODbL
Cancel
Edit Map
Done


Files
Plots
Packages
Help
Viewer
Presentation
Paige
Laboratory
P
Lot 65
Red Permit
Natural Resources Road
Red Permit
Holdsworth
Hall
Click map to place marker.
Lot 65
Red Permit
Cancel
Hatch
Laboratory
Bowditch
Agricultural
Engineering
North
Natural Re
Bowker
Auditorium
Chenoweth
Lab Addition
Agricultural
Engineering
Lot 65
Leaflet | © OpenStreetMap, ODbL
Cancel
Edit Map
Done


Files
Plots
Packages
Help
Viewer
Presentation
Lot 65
Red Permit
Finish
Delete last point
Cancel
Holdsworth
Hall
Paige
Laboratory
Hatch
Laboratory
Bowditch
Agricultural
Engineering
North
Agricultural
Engineering
Chenoweth
Lab Addition
Bowker
Auditorium
Leaflet | © OpenStreetMap, ODbL
Cancel
Edit Map
Done

How to setup mapedit!

```
# Prepare a leaflet map for mapedit
my_map = leaflet() %>% addTiles()

# Start a mapedit session
drawn_features = editMap(my_map)
```

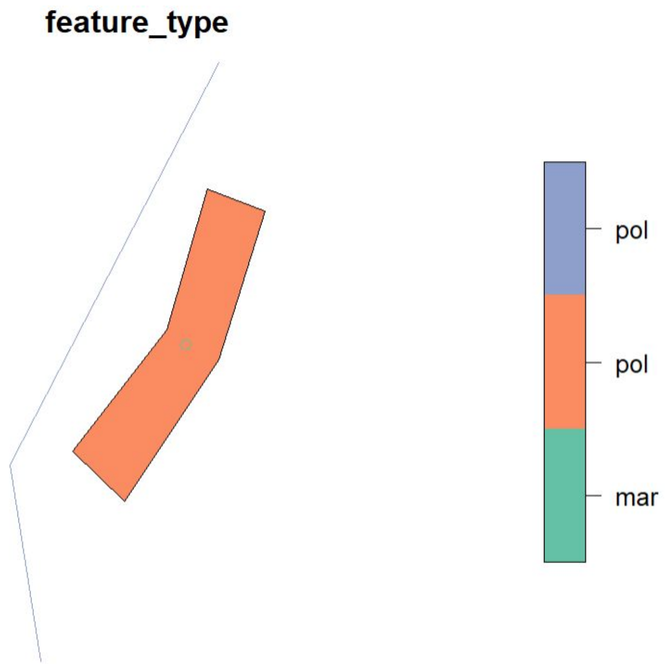
Working with mapedit output

Name	Type	Value
drawn_features	list [5]	List of length 5
drawn	list [3 x 3] (S3: sf, data.frame)	A data.frame with 3 rows and 3 columns
edited	NULL	Pairlist of length 0
deleted	NULL	Pairlist of length 0
finished	list [3 x 3] (S3: sf, data.frame)	A data.frame with 3 rows and 3 columns
all	list [3 x 3] (S3: sf, data.frame)	A data.frame with 3 rows and 3 columns

- The default mapedit output is a list, which is harder to work with in R
- But we can convert it to an sf dataframe!

Working with mapedit output cont.

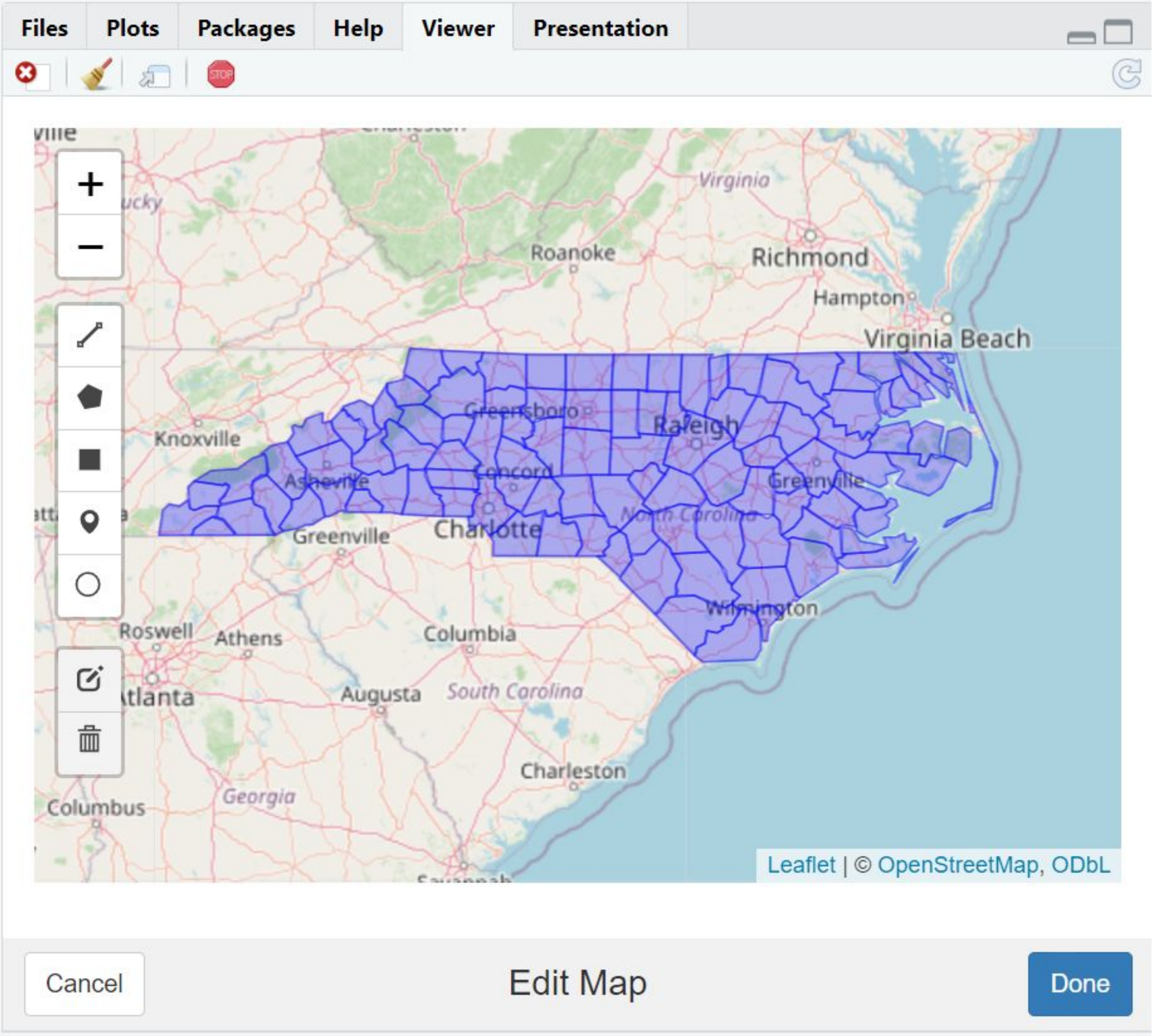
```
# Convert drawn_features list to an sf dataframe
drawn_features_sf = drawn_features$finished
plot(drawn_features_sf[2])
```



Leaflet Layers

- You can utilize your own layers by adding to the leaflet map
- Note: You will need to transform your layer to WGS84 in order to add it to the leaflet interactive map!

```
nc <- st_read(system.file("shape/nc.shp", package="sf"))
nc_wgs84 = st_transform(nc, crs = "EPSG:4326")
my_map <- leaflet() %>%
  addTiles() %>% # Add default OpenStreetMap tiles
  addPolygons(data = nc_wgs84,
              fillOpacity = 0.3, # Set fill opacity
              color = "blue", # Border color
              weight = 1) # Border thickness
drawn_features = editMap(my_map)
```


Files
Plots
Packages
Help
Viewer
Presentation
Virginia
Roanoke
Richmond
Hampton
Virginia Beach
Knoxville
Greensboro
Raleigh
Asheville
Concord
Greenville
North Carolina
Charlotte
Greenville
Wilmington
Roswell
Athens
Columbia
Atlanta
Augusta
South Carolina
Charleston
Columbus
Georgia
Leaflet | © OpenStreetMap, ODbL
Cancel
Edit Map
Done

3. Rasters in R w/ Terra

Terra Essentials - Cheat Sheet

- rast(): read in a raster
- writeRaster(): write a raster output
- app(): apply a function on a raster object
- global() : apply global statistics on a raster object
- mask(): mask a raster using another raster or polygon (essentially a clip)
- merge(): merge multiple rasters
- aggregate(): aggregate raster data to a lower resolution.
- disagg(): disaggregate raster data to a higher resolution.
- crs() + project(): get CRS and reproject rasters.

Reading Rasters

- rast()
- Example:
  - > **raster = rast(here::here('data', 'raster.tif'))**
- Note: R has a weird time retaining SpatRasters after you shutdown R. If you are getting empty plots or weird errors for your SpatRaster operations, just try reloading the SpatRaster.

Inspecting Rasters - Cheat Sheet

- summary() : returns general stats on the raster
- cellSize() : returns general info like dimensions of cells in raster
- ext() : returns extent of raster layer

Inspecting Rasters - summary(raster) Example

```
> summary(p80_1)
 wc2.1_10m_prec_1980.01.tif
 Min.   :  0.00
 1st Qu.:  7.43
 Median : 21.07
 Mean   : 53.18
 3rd Qu.: 56.37
 Max.   :930.36
 NA's   :75438
```

Inspecting Rasters - cellSize(raster)

- cellSize() will give you all the essential information on your raster
  - Dimensions: The number of rows, columns, and layers (bands) in the raster.
  - Resolution: The size of each cell in units of the coordinate reference system.
  - Spatial Extent
  - Coordinate Reference System (CRS)
  - Source: Information about the source of the raster data.
  - Basic statistics for each layer, such as the min, max, mean, and sd.
- This is useful, but knowing how to retrieve this information individually is better done with the specific commands like ext() or crs().

Inspecting Rasters - cellSize()

- cellSize() returns the size of each cell in your Raster

```
> cellSize(p80_1)
class       : SpatRaster
dimensions  : 1080, 2160, 1  (nrow, ncol, nlyr)
resolution  : 0.1666667, 0.1666667  (x, y)
extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
coord. ref. : lon/lat WGS 84 (EPSG:4326)
source(s)   : memory
varname     : wc2.1_10m_prec_1980-01
name        :        area
min value   :    504025.1
max value   : 341918442.3
```

Inspecting Rasters - ext()

```
> ext(p80_1)
SpatExtent : -180, 180, -90, 90 (xmin, xmax, ymin, ymax)
```

- This is an alternative to cellSize() if you only wanted the extent

Terra Essentials - Cheat Sheet

- rast(): read in a raster
- writeRaster(): write a raster output
- app(): apply a function on a raster object
- global() : apply global statistics on a raster object
- mask(): mask a raster using another raster or polygon (essentially a clip)
- merge(): merge multiple rasters
- aggregate(): aggregate raster data to a lower resolution.
- disagg(): disaggregate raster data to a higher resolution.
- crs() + project(): get CRS and reproject rasters.

Raster Calculations - global()

- global() can perform a variety of summary operations.
- Example: Let's say you want to calculate the mean of all the raster values.
  - > mean_raster = **global(raster, fun="mean")**
- fun specifies the function you want to apply.
  - "mean"
  - "sum"
  - "min" / "max"
  - "range"
  - "prod"
  - etc.

Raster Calculations - Global Stats

```
# Example: Get mean of global values in a raster
mean_prec_jan = global(p80_1, fun = mean, na.rm = TRUE)
print(mean_prec_jan)
```

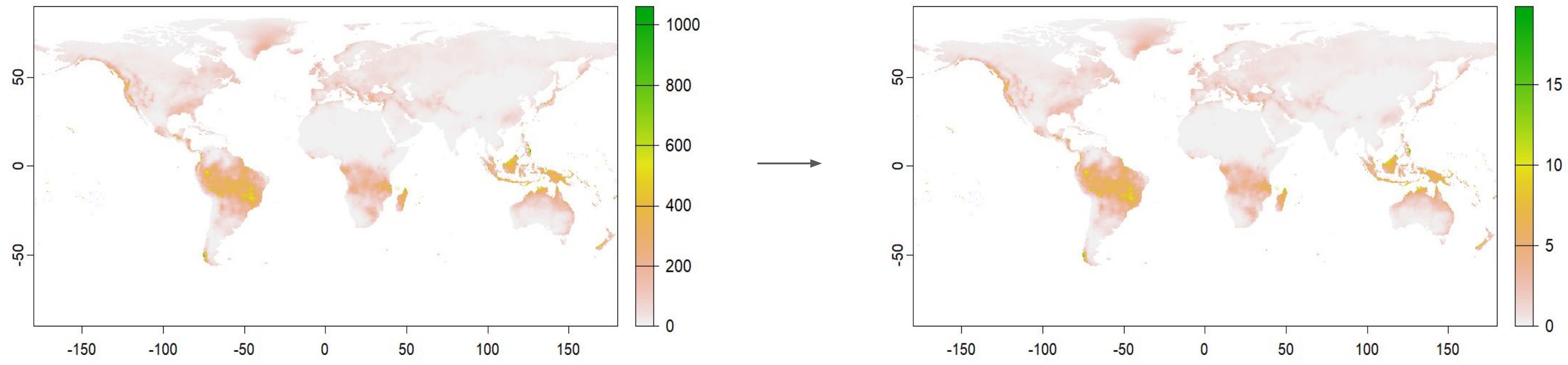
```
                              mean
wc2.1_10m_prec_1980-01.tif 53.3882
```

Raster Calculations - app()

- app()
- Example:
  - **> app_calc = app(raster, fun = FUNCTION, na.rm=TRUE)**
- Note: app() can leverage parallel processing!

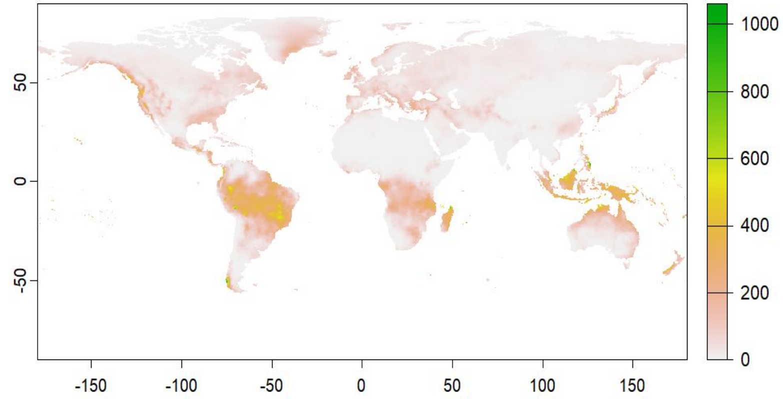
Raster Calculations - Single Raster Example

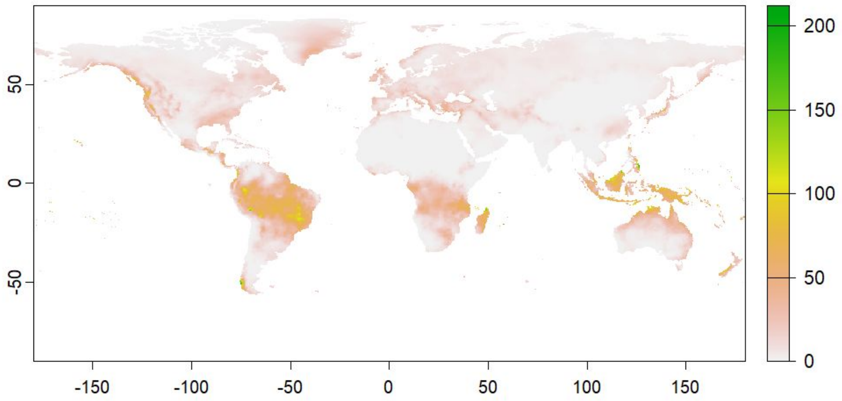
```
# Example: Simple Raster Arithmetic
p80_1_stand = p80_1 / mean_prec_jan[1,1]
plot(p80_1_stand)
```



Raster Calculations - Single Raster Custom Example

```
# Example: Custom Raster Calculation Functions with app()
p80_1_customfun = app(p80_1, fun = function(x) { x / 10 * 2})
plot(p80_1_customfun)
```



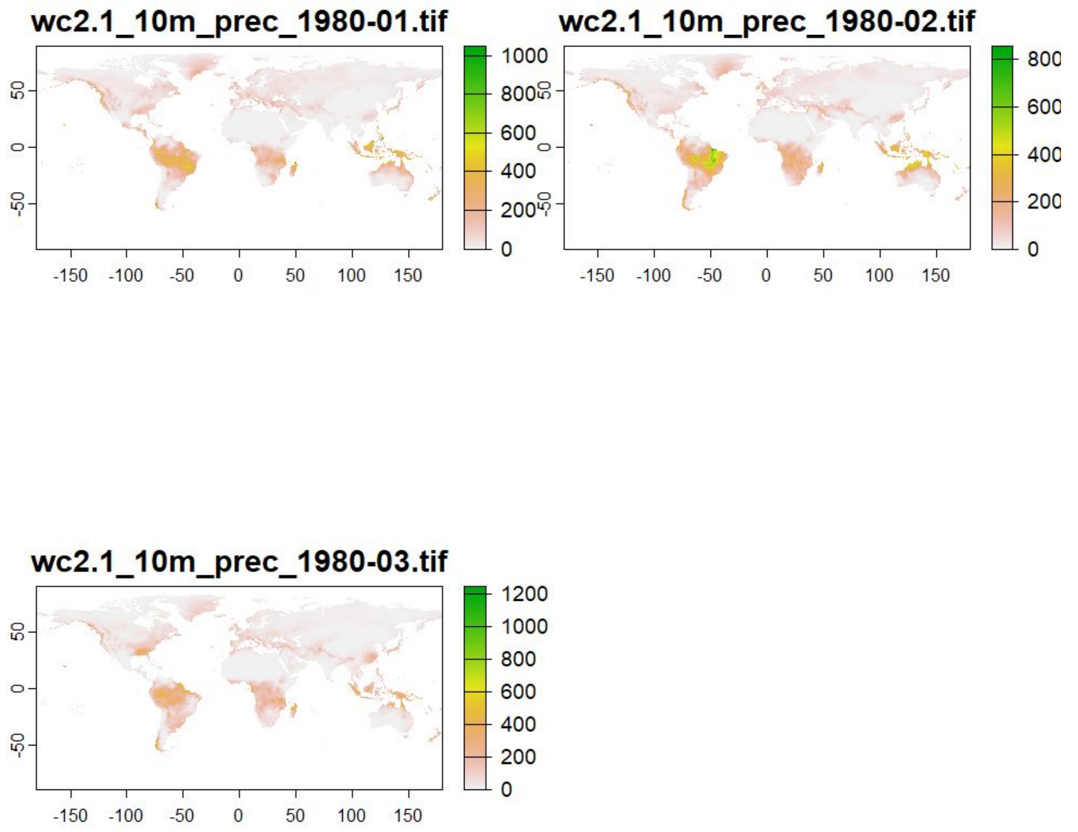


Raster Stacks

- A raster stack is a SpatRaster that contains multiple rasters in it!
- Useful for multilayer spatial analysis and calculations.
- Some packages expect a raster stack as an input.
- Example:
    - > raster1 = rast(here::here('data', 'X.tif')
    - > raster2 = rast(here::here('data', 'Y.tif')
    - > raster3 = rast(here::here('data', 'Z.tif')
    - **> raster_stack = c(raster1, raster2, raster3)**

Raster Stacks - Plotting

- Plotting a raster stack will typically plot all the rasters within the stack individually.



Raster Stack Calculations - app()

- app()
  - Purpose: Designed to apply a function **to each layer of a raster stack individually**, or to each cell of a single layer.
  - Custom and Complex Operations: It's particularly useful for applying custom functions that are not inherently vectorized or when the operation depends on other factors (like conditional operations based on cell values).
    - Good for complex functions that involve conditional logic and looping etc.
  - Output depends on the function being applied.
    - It can be a raster layer, a raster stack, or even non-raster outputs if the function is designed that way.

Raster Stack Calculations - Example

- app(raster_stack, fun = function)
  - # Define a custom function to convert from Fahrenheit to Celsius
  - > convert_to_celsius <- function(fahrenheit) {
  - celsius <- (fahrenheit - 32) * 5/9
  - return(celsius)
  - }
  - # Apply the function to each layer
  - > adjusted_temp_stack <- **app(raster_stack, fun=convert_to_celsius)**
  - # Output will be a single raster stack
  - # but with values of each layer converted to Celsius

aggregate()

- Used for summarizing or aggregating data based on certain criteria, such as finding averages, sums, or counts within grouped data.
- Basic Syntax: aggregate(x, by, FUN)
  - x: The data to be aggregated.
  - by: A list of grouping variables.
    - The aggregation is performed separately for each combination of these variables.
  - FUN: The function to be applied for aggregation, such as mean, sum, max, etc.

aggregate() Example

```
> head(state.x77)
           Population Income Illiteracy Life Exp Murder HS Grad Frost   Area
Alabama          3615   3624        2.1    69.05   15.1    41.3    20  50708
Alaska            365   6315        1.5    69.31   11.3    66.7   152 566432
Arizona          2212   4530        1.8    70.55    7.8    58.1    15 113417
Arkansas         2110   3378        1.9    70.66   10.1    39.9    65  51945
California      21198   5114        1.1    71.71   10.3    62.6    20 156361
Colorado         2541   4884        0.7    72.06    6.8    63.9   166 103766
```

```
# Compute the averages for the variables in 'state.x77', grouped
# according to the region (Northeast, South, North Central, West) that
# each state belongs to.
aggregate(state.x77, list(Region = state.region), mean)
```

Region	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Northeast	5495.111	4570.222	1.000000	71.26444	4.722222	53.96667	132.7778	18141.00
South	4208.125	4011.938	1.737500	69.70625	10.581250	44.34375	64.6250	54605.12
North Central	4803.000	4611.083	0.700000	71.76667	5.275000	54.51667	138.8333	62652.00
West	2915.308	4702.615	1.023077	71.23462	7.215385	62.00000	102.1538	134463.00

Scaling Rasters - terra::aggregate() & disagg()

- Changing raster resolutions can be the difference between your code running or not!
- Hardware limitations are very real in science, and limits on RAM are especially troublesome when doing raster analytics.
    - RAM = Random Access Memory: Temporary, fast computer memory
- aggregate() downsamples your data to a more coarse resolution
- disagg() upsamples your data to a finer resolution

aggregate() & disagg() Examples

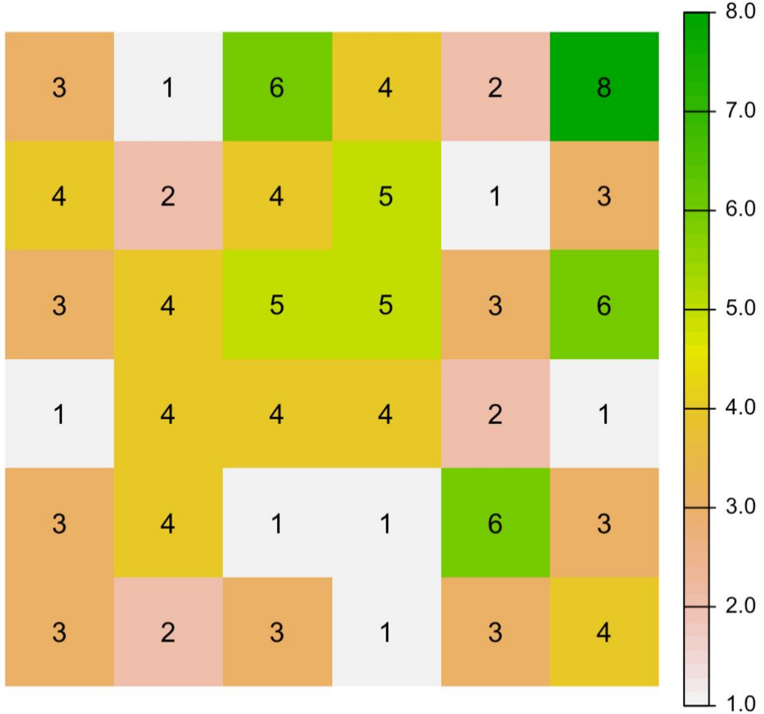
aggregate()

- > # Downsample your raster by a factor of 2
- > terra::**aggregate(raster, fact=2)**
- > # A 100x100 raster would downsample to 50x50 cells

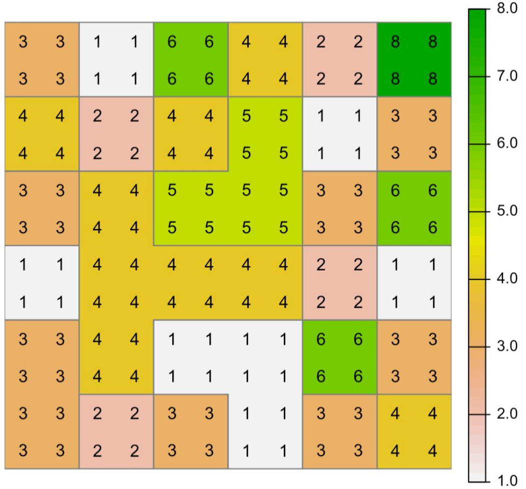
disagg()

- > # Upsample your raster by a factor of 2
- > **disagg(raster, fact=2, method=”near”)**
- > # A 100x100 raster would upsample to 200x200 cells

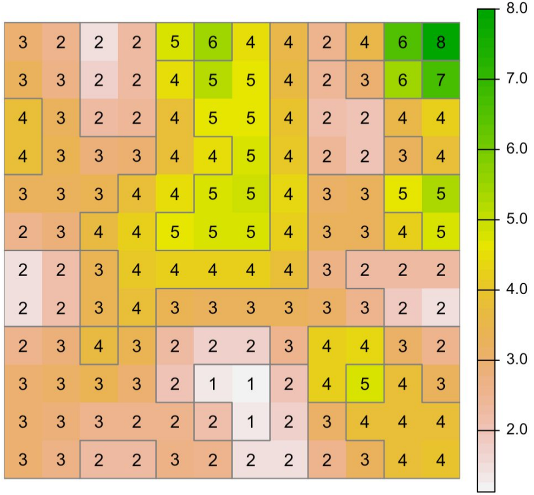
disagg() - Raster Upsample



Nearest



Bilinear Interpolation



write.Raster()

- Used to write your raster to the format of your liking
- Input the SpatRaster and the name of the output you want
    - Note: Must specify the data type you want to output as in the name!
- Example:
    - write.raster(raster, “name_of_raster.tif”)
    - write.raster(raster, “name_of_raster.ascii”)

Projecting Rasters - Example

```
p80_1_wgs84 = project(p80_1, "EPSG:4326",
                      method = 'bilinear')
```

- methods
  - near: nearest neighbor. This method is fast, but not a good choice for continuous values.
  - bilinear: bilinear interpolation
  - cubic: cubic interpolation.
  - cubicspline: cubic spline interpolation.
  - lanczos: Lanczos windowed sinc resampling.

Other useful Terra Functions

- contour() : Generate contours for your SpatRaster
- terrain() : Terrain analytics for your SpatRaster
- Terra can handle vectors as well (SpatVectors)
  - interpIDW() : Inverse Distance Weighting Interpolation
  - interpNear() : Nearest Neighbor Interpolation
  - interpolation() : General Interpolation Function
  - cartogram(): Map where the area of polygons is made proportional to another variable.