Spatial Data Operations

Ivan Lopez

2/28/2022

packages

```r
library(sf)       # vector data package
## Linking to GEOS 3.6.2, GDAL 2.2.3, PROJ 4.9.3; sf_use_s2() is TRUE
library(terra)    # raster data package
## terra 1.5.21
library(dplyr)    # tidyverse package for data frame manipulation
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:terra':
##
##     intersect, src, union
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
library(spData)  # loads datasets used here
## Warning: multiple methods tables found for 'direction'
## Warning: multiple methods tables found for 'gridDistance'
library(here)
## here() starts at /home/ilopezcr/Project/Practices/class_chap_4
```

preamble

- We will use information from the last lecture.

```
elev = rast(system.file("raster/elev.tif", package = "spData"))
grain = rast(system.file("raster/grain.tif", package = "spData"))
```

introduction

- Spatial joins between vector datasets and local and focal operations on raster datasets
- **Goal:** modify geometries based on their location and shape.
- There is a link between attribute operations and spatial ones:
  - spatial subsetting: select rows based on **geom.**
  - spatial joining: combine tables based on **geom.**
  - aggregation: group observation based on **geom.**

introduction

- Spatial joins, for example, can be done in a number of ways:
  - matching entities that intersect with or are close enough to the target spot.
- To explore the spatial relationships (contained, overlaps, etc.) between obkects:
  - use functions (**topological relations**) on sf objects.
- Distances: all spatial objects are related through space.
  - Distance calculations can be used to explore the strength of this relationship.

introduction

- Spatial operations on raster objects include subsetting and merging several raster ‘tiles’ into a single object.
- Map algebra covers a range of operations that modify raster cell values, with or without reference to surrounding cell values
  - vital for many applications.
- We will also compute distances within rasters.
- Note that to apply any function on two spatial objects, the latter most share the same CRS!

Vector data: subsetting

- **Goal:** reshape an existing object in reference to another object.
- Subsets of `sf` data frames can be created with **square bracket ([) operator**.
  - Syntax `x[y, , op = st_intersects]`, where `x` is an `sf` object from which a subset of rows will be returned.
  - `y` is the 'subsetting object' `op = st_intersects` specifies the topological relation to do the subsetting.
- The **default** topological relation is st_intersects()
  - the command x

    $$y,$$

    is identical to x

    $$y,,op = st_intersects$$
- The `filter()` function from the `tidyverse` can also be used.

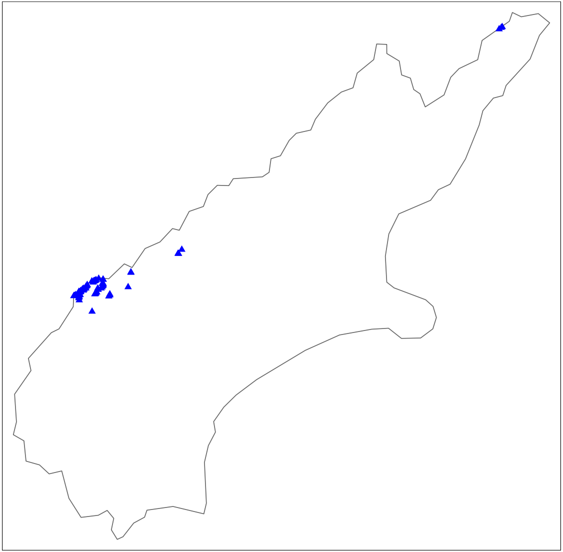
Vector data: subsetting

- Demonstration: `nz` and `nz_height` datasets.
  - contain geographic data on the 16 main regions and 101 highest points in New Zealand (projected CRS).
- Create an object representing Canterbury and return all high points in the region:

```
# filter out Canterbury
canterbury = nz %>% filter(Name == "Canterbury")
# subset the high points that "intersect" the above.
(canterbury_height = nz_height[canterbury, ])
## Simple feature collection with 70 features and 2 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 1365809 ymin: 5158491 xmax: 1654899 ymax: 5350463
## CRS:           EPSG:2193
## First 10 features:
##    t50_fid elevation                geometry
## 5  2362630      2749 POINT (1378170 5158491)
## 6  2362814      2822 POINT (1389460 5168749)
## 7  2362817      2778 POINT (1390166 5169466)
## 8  2363991      3004 POINT (1372357 5172729)
## 9  2363993      3114 POINT (1372062 5173236)
## 10 2363994      2882 POINT (1372810 5173419)
## 11 2363995      2796 POINT (1372579 5173989)
## 13 2363997      3070 POINT (1373796 5174144)
## 14 2363998      3061 POINT (1373955 5174231)
## 15 2363999      3077 POINT (1373984 5175228)
```

Vector data: subsetting

```
tmap::tm_shape(canterbury) + tmap::tm_borders() +
tmap::tm_shape(canterbury_height) + tmap::tm_symbols(shape = 17, col = "blue", size = .2)
```



Vector data: subsetting

- The command `x[y, ]` subsets features of a **target** `x` w.r.t. object `y`.
- Both `x` and `y` must be geographic objects (`sf`).
- Various topological relations for subsetting:
  - `touches`, `crosses` or `within` (among others).
- `st_intersects` is a ‘catch all’ instruction
  - catches everything that touches, crosses or falls within the source ‘subsetting’ object
- Alternative spatial operators: write desired op = argument.
  - the opposite to st_intersects:
  - `nz_height[canterbury, , op = st_disjoint]`
- plot the map of New Zealand and the high points outside Canterbury.

Vector data: subsetting

- Note the empty argument — denoted with , , — is included to highlight **op**, the third argument in **[** for `sf` objects.
- The second argument may change the subsetting operation:
  - `nz_height[canterbury, 2, op = st_disjoint]`
- The above returns the same rows but only includes the second attribute column.

Vector data: subsetting

topological operators outputs - They return objects that can be used for subsetting. - In the below code, we create an object with `(empty)` and `1`. - `empty` indicates no intersection between the target object and the subsetting object. - it is an empty vector with length zero. - Then we transform the latter into a logical vector. - Finally we conduct the subsetting operation.

Vector data: subsetting

```
# intersect heights and Canterbury
sel_sgbp = st_intersects(x = nz_height, y = canterbury)

class(sel_sgbp)
## [1] "sgbp" "list"

sel_sgbp
## Sparse geometry binary predicate list of length 101, whe
## predicate was `intersects'
## first 10 elements:
##  1: (empty)
##  2: (empty)
##  3: (empty)
##  4: (empty)
##  5: 1
##  6: 1
##  7: 1
##  8: 1
```

Vector data: subsetting

- One can repurpose the above operation.
    - For instance: keep those elements that intersect with more than one element in the subsetting object.
- `st_filter`: similar to the standard `dplyr`.

```
canterbury_height3 = nz_height %>%
  st_filter(y = canterbury, .predicate = st_intersects)
```

Vector data: spatial relations

- Sometimes it is important to establish whether two objects are spatially related.
  - **Topological relations**: pindown the existence of a spatial relation.
- Symmetric operators:

1. equals
2. intersects
3. crosses
4. touches
5. overlaps

- Asymmetric operators:

1. contains
2. within

Vector data: spatial relations

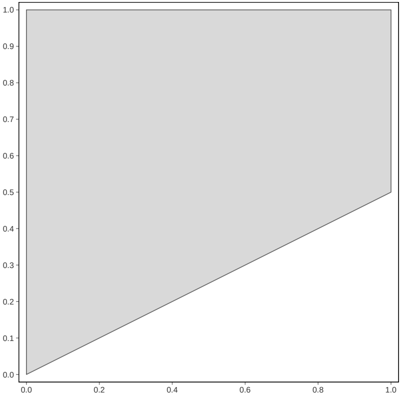
visualization

Vector data: spatial relations

- Let's create an example.
- First, we create a polygon: use `cbind` to generate a matrix of vertices.
- use `st_sfc` and `st_polygon` to create an `sf`.
- we will create a line and group of points.
- we will visually examine the spatial relationships.
- Finally, we will use the operators (binary predicates) to corroborate our visual inspection.

Vector data: spatial relations

```
polygon_matrix = cbind(
  x = c(0, 0, 1, 1,   0),
  y = c(0, 1, 1, 0.5, 0)
)
polygon_sfc = st_sfc(st_polygon(list(polygon_matrix)))

tmap::tm_shape(polygon_sfc) + tmap::tm_polygons() + tmap::tm_grid(lines = FALSE)
## Warning: Currect projection of shape polygon_sfc unknown. Long-lat (WGS84) is
## assumed.
```

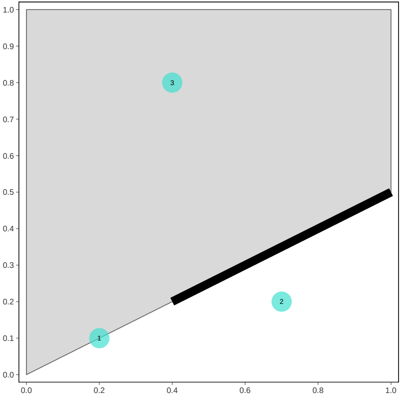


Vector data: spatial relations

```
line_matrix = cbind(
  x = c(0.4, 1),
  y = c(0.2, 0.5))

line_sfc = st_sfc(st_linestring(line_matrix))

# create a data frame of points
(point_df = data.frame(
  x = c(0.2, 0.7, 0.4),
  y = c(0.1, 0.2, 0.8)
))
##     x   y
## 1 0.2 0.1
## 2 0.7 0.2
## 3 0.4 0.8
point_sf = st_as_sf(point_df, coords = c("x", "y")) %>%
  tibble::rowid_to_column("ID") %>% mutate(ID=as.character(
```

Vector data: spatial relations

```
oldw <- getOption("warn")
options(warn = -1)

tmap::tm_shape(polygon_sfc) + tmap::tm_polygons() +
  tmap::tm_shape(line_sfc) + tmap::tm_lines(scale = 10) +
  tmap::tm_shape(point_sf) +
  tmap::tm_dots(scale=5, legend.show = F,col = "turquoise", alpha = .7, size= .1) +
  tmap::tm_text("ID", size = .5) +
  tmap::tm_grid(lines = FALSE)
```




```
options(warn = oldw)
```

Vector data: spatial relations

- Let's conduct a simple **query**.
- Which of the points in `point_sf` intersect in some way with `polygon_sfc`?
- This question can be answered with the spatial predicate st_intersects() as follows:

```
# The code below sets sparse=FALSE to coerce the output
# into a logical vector, instead of a sparse matrix.

st_intersects(point_sf, polygon_sfc, sparse = FALSE)
##       [,1]
## [1,]  TRUE
## [2,] FALSE
## [3,]  TRUE

# A sparse matrix is a list of vectors with
# empty elements where a match doe not exists.
```

Vector data: spatial relations

- Which points lie within the polygon?
- Which features are on or contain a shared boundary with y?
- These can be answered as follows:

```
st_within(point_sf, polygon_sfc)
## Sparse geometry binary predicate list of length 3, where
## was `within'
##  1: (empty)
##  2: (empty)
##  3: 1
st_touches(point_sf, polygon_sfc)
## Sparse geometry binary predicate list of length 3, where
## was `touches'
##  1: 1
##  2: (empty)
##  3: (empty)
```

Vector data: spatial relations

- The opposite of `st_intersects()` is `st_disjoint()`, which returns only objects that do not spatially relate in any way to the selecting object

```
# note [, 1] converts the result into a vector:

st_disjoint(point_sf, polygon_sfc, sparse = FALSE)[, 1]
## [1] FALSE  TRUE FALSE
```

Vector data: spatial relations

- `st_is_within_distance()` detects features within a distance from the target.
- It can be used to set how close target objects need to be before they are selected.
  - recall the hydrocarbon processing plants!
- Although **point 2** is more than 0.2 units of distance from the nearest vertex of **polygon_sfc**, it is **still selected** when the distance is set to 0.2.
- This is because distance is measured to the **nearest edge**,
  - In this case the part of the the polygon that lies directly above **point 2**.
  - Verify the actual distance between **point 2** and the polygon is 0.13 with the command `st_distance(point_sf, polygon_sfc)`.

Vector data: spatial relations

- The **'is within distance'** binary spatial predicate is demonstrated in the code chunk below,
- Indeed, every point is within 0.2 units of the polygon:

```
st_is_within_distance(point_sf, polygon_sfc,
                            dist = 0.2, sparse = FALSE)[, 1]
## [1] TRUE TRUE TRUE
```

Vector data: spatial joining

- Joining two non-spatial datasets relies on a shared ‘key’ variable
- Spatial data joining applies the same concept drawing on spatial relations
- Joining adds new columns to the target object **x**, from a source object **y**.
- *Example*:
  - ten points randomly distributed across the Earth’s surface
  - for the points that are on land, which countries are they in?
  - Implementing this idea in a reproducible example will build your geographic data handling skills and show how spatial joins work.

Vector data: spatial joining

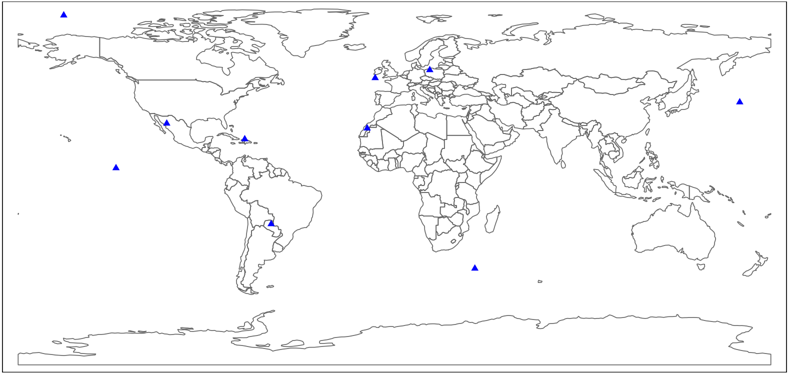
1. Establish the bbox for the analysis: "the entire globe"
2. Create points that are randomly scattered over the Earth's surface. Use the r's uniform distribution, and make sure the values fall into the bbox
3. Set the points as an sf object.

```
set.seed(2018) # set seed for reproducibility
(bb = st_bbox(world)) # the world's bounds
##      xmin      ymin      xmax      ymax
## -180.00000  -89.90000  179.99999   83.64513
random_df = data.frame(
  x = runif(n = 10, min = bb[1], max = bb[3]),
  y = runif(n = 10, min = bb[2], max = bb[4])
)
random_points = random_df %>%
  st_as_sf(coords = c("x", "y")) %>% # set coordinates
  st_set_crs("EPSG:4326") # set geographic CRS
```

Vector data: spatial joining

4. Now, plot the points on an earth's map.

```
st_crs(world) <- 4326
## Warning: st_crs<- : replacing crs does not reproject data; use st_transform for
## that

tmap::tm_shape(world) + tmap::tm_borders() +
  tmap::tm_shape(random_points) + tmap::tm_symbols(shape = 17, col = "blue", size = .2)
```



Vector data: spatial joining

- The object `world_random`yields only countries that contain random points
  - we will obtain it again via a spatial join.

```
# find the countries "touched" by random points
(world_random = world[random_points,])
## Simple feature collection with 4 features and 10 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -117.1278 ymin: -27.5485 xmax: 24.02999 ymax: 54.85154
## CRS:           EPSG:4326
## # A tibble: 4 x 11
##   iso_a2 name_long continent   region_un subregion type  area_km2    pop lifeExp
##   <chr>  <chr>     <chr>       <chr>     <chr>     <chr>    <dbl>  <dbl>   <dbl>
## 1 MX     Mexico    North Amer~ Americas  Central ~ Sove~ 1969480. 1.24e8    76.8
## 2 PL     Poland    Europe      Europe    Eastern ~ Sove~  310402. 3.80e7    77.6
## 3 PY     Paraguay  South Amer~ Americas  South Am~ Sove~  401336. 6.55e6    72.9
## 4 MA     Morocco   Africa      Africa    Northern~ Sove~  591719. 3.43e7    75.3
## # ... with 2 more variables: gdpPercap <dbl>, geom <MULTIPOLYGON [°]>
```

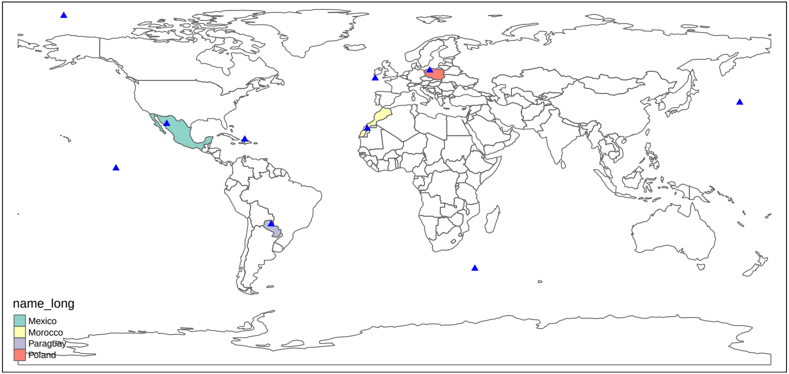
Vector data: spatial join

- `st_join` is the key function here.

```
# find the points that touch a country.
(random_joined =
   st_join(random_points, select(world,name_long),
           join = st_intersects))
## Simple feature collection with 10 features and 1 field
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -158.1893 ymin: -42.91501 xmax: 165.1157 ymax: 80.5408
## CRS:           EPSG:4326
##    name_long                    geometry
## 1   Paraguay POINT (-58.98475 -21.24278)
## 2    Morocco  POINT (-13.05963 25.42744)
## 3       <NA>   POINT (-158.1893 80.5408)
## 4     Mexico  POINT (-108.9239 27.80098)
## 5       <NA>   POINT (-9.246895 49.9822)
## 6       <NA>  POINT (-71.62251 20.15883)
## 7       <NA>  POINT (38.43318 -42.91501)
## 8       <NA>  POINT (-133.1956 6.053818)
## 9       <NA>   POINT (165.1157 38.16862)
## 10    Poland   POINT (16.86581 53.86485)
```

Vector data: spatial join

```
tmap::tm_shape(world)+tmap::tm_borders() +
  tmap::tm_shape(world_random) + tmap::tm_polygons("name_long") +
  tmap::tm_shape(random_points) + tmap::tm_symbols(shape = 17, col = "blue", size = .2)
```



Vector data: spatial join

- By default, st_join() performs a left join
- all rows from x including rows with no match in y.
- It can also do inner joins
  - set the argument left = FALSE.
- The default topological operator used by st_join() is st_intersects()
- The example above demonstrates the addition of a column from a polygon layer to a point layer same approach works regardless of geometry types.
  - In such cases, for example when x contains polygons, each of which match multiple objects in y, spatial joins will result in duplicate features, creates a new row for each match in y (**see the homework**).

Non-overlapping joins

- Sometimes two geographic datasets do not touch but still have a strong geographic relationship.
- The datasets `cycle_hire` and `cycle_hire_osm` provide a good example.
- Plotting them shows that they are often closely related but they do not touch.

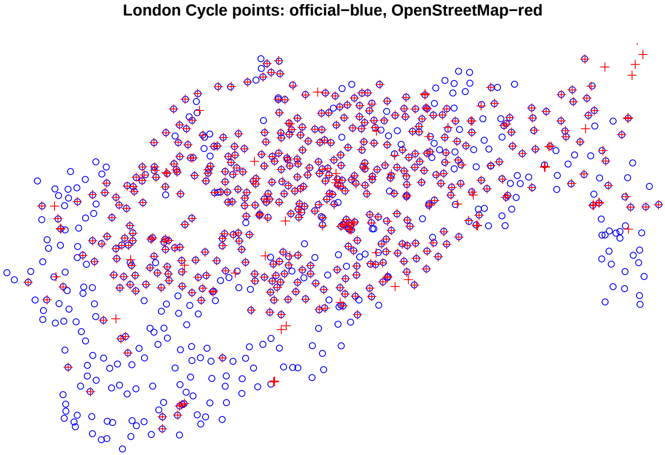
Non-overlapping joins

London bike hire key information

- You can hire bikes using London's public cycle hire scheme, Santander Cycles.
- Riders will find 800 docking stations and 12,000 bikes to hire around London.
- Bikes can be hired using a bank card at the docking station or using the official Santander Cycles app.

Non-overlapping joins

```
plot(st_geometry(cycle_hire), col = "blue", main = "London Cycle points: official-blue, OpenStreetMap-red"
plot(st_geometry(cycle_hire_osm), add = TRUE, pch = 3, col = "red")
```



Non-overlapping joins

- We can check if any points are the same:
  - any: given a set of logical vectors, is at least one of the values true?

```
st_touches(cycle_hire, cycle_hire_osm, sparse = FALSE) %>%
any()
## [1] FALSE
```

Non-overlapping joins

- Imagine that we need to join the capacity variable in `cycle_hire_osm` onto the official 'target' data contained in `cycle_hire`.
- This is when a non-overlapping join is needed.
- The simplest method is to use the topological operator st_is_within_distance()
  - use a threshold distance of 20 m.
  - that is, assume that if two points, belonging each to a different dataset, are close enough, then they speak about the same spot.

Non-overlapping joins

```
head(cycle_hire)
## Simple feature collection with 6 features and 5 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -0.1975742 ymin: 51.49313 xmax: -0.08460569 ymax: 51.53006
## CRS:           EPSG:4326
##   id                name              area nbikes nempty
## 1  1        River Street      Clerkenwell      4     14
## 2  2 Phillimore Gardens       Kensington      2     34
## 3  3 Christopher Street Liverpool Street      0     32
## 4  4  St. Chad's Street     King's Cross      4     19
## 5  5     Sedding Street    Sloane Square     15     12
## 6  6 Broadcasting House       Marylebone      0     18
##                       geometry
## 1  POINT (-0.1099705 51.52916)
## 2  POINT (-0.1975742 51.49961)
## 3 POINT (-0.08460569 51.52128)
## 4  POINT (-0.1209737 51.53006)
## 5   POINT (-0.156876 51.49313)
## 6  POINT (-0.1442289 51.51812)
```

Non-overlapping joins

```
head(cycle_hire_osm)
## Simple feature collection with 6 features and 5 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -0.1293092 ymin: 51.52583 xmax: -0.090836 ymax: 51.53402
## CRS:           EPSG:4326
##      osm_id                        name capacity cyclestreets_id description
## 1    108539            Windsor Terrace       14            <NA>        <NA>
## 2 598093293 Pancras Road, King's Cross       NA            <NA>        <NA>
## 3 772536185 Clerkenwell, Ampton Street       11            <NA>        <NA>
## 4 772541878                       <NA>       NA            <NA>        <NA>
## 5 781506147                       <NA>       NA            <NA>        <NA>
## 6 783824668      Finsbury Library, EC1       NA            <NA>        <NA>
##                     geometry
## 1 POINT (-0.0933878 51.52913)
## 2 POINT (-0.1293092 51.53402)
## 3 POINT (-0.1182352 51.52729)
## 4  POINT (-0.090836 51.52583)
## 5 POINT (-0.1210572 51.53001)
## 6 POINT (-0.1038272 51.52594)
```

Non-overlapping joins

```
(sel = st_is_within_distance(cycle_hire, cycle_hire_osm, dist = 20))
## Sparse geometry binary predicate list of length 742, where the
## predicate was `is_within_distance'
## first 10 elements:
##  1: 233
##  2: 278
##  3: 294
##  4: 5
##  5: (empty)
##  6: 59
##  7: 68
##  8: 63
##  9: 23
##  10: 100
```

Non-overlapping joins

- The code below tells us that there are 438 points in the target object `cycle_hire` within the threshold distance of `cycle_hire_osm`

```
summary(lengths(sel) > 0)
##    Mode   FALSE    TRUE
## logical     304     438
```

Non-overlapping joins

- How to retrieve the values associated with the respective `cycle_hire_osm` points?
- The solution is again with `st_join()`.

```
aux = st_join(cycle_hire, select(cycle_hire_osm,capacity), join = st_is_within_distance,
          dist = 20)
nrow(cycle_hire)
## [1] 742
nrow(aux)
## [1] 762
head(aux)
## Simple feature collection with 6 features and 6 fields
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -0.1975742 ymin: 51.49313 xmax: -0.08460569 ymax: 51.53006
## CRS:           EPSG:4326
##   id                name             area nbikes nempty capacity
## 1  1        River Street      Clerkenwell      4     14        9
## 2  2 Phillimore Gardens       Kensington      2     34       27
## 3  3 Christopher Street Liverpool Street      0     32       NA
## 4  4  St. Chad's Street     King's Cross      4     19       NA
## 5  5     Sedding Street    Sloane Square     15     12       NA
## 6  6 Broadcasting House       Marylebone      0     18        8
##                      geometry
## 1  POINT (-0.1099705 51.52916)
## 2  POINT (-0.1975742 51.49961)
## 3 POINT (-0.08460569 51.52128)
## 4  POINT (-0.1209737 51.53006)
## 5   POINT (-0.156876 51.49313)
## 6  POINT (-0.1442289 51.51812)
```

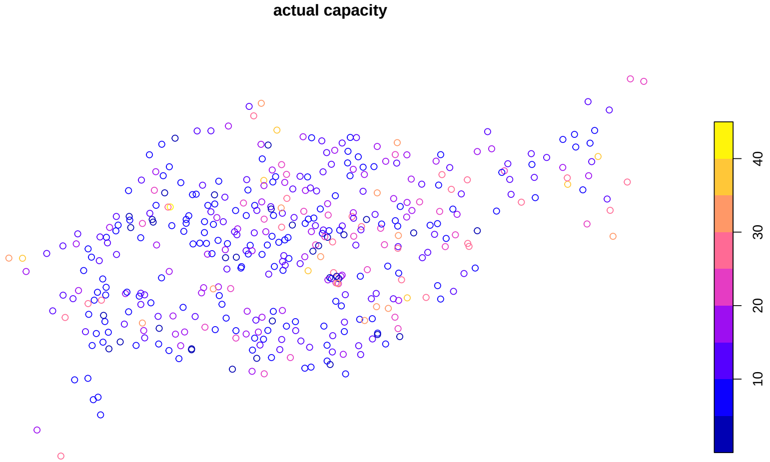
Non-overlapping joins

- Note that the number of rows in the joined result is greater than the target.
- This is because some cycle hire stations in cycle_hire have multiple matches in cycle_hire_osm.
  - our method generated multiple candidate points to be coupled with the official data.
- Use aggregation methods:
  - Take the capacity mean of the candidates and assign that to the corresponding point in the official data.

```
aux = aux %>%
  group_by(id) %>%
  summarize(capacity = mean(capacity))
nrow(aux) == nrow(cycle_hire)
## [1] TRUE
#> [1] TRUE
```

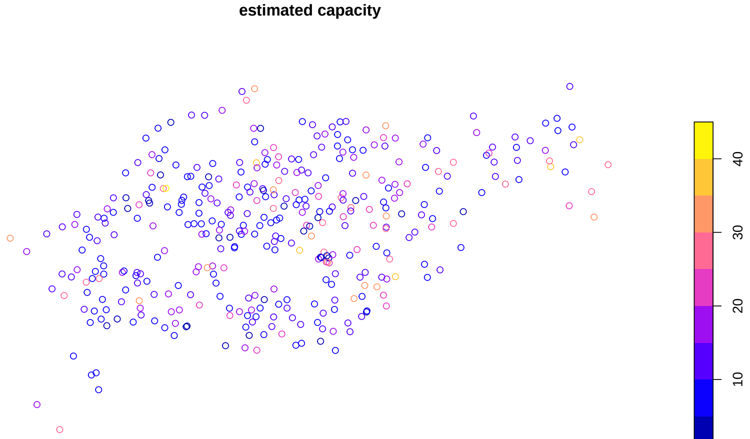
Non-overlapping joins

```
plot(cycle_hire_osm["capacity"], main="actual capacity")
```



Non-overlapping joins

```r
plot(aux["capacity"], main= "estimated capacity")
```



Spatial Aggregation

- Spatial data aggregation condenses data:
  - aggregated outputs have fewer rows than non-aggregated inputs.
- Statistical aggregation (mean average or sum) return a single value per grouping variable.
- Consider New Zealand: find out the average height of high points in each region
  - it is the geometry of the source (`nz`) that defines how values in the target object (`nz_height`) are grouped.
- Show the average value of features in nz_height within each of New Zealand's 16 regions.
  - pipe the output from st_join() into the 'tidy' functions group_by() and summarize().

Spatial Aggregation

- The code below says: *from nz, take those elements that intersect with* `nz_height`

```
(nz_agg2 = st_join(x = nz, y = nz_height, join = st_intersects))
## Simple feature collection with 110 features and 8 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 1090144 ymin: 4748537 xmax: 2089533 ymax: 6191874
## CRS:           EPSG:2193
## First 10 features:
##                   Name Island Land_area Population Median_income Sex_ratio
## 1            Northland  North 12500.561     175500         23400 0.9424532
## 2             Auckland  North  4941.573    1657200         29600 0.9442858
## 3              Waikato  North 23900.036     460100         27900 0.9520500
## 3.1            Waikato  North 23900.036     460100         27900 0.9520500
## 3.2            Waikato  North 23900.036     460100         27900 0.9520500
## 4        Bay of Plenty  North 12071.145     299900         26200 0.9280391
## 5             Gisborne  North  8385.827      48500         24400 0.9349734
## 6           Hawke's Bay  North 14137.524     164000         26100 0.9238375
## 7             Taranaki  North  7254.480     118000         29100 0.9569363
## 8   Manawatu-Wanganui  North 22220.608     234500         25000 0.9387734
##     t50_fid elevation                           geom
## 1        NA        NA MULTIPOLYGON (((1745493 600...
## 2        NA        NA MULTIPOLYGON (((1803822 590...
## 3   2408397      2751 MULTIPOLYGON (((1860345 585...
## 3.1 2408406      2720 MULTIPOLYGON (((1860345 585...
## 3.2 2408411      2732 MULTIPOLYGON (((1860345 585...
## 4        NA        NA MULTIPOLYGON (((2049387 583...
## 5        NA        NA MULTIPOLYGON (((2024489 567...
## 6        NA        NA MULTIPOLYGON (((2024489 567...
## 7        NA        NA MULTIPOLYGON (((1740438 571...
## 8   2408394      2797 MULTIPOLYGON (((1866732 566...
```

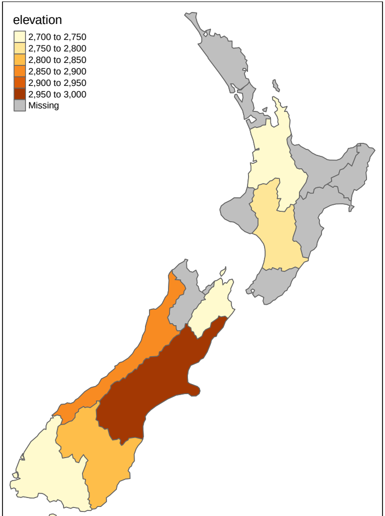
Spatial Aggregation

- The code below aggregates nz_agg2

```
nz_agg2 = nz_agg2 %>%
  group_by(Name) %>%
  summarize(elevation = mean(elevation, na.rm = TRUE))
  head(nz_agg2)
## Simple feature collection with 6 features and 2 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 1325039 ymin: 5004766 xmax: 2089533 ymax: 6007878
## CRS:           EPSG:2193
## # A tibble: 6 x 3
##   Name              elevation                                         geom
##   <chr>                 <dbl>                           <MULTIPOLYGON [m]>
## 1 Auckland                NaN  (((1803822 5900006, 1791443 5900571, 1790082 5883~
## 2 Bay of Plenty           NaN  (((2049387 5832785, 2051016 5826423, 2040276 5825~
## 3 Canterbury            2995.  (((1686902 5353233, 1679996 5344809, 1673699 5328~
## 4 Gisborne                NaN  (((2024489 5674920, 2019037 5677334, 2016277 5683~
## 5 Hawke's Bay             NaN  (((2024489 5674920, 2024126 5663676, 2032576 5659~
## 6 Manawatu-Wanganui     2777   (((1866732 5664323, 1868949 5654440, 1865829 5649~
```

Spatial Aggregation

```
tmap::tm_shape(nz)+tmap::tm_borders() +
  tmap::tm_shape(nz_agg2) + tmap::tm_polygons("elevation")
```



Spatial Aggregation

The resulting `nz_agg` objects have the same geometry as the aggregating object `nz` but with a new column summarizing the values of `x` in each region using the function `mean()` - It is a left-join.

Joining incongruent layers

- Spatial congruence: an aggregating object (y) is congruent with the target object (x) if the two objects have shared borders. -Often true for administrative boundary data, counties are congruent with states.
- Incongruent aggregating objects: do not share common borders with the target.
  - Problematic for spatial aggregation
  - Aggregating the centroid of each sub-zone will not return accurate results.
- **Areal interpolation** overcomes this issue by **transferring values** from one set of areal units to another.
  - consists of algorithms including simple area weighted approaches.

Joining incongruent layers

Joining incongruent layers

- The dataset `incongruent`
  - colored polygons with black borders in the right panel
- The data set `aggregating_zones`
  - the two polygons with the translucent blue border.
- Assume that the value column of `incongruent` refers to the total regional income.
  - How can we *transfer the values* of the *underlying nine spatial polygons* into the two polygons of *aggregating_zones*?

Joining incongruent layers

Area weighted spatial interpolation

- Transfers values from the `incongruent` object to a new column in `aggregating_zones` **in proportion with the area of overlap**:
  - the larger the spatial intersection between input and output features, the larger the corresponding value.
  - This is implemented in `st_interpolate_aw()`

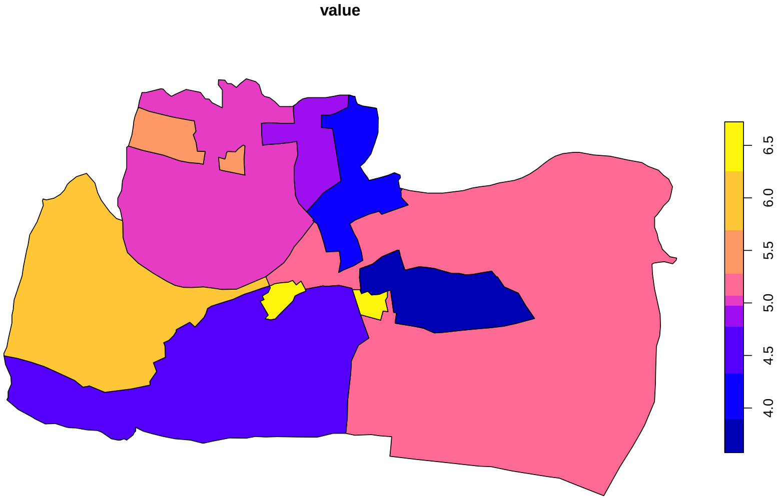
Joining incongruent layers

- The code below reads: take the income values from the smaller regions to estimate the income in the larger regions.
  - the weights of this sum correspond to the smaller areas relative size.

```
iv = incongruent %>% select(value) # keep only the values t
agg_aw = st_interpolate_aw(iv, aggregating_zones,
                           ext = TRUE)
## Warning in st_interpolate_aw.sf(iv, aggregating_zones, e
## st_interpolate_aw assumes attributes are constant or uni
```

Joining incongruent layers

```
plot(iv)
```



Joining incongruent layers

- Total income is a so-called **spatially extensive** variable (*which increases with area*)
  - Our aggregating method assumes income is evenly distributed across the smaller zones.
- This would be different for **spatially intensive** variables such as income *per capita* or percentages.
  - these do not increase as the area increases.
- `st_interpolate_aw()` works equally with spatially intensive variables
  - set the **extensive parameter** to FALSE and it will use an **average** rather than a weighted-sum function when doing the aggregation.

Distance relations

The distance between two objects is calculated with the st_distance() function. This is illustrated in the code chunk below, which finds the distance between the highest point in New Zealand and the geographic centroid of the Canterbury region

```
# with respect to elevation,
# take the top 1 observation.

nz_heighest = nz_height %>% top_n(n = 1, wt = elevation)
canterbury_centroid = st_centroid(canterbury)
## Warning in st_centroid.sf(canterbury): st_centroid assu
## constant over geometries of x
st_distance(nz_heighest, canterbury_centroid)
## Units: [m]
##          [,1]
## [1,] 115540
```

Distance Relations

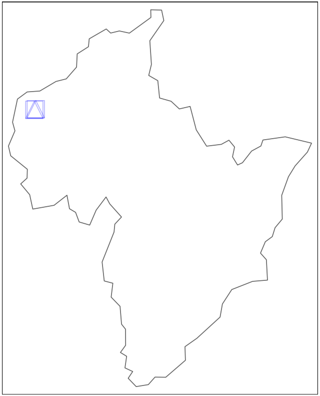
-There are two potentially surprising things about the result: - It has units (meters) - It is a matrix. - This second observation hints at another **useful feature** of `st_distance()` - it returns a distance matrix describing all combinations of features in objects x and y. - Find the distances between the first three features in nz_height and the Otago and Canterbury regions of New Zealand.

```
co = filter(nz, Name=="Canterbury" | Name=="Otago")
st_distance(nz_height[1:3, ], co)
## Units: [m]
##           [,1]     [,2]
## [1,] 123537.16 15497.72
## [2,]  94282.77     0.00
## [3,]  93018.56     0.00
```

Distance Relations

- Note that the distance between the second and third features in nz_height and the second feature in co is zero.
- This demonstrates the fact that distances between points and polygons refer to the distance **to any part of the polygon** - The second and third points in `nz_height` are in Otago, which can be verified by plotting them:

```
tmap::tm_shape(st_geometry(co)[2]) +tmap::tm_borders() +
tmap::tm_shape(st_geometry(nz_height)[2:3]) + tmap::tm_symbols(shape = 14, col = "blue", size = 2, alpha =
```



Raster data: subsetting

- We know how to retrieve values associated with specific cell IDs
  - or row and column combinations.
- Raster extraction can be by location (coordinates) and other spatial objects.
- **Coordinates subsetting**:
  - 'translate' them into a cell ID with `cellFromXY()`.
  - alternatively, use `terra::extract()` (clashes with `tidyverse`).
- Find the value of the cell that covers a point located at coordinates of 0.1, 0.1.
  - use `elev`

Raster data: subsetting

```
id = cellFromXY(elev, xy = matrix(c(0.1, 0.1), ncol = 2))
elev[id]
##   elev
## 1   16
# the same as
terra::extract(elev, matrix(c(0.1, 0.1), ncol = 2))
##   elev
## 1   16
```

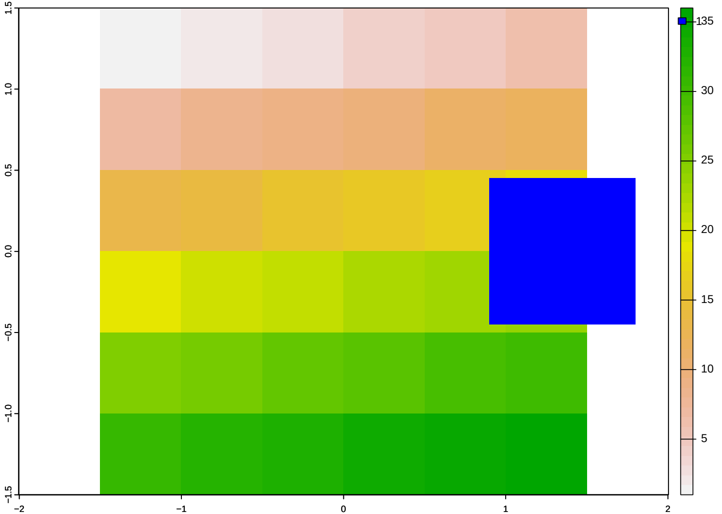
Raster data: subsetting

- You can **subset** one raster with **another raster**, as demonstrated below:

```
# raster with only 1s across 9 cells

clip = rast(xmin = 0.9, xmax = 1.8, ymin = -0.45, ymax = 0.
            resolution = 0.3, vals = rep(1, 9))
elev[clip]
##      elev
## [1,]   18
## [2,]   24
# we can also use extract
# terra::extract(elev, ext(clip))
#plot(elev)
#plot(clip, add=T,col="blue")
```

Raster data: subsetting

```
plot(elev)
plot(clip, add=T,col="blue")
```



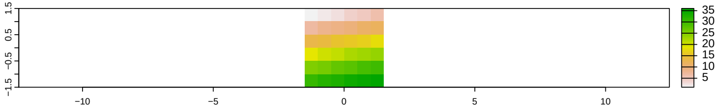
Raster data: subsetting

- This amounts to retrieving the values of the first raster object (in this case elev) that fall within the extent of a second raster (here: clip).

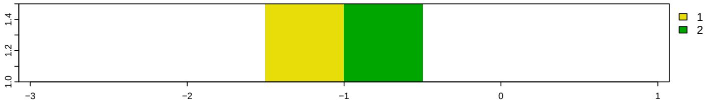
Raster data: subsetting

- The preceding example returned **the values** of specific cells.
- In many cases one needs spatial outputs from subsetting rasters.
    - This can be done using the **[** operator, with drop = FALSE.
    - obtain the first two cells of `elev` as a raster object (the first two cells on the top row).

```
plot(elev)
```



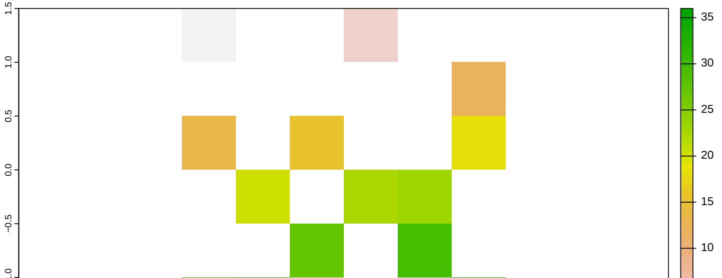

```
plot(elev[1:2, drop = FALSE] )
```



Raster data: subsetting

- Another common use case of spatial subsetting is when a raster with logical (or NA) values is used to access (mask) another raster with the same extent and resolution.
  - In this case, the mask() function can be used.
  - first, create a mask object (called `rmask`) with random NA and TRUE values.
  - next, keep those values of elev which are TRUE in `rmask`.

```
# create raster mask
rmask = elev
values(rmask) = sample(c(NA, TRUE), 36, replace = TRUE)

# spatial subsetting

plot(mask(elev, rmask))                    # with mask()
```



Raster data: subsetting

- We can also use the square bracket operator to overwrite some values

```
elev[elev < 20] = 100
plot(elev)
```

Map algebra operations

- There are 4 categories of MAOs
  - Depend on the specifics of the neighboring cells used for processing.

1. Local or per-cell operations.
2. Focal or neighborhood operations. Most often the output cell value is the result of a 3 x 3 input cell block.
3. Zonal operations are similar to focal operations, but the surrounding pixel grid on which new values are computed can have irregular sizes and shapes.
4. Global or per-raster operations; that means the output cell derives its value potentially from one or several entire rasters.

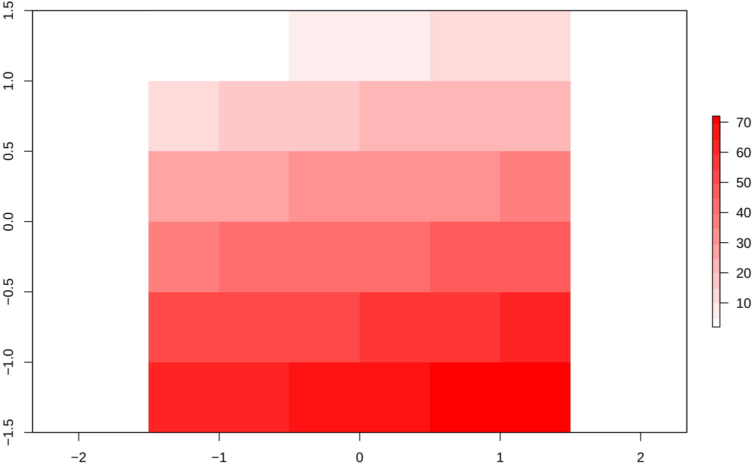
MAO: local operations

- cell-by-cell operations in one or several layers.
- Raster algebra: includes adding or subtracting values from a raster, squaring and multiplying rasters.
  - includes logical operations: find all raster cells that are greater than a specific value.
  - The `terra` package supports all these operations

```
data(elev)
elev_sum = elev + elev
elev_square = elev^2
elev_log = log(elev)
elev_5 = elev > 5
```

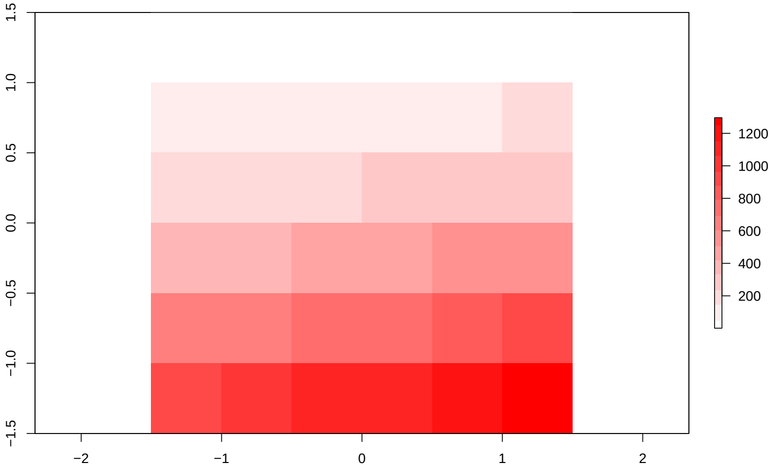
MAO: local operations

```
pal <- colorRampPalette(c("white","red"))
plot(elev_sum, col=pal(15))
```



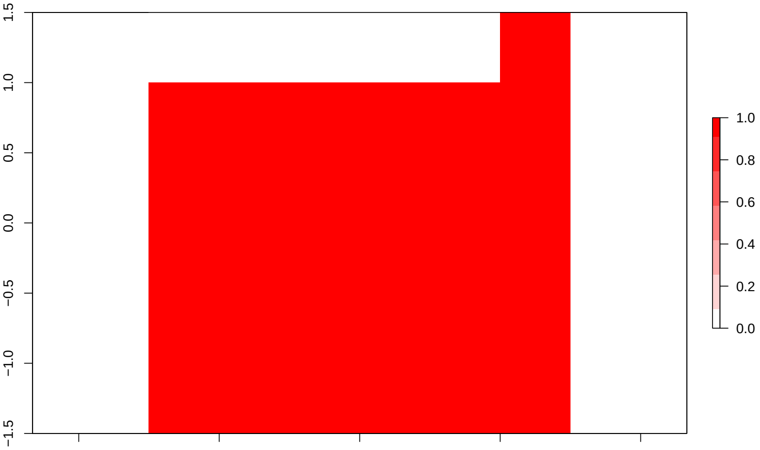
MAO: local operations

```
pal <- colorRampPalette(c("white","red"))

plot(elev_square,col=pal(15))
```



MAO: local operations

```
pal <- colorRampPalette(c("white","red"))

plot(elev_5, col=pal(7))
```



MAO: local operations

- Another local operation consist in creating groups of values:
  - low (class 1), middle (class 2) and high elevations (class 3).
- We need first to construct a reclassification matrix.
  - the first column corresponds to the **lower end** of the class.
  - the second column corresponds to the **upper end** of the class.
  - the third column represents the **new value** for the **specified ranges** in column one and two.
- Use the `classify()` command.

MAO: local operations

```r
rcl = matrix(c(0, 12, 1, 12, 24, 2, 24, 36, 3),
             ncol = 3, byrow = TRUE)
rcl
##      [,1] [,2] [,3]
## [1,]    0   12    1
## [2,]   12   24    2
## [3,]   24   36    3
```

- Here, we assign the raster values in the ranges 0–12, 12–24 and 24–36 are reclassified to take values 1, 2 and 3, respectively.

MAO: local operations

```
recl = classify(rast(elev), rcl = rcl)
```

- Note that classify is a function from `terra`. We need to use `rast` on `elev` to make it a suitable `terra`'s input.

MAO: local operations

- The `classify()` function can be also used when we want to reduce the number of classes in our categorical rasters.
- Apart of arithmetic operators, one can also use the app(), tapp() and lapp() functions.
- They are more efficient, hence, they are preferable in the presence of large raster datasets.
- Additionally, they allow you to save an output file directly.
- The app() function applies a function to each cell of a raster.
  - summarizes (e.g., calculating the sum) the values of multiple layers into one layer.
- `tapp()` extends app(), allowing us to select a subset of layers for which we want to perform a certain operation.
- `lapp()` applies a function to each cell using layers as arguments (more in a minute).

MAO: local operations

example: Normalized difference vegetation index (NDVI) - is a well-known local (pixel-by-pixel) raster operation. - It returns a raster with values between -1 and 1; - Positive values indicate the presence of living plants (mostly > 0.2). - Components of **NDVI** - **NIR** is a measure of light as well as **Red** calculated from satellite systems images. - The NVDI formula:

$$NDVI = \frac{NIR - Red}{NIR + Red}$$

MAO: local operations

- Below we calculate the NDVI for a raster reflecting the Zion National Park.

```
multi_raster_file = system.file("raster/landsat.tif", package = "spDataLarge")
multi_rast = rast(multi_raster_file)

# The raster object has four satellite bands - blue, green, red,
# and near-infrared (NIR).
# Our next step should be to implement the NDVI formula into an R function:

# create a function that takes the two
# types of light and computes NVDI

ndvi_fun = function(nir, red){
  (nir - red) / (nir + red)
}
```

MAO: local operations

- our function:
  - accepts two numerical arguments (`nir` and `red`).
  - returns a numerical vector with NDVI values
- It can be used as the `fun` argument of `lapp`.
- The raster contains 4 layers of light.
- We need two light layers: NIR and red from the raster which are the last two in a list of 4! (**mind the order**).
- That is why we subset the input raster with `multi_rast[[c(4, 3)]]` before doing any calculations.
  - This takes only the fourth and third of the layers.

MAO: local operations

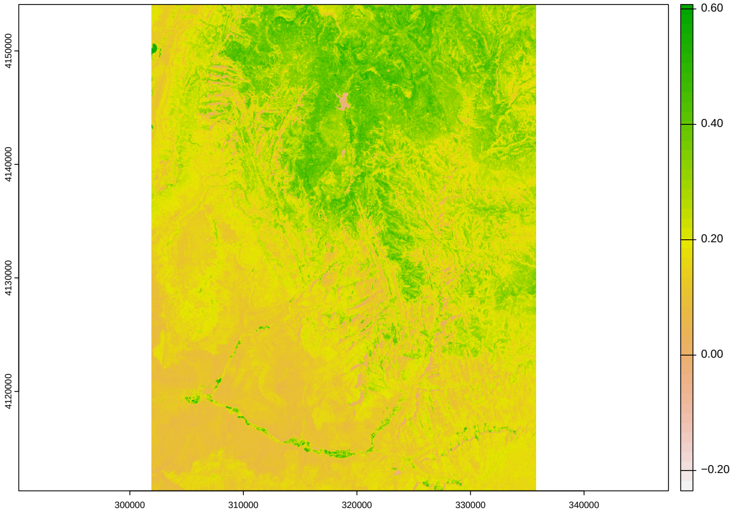
lapp: Apply a function to layers of a SpatRaster, or sub-datasets

```
ndvi_rast = lapp(multi_rast[[c(4, 3)]], fun = ndvi_fun)
```

MAO: local operations

- The largest NDVI values are connected to areas of dense forest in the North,
- The lowest values are related to a lake and snowy mountain ridges.

```
plot(ndvi_rast)
```



MAO: local operations

- Predictive mapping is another interesting application of local raster operations.
- The **dependent variable** corresponds to measured or observed points in space: pollution detectors.
- We can employ space predictor variables from various rasters (elevation, population density, temperature, etc.).
- Subsequently, we model our response as a function of our predictors
  - using `lm()`, `glm()`, `gam()` or a machine-learning technique.
- Then we construct predicted pollution values applying estimated coefficients to the predictor raster values.
  - Do you remember pollution and housing prices' example?

MAO: focal operations

- Focal operations work on a central (focal) cell and its neighbors.
- The neighborhood (kernel) is typically of size 3-by-3 cells
  - central cell and its eight surrounding neighbors,
  - but can take on any other shape as defined by the user.
- A focal operation applies an aggregation function to all cells within the specified neighborhood.
  - the output is set as the new value for the the central cell
  - the algorithm then moves on to the next central cell.
- Other names for this operation are spatial filtering and convolution.

MAO: focal operations

MAO: focal operations

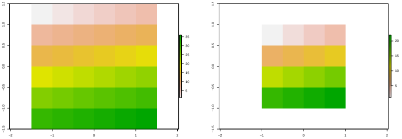
- In R, we can use the `focal()` function to perform **spatial filtering**.
- We define the kernel with a matrix whose values correspond to weights
- Secondly, the `fun` parameter lets us specify the aggregation function we wish to apply to this neighborhood.
- In what follows we choose the minimum.

MAO: focal operations

```
# First construct the kernel (AKA mooving window)

(w = matrix(1, nrow = 3, ncol = 3))
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1

# Now apply focal to the elevation data.

r_focal = focal(elev, w, fun = min)
```

MAO: focal operations

```
par(mar = c(4, 4, .3, .3))
plot(elev)
plot(r_focal)
# Use terra's values() to visualize the output.
matrix(terra::values(elev),nrow = 6, ncol = 6)
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    7   13   19   25   31
## [2,]    2    8   14   20   26   32
## [3,]    3    9   15   21   27   33
## [4,]    4   10   16   22   28   34
## [5,]    5   11   17   23   29   35
## [6,]    6   12   18   24   30   36
matrix(terra::values(r_focal),nrow = 6, ncol = 6)
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]  NaN  NaN  NaN  NaN  NaN  NaN
## [2,]  NaN    1    7   13   19  NaN
## [3,]  NaN    2    8   14   20  NaN
## [4,]  NaN    3    9   15   21  NaN
## [5,]  NaN    4   10   16   22  NaN
## [6,]  NaN  NaN  NaN  NaN  NaN  NaN
```



MAO: focal operations

- In this example, the weighting matrix consists only of 1s, meaning each cell has the same weight on the output, but this can be changed.
  - Focal functions play a dominant role in image processing.
  - Low-pass or smoothing focal functions use `mean` to remove extremes.
  - With categorical data, we can replace the mean with the mode (most common value).
- By contrast, high-pass filters accentuate features.
  - The Laplace and Sobel filters might serve as an example here.

MAO: focal operations

Terrain processing

- Calculation of topographic characteristics such as ground **slope** relies on focal functions.
- terrain() can be used to calculate these metrics
  - R provides several ground-processing algorithms including curvature and wetness indices.

MAO: Zonal operations

- Zonal operations apply an aggregation function to multiple raster cells.
- However, a second raster (with categorical values) defines the zones of interest
  - as opposed to a predefined neighborhood window (focal)
  - consequently, raster cells defining the zonal filter do not necessarily have to be neighbors
- Our grain (or ground, as we defined it earlier) raster is a good example: different grain sizes are spread irregularly throughout the raster.
- The result of a zonal operation is a **summary table** grouped by zone.
  - which is why this operation is also known as zonal statistics in GIS jargon.
  - This is in **contrast to focal operations** which return a **raster** object.

MAO: Zonal operations

- The following code chunk uses `terra`'s `zonal()` function to calculate the **mean elevation** associated with each grain size class, for example.

```
# first let's check the grain's structure
dim(grain)
## [1] 6 6 1
matrix(values(grain),nrow = 6,ncol = 6)
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    0    0    0    1    2
## [2,]    0    2    2    0    1    1
## [3,]    1    0    2    1    1    2
## [4,]    2    0    0    1    2    2
## [5,]    2    2    0    1    1    0
## [6,]    2    1    2    1    1    2
cats(grain)
## [[1]]
##   value grain
## 1     0  clay
## 2     1  silt
## 3     2  sand
# let's apply the zonal function to elev using grain as a filter provider.
# it tell us about the elevation per grain-type/size
z = zonal(rast(elev), grain, fun = "mean")
z
##   grain    layer
## 1  clay 14.80000
## 2  silt 21.15385
## 3  sand 18.69231
```

- This returns the the mean altitude for each grain size class. - it is also possible to get a raster with calculated statistics for each zone by setting the `as.raster` argument to `TRUE`.

MAO: global operations and distances

- Global operations: zonal operations with the entire raster dataset representing a single zone.
  - The most common global operations are descriptive statistics for the entire raster dataset (the minimum or maximum).
- Useful for the computation of distance and weight rasters.
- In the first case, one can calculate the distance from each cell to a specific target cell.
- For example, one might want to compute the distance to the nearest coast (see also `terra::distance()`).
- We might also want to consider mountains:
  - instead of pure distances, we would like also to consider that a trip is longer when mountain are amid the way.
  - we can weight the distance with elevation to 'prolong' the Euclidean distance.

MAO: global operations and distances

example - build a raster of the continents of the world where each cell equals the distance of that cell to the nearest coast. - This map must highlight the land areas that are most isolated inland. - Use raster::distance - it calculates the distance from each NA cell to the closest non-NA cell. - we need to create a raster that has NA for land pixels, and some other value for non-land pixels. - Use `raster::rasterize` - It transfers values associated with countries (polygons) to raster cells. - Values are transferred if the polygon covers the center of a raster cell.

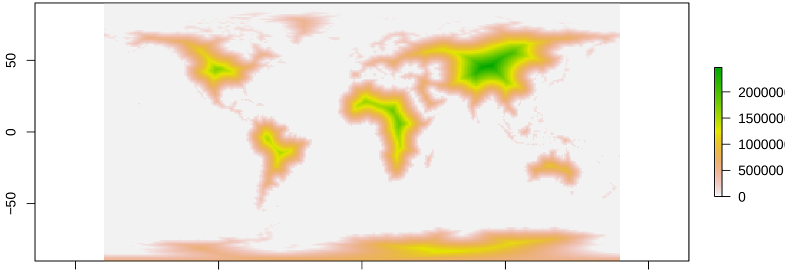
MAO: global operations and distances

example

```
library(maptools) # for data below
## Loading required package: sp
## Checking rgeos availability: TRUE
## Please note that 'maptools' will be retired by the end of 2023,
## plan transition at your earliest convenience;
## some functionality will be moved to 'sp'.
data(wrld_simpl)

# Here we use the raster package, but could have used terra instaed.
# Create a raster template.
# (set the desired grid resolution with res)

r <- raster::raster(xmn=-180, xmx=180, ymn=-90, ymx=90, res=1)

# Rasterize the countries polygons: 1 for land cells
# Nas for water.

r2 <- raster::rasterize(wrld_simpl, r, 1)
```

MAO: global operations and distances

```r
# the condition below is a land-sea indicator
# cond = is.na(r2)

# set land pixels to NA # water-pixels to sea
# maskvalue: what we want for water.
# updatevalue: what we want for land

r3 <- mask(is.na(r2), r2, maskvalue=1, updatevalue=NA)

# Calculate distance to nearest non-NA pixel
# don't run it, it takes too long

d <- raster::distance(r3)

plot(d)
```



Map algebra counterparts in vector processing

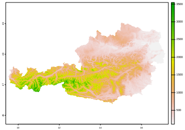
- Many map algebra operations have a counterpart in vector processing.
- Vector buffer operation: parallels computing a distance raster (global operation) while only considering a maximum distance (logical focal operation).
- Reclassifying raster data (either local or zonal function depending on the input) is equivalent to dissolving vector data (Section 4.2.4).
- Overlaying two rasters (local operation), where one contains NULL or NA values representing a mask, is similar to vector clipping (more later).
- Quite similar to spatial clipping is intersecting two layers.

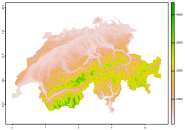
Merging rasters

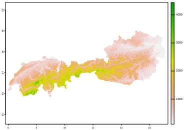
example - Suppose we need to conduct a study in an area that covers both Austria and Switzerland. - but we have separate raster for both countries. - In the following code chunk we first download the elevation data for Austria and Switzerland. - For the country codes, see the geodata function country_codes() - In a second step, we merge the two rasters into one.

Merging rasters

```
aut = geodata::elevation_30s(country = "AUT", path = tempdir())
ch = geodata::elevation_30s(country = "CHE", path = tempdir())
aut_ch = merge(aut, ch)
par(mar = c(4, 4, .3, .3))
plot(aut)
plot(ch)
plot(aut_ch)
# terra's merge() command combines two images,
# and in case they overlap, it uses the value of the first raster.
```







Merging rasters

- The function `mosaic()` allows you to define what todo when the rasters overlap but the variable's values are different.
  - you can for instance, take the mean of both rasters' values within the overlapping region.

Bonus: elevation weighted distance

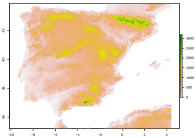
- get an elevation-raster of Spain
- Compute a raster which represents the distance to the coast.
- For speed, before computing the distance raster, increase the resolution of the input raster
- Secondly, weight the distance raster with elevation.
  - Every 100 **altitudinal meters** should increase the distance to the coast by 10 km.
  - Finally, compute the difference between the raster using the euclidean distance and the raster weighted by elevation.

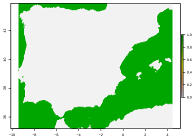
Bonus: elevation weighted distance

```
library(raster)
##
## Attaching package: 'raster'
## The following object is masked from 'package:dplyr':
##
##     select
## The following objects are masked from 'package:terra':
##
##     direction, gridDistance
# find out the ISO_3 code of Spain
dplyr::filter(ccodes(), NAME %in% "Spain")
##    NAME ISO3 ISO2 NAME_ISO NAME_FAO NAME_LOCAL SOVEREIGN       UNREGION1
## 1 Spain  ESP   ES    SPAIN    Spain     España    España Southern Europe
##   UNREGION2 continent
## 1    Europe    Europe
# retrieve a data -elevation-model of Spain
dem = getData("alt", country = "ESP", mask = FALSE)
# change the resolution to decrease computing time
agg = aggregate(dem, fact = 5)
#spain polygons
esp = getData("GADM", country = "ESP", level = 1)
```

Bonus: elevation weighted distance

```
par(mar = c(4, 4, .3, .3))
plot(dem)
# visualize NAs
plot(is.na(agg))
```



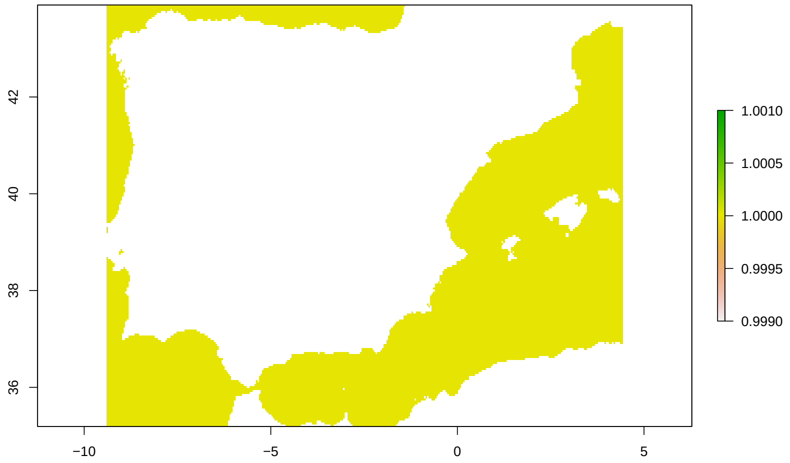


Bonus: elevation weighted distance

```
# construct a distance input raster
# we have to set the land cells to NA
# and the sea cells to an arbitrary value since
# raster::distance computes the distance to
# the nearest non-NA cell
dist = is.na(agg)
cellStats(dist, summary)
##    Mode   FALSE    TRUE
## logical   44595   24793
# convert land cells into NAs and sea cells into 1s
dist[dist == FALSE] = NA
dist[dist == TRUE] = 1
#plot(dist)
```

Bonus: elevation weighted distance

```
plot(dist)
```



Bonus: elevation weighted distance

- So far we have a raster of land-sea indicators (`dist`)
- We also have an elevation raster of Spain (`agg`)
- And the Spain polygons (`esp`)
- Let's compute the the land-to-water distances.
  - `dist = raster::distance(dist)`
- Restrict the focus to inland cells.
  - Use the Spain polygons to do that.
- Recall that the elevation data is measured in altitudinal meters.

Bonus: elevation weighted distance

```r
# compute distance to nearest non-NA cell
dist = raster::distance(dist)
# erase cells' contents that are not mainland.
dist = mask(dist, esp)
agg = mask(agg, esp)
# convert distance into km
dist = dist / 1000
# now let's weight each 100 altitudinal m.
# by an additional distance of 10 km.
agg[agg < 0] = 0 # only positive elevation.
# now create a raster with
# elev-weighted distance data.
weight = dist + agg / 100 * 10
#plot(weight - dist)
```

Bonus: elevation weighted distance

```
plot(weight - dist)
```