Lecture 10: Spatial Data in R

Vector Data Manipulation

James Sears

AFRE 891 SS 24

Michigan State University

Table of Contents: Today

Part 2: Vector Data Manipulation

Table of Contents: Next Time

Part 3: Raster Data

Prologue

To start, load in some packages/data from Part 1:

```r
if (!require("pacman")) install.packages("pacman")
pacman::p_load(sf, tidyverse, tmap, units)

counties<- st_read("data/MichiganCounties/MichiganCounties.shp")
```

```
## Reading layer `MichiganCounties' from data source
##   `F:\OneDrive - Michigan State University\Teaching\MSU 2023-2024\AFRE 891 S
##   using driver `ESRI Shapefile'
## Simple feature collection with 83 features and 15 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -90.41829 ymin: 41.69613 xmax: -82.41348 ymax: 48.2626
## Geodetic CRS:  WGS 84
```

```r
ingham <- filter(counties, NAME == "Ingham")
```

Preamble

Let's also define a custom ggplot map theme

```
# add ggplot map theme
maptheme <- theme(
  panel.background = element_rect(fill = NA),
  # panel.border = element_rect(fill = NA, color = "grey75"),
  axis.text.x=element_blank(), #remove x axis labels
  axis.ticks.x=element_blank(), #remove x axis ticks
  axis.text.y=element_blank(),  #remove y axis labels
  axis.ticks.y=element_blank(),  #remove y axis ticks
  panel.grid.major = element_blank(),
  panel.grid.minor = element_blank(),
  legend.key = element_blank())
```

Spatial Queries: Measurement

Spatial Queries

Spatial queries ask questions of our data and return data metrics, subsets, or new data objects.

The two basic types of spatial queries are

1. Spatial Measurement

- Return **continuous, numerical** answers
  - **Area:** How many square meters is Lake Superior?
  - **Length:** What is the length of I-96 in Michigan?
  - **Distance:** How far is East Lansing from Traverse City?

Spatial Queries

Spatial queries ask questions of our data and return data metrics, subsets, or new data objects.

2. Spatial Relationships

- Return **TRUE or FALSE** (binary predicates)
  - **Intersects, Overlaps, Touches, Disjoint, Crosses**
  - **Contains, Covered by, Covers, Within**
  - **Equal**
- Or the **set of matching features**
  - **Intersection**
  - **Difference**
  - **Union**
  - **Crop**

Spatial Queries

Note: sf can do a ***lot*** of additional spatial manipulations

Spatial manipulation with sf: : CHEAT SHEET

The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.

Geometric confirmation

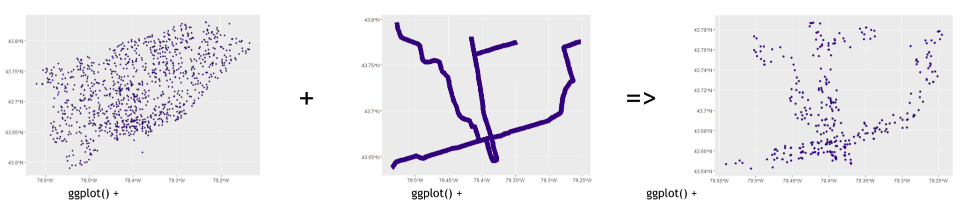
- st_contains(x, y, ...) Identifies if x is within y (i.e. point within polygon)
- st_covered_by(x, y, ...) Identifies if x is completely within y (i.e. polygon completely within polygon)
- st_covers(x, y, ...) Identifies if any point from x is outside of y (i.e. polygon outside polygon)
- st_crosses(x, y, ...) Identifies if any geometry of x have commonalities with y
- st_disjoint(x, y, ...) Identifies when geometries from x do not share space with y
- st_equals(x, y, ...) Identifies if x and y share the same geometry
- st_intersects(x, y, ...) Identifies if x and y geometry share any space
- st_overlaps(x, y, ...) Identifies if geometries of x and y share space, are of the same dimension, but are not completely contained by each other
- st_touches(x, y, ...) Identifies if geometries of x and y share a common point but their interiors do not intersect
- st_within(x, y, ...) Identifies if x is in a specified distance to y

Geometric operations

- st_boundary(x) Creates a polygon that encompasses the full extent of the geometry
- st_buffer(x, dist, nQuadSegs) Creates a polygon covering all points of the geometry within a given distance
- st_centroid(x, ..., of_largest_polygon) Creates a point at the geometric centre of the geometry
- st_convex_hull(x) Creates geometry that represents the minimum convex geometry of x
- st_line_merge(x) Creates linestring geometry from sewing multi linestring geometry together
- st_node(x) Creates nodes on overlapping geometry where nodes do not exist
- st_point_on_surface(x) Creates a point that is guarenteed to fall on the surface of the geometry
- st_polygonize(x) Creates polygon geometry from linestring geometry
- st_segmentize(x, dfMaxLength, ...) Creates linesting geometry from x based on a specified length
- st_simplify(x, preserveTopology, dTolerance) Creates a simplified version of the geometry based on a specified tolerance

Geometry creation

- st_triangulate(x, dTolerance, bOnlyEdges) Creates polygon geometry as triangles from point geometry
- st_voronoi(x, envelope, dTolerance, bOnlyEdges) Creates polygon geometry covering the envelope of x, with x at the centre of the geometry
- st_point(x, c(numeric vector), dim = "XYZ") Creating point geometry from numeric values
- st_multipoint(x = matrix(numeric values in rows), dim = "XYZ") Creating multi point geometry from numeric values
- st_linestring(x = matrix(numeric values in rows), dim = "XYZ") Creating linestring geometry from numeric values
- st_multilinestring(x = list(numeric matricesin rows), dim = "XYZ") Creating multi linestring geometry from numeric values
- st_polygon(x = list(numeric matrices in rows), dim = "XYZ") Creating polygon geometry from numeric values
- st_multipolygon(x = list(numeric matrices in rows), dim = "XYZ") Creating multi polygon geometry from numeric values



Spatial Queries

Note: sf can do a ***lot*** of additional spatial manipulations

Spatial manipulation with sf: : CHEAT SHEET

The sf package provides a set of tools for working with geospatial vectors, i.e. points, lines, polygons, etc.

Geometry operations

st_contains(x, y, ...) Identifies if x is within y (i.e. point within polygon)

st_crop(x, y, ..., xmin, ymin, xmax, ymax) Creates geometry of x that intersects a specified rectangle

st_difference(x, y) Creates geometry from x that does not intersect with y

st_intersection(x, y) Creates geometry of the shared portion of x and y

st_sym_difference(x, y) Creates geometry representing portions of x and y that do not intersect

st_snap(x, y, tolerance) Snap nodes from geometry x to geometry y

st_union(x, y, ..., by_feature) Creates multiple geometries into a a single geometry, consisiting of all geometry elements

Geometric measurement

st_area(x) Calculate the surface area of a polygon geometry based on the current coordinate reference system

st_distance(x, y, ..., dist_fun, by_element, which) Calculates the 2D distance between x and y based on the current coordinate system

st_length(x) Calculates the 2D length of a geometry based on the current coordinate system

Misc operations

st_as_sf(x, ...) Create a sf object from a non-geospatial tabular data frame

st_cast(x, to, ...) Change x geometry to a different geometry type

st_coordinates(x, ...) Creates a matrix of coordinate values from x

st_crs(x, ...) Identifies the coordinate reference system of x

st_join(x, y, join, FUN, suffix, ...) Performs a spatial left or inner join between x and y

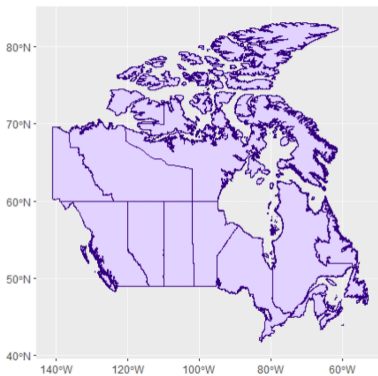
st_make_grid(x, cellsize, offset, n, crs, what) Creates rectangular grid geometry over the bounding box of x

st_nearest_feature(x, y) Creates an index of the closest feature between x and y

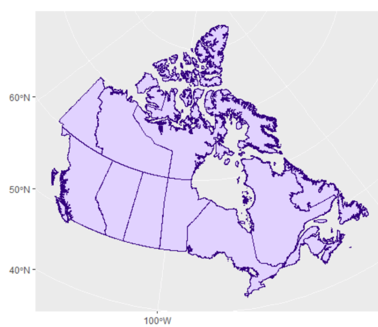
st_nearest_points(x, y, ...) Returns the closest point between x and y

st_read(dsn, layer, ...) Read file or database vector dataset as a sf object

st_transform(x, crs, ...) Convert coordinates of x to a different coordinate reference system




```
ggplot() +
  geom_sf(data = cdn) +
  coord_sf(crs = st_crs(4326))
```




```
ggplot() +
  geom_sf(data = cdn) +
  coord_sf(crs = st_crs(3347))
```

Calculating Area

```
st_area(ingham)
```

```
## 1449829600 [m^2]
```

Too big!

```
st_area(ingham)/1000000
```

```
## 1449.83 [m^2]
```

Wait, the units are wrong!

```
units::set_units(st_area(ingham), km^2)
```

```
## 1449.83 [km^2]
```

[**Resource: Measurement units in R**]()

Calculating Area

Next, let's add an `area` variable to our dataframe for each county:

```
counties2 <- mutate(counties,
                   area = units::set_units(st_area(counties), km^2))
mean(counties2$area)
```

```
## 1818.946 [km^2]
```

```
head(counties2$area)
```

```
## Units: [km^2]
## [1] 1795.1265 2420.5733 2178.4515 1536.1768 1356.7017  951.2289
```

Calculating Area: Impact of CRS

Recall that spatial measurements can differ greatly depending on the **CRS/projection we use!**

```
# Calculate area using data in WGS84 CRS (4326)
counties2$area_wgs84 <- st_transform(counties, 4326) %>% st_area() %>% se

# Calculate area using data in Web Mercator CRS (3857)
counties2$area_web <- st_transform(counties, 3857) %>% st_area() %>% set_u

# Calculate are using data in UTM Zone 16N, NAD83 (26916)
counties2$area_utm16 <- st_transform(counties, 26916) %>% st_area() %>% se

# Take a look at the results
counties2 %>% select(starts_with("area")) %>% st_drop_geometry() %>% head(
```

```
##                area        area_wgs84         area_web        area_utm16
## 1 1795.1265 [km^2] 1795.1265 [km^2] 3559.229 [km^2] 1800.7854 [km^2]
## 2 2420.5733 [km^2] 2420.5733 [km^2] 5102.846 [km^2] 2424.9251 [km^2]
## 3 2178.4515 [km^2] 2178.4515 [km^2] 4028.376 [km^2] 2180.7878 [km^2]
## 4 1536.1768 [km^2] 1536.1768 [km^2] 3082.999 [km^2] 1541.0568 [km^2]
## 5 1356.7017 [km^2] 1356.7017 [km^2] 2719.403 [km^2] 1359.3591 [km^2]
```

Calculating Area: Impact of CRS

Your choice of CRS is absolutely critical to accurate calculations.

- ***WGS84** is a geographic (unprojected) CRS in degrees... but `sf` calculates area using spherical geometry!
- **UTM16N** is optimized for most of MI. Calculations outside the zone will be increasingly distorted as we move farther away from the zone
- **Web Mercator** preserves shape and explicitly **distorts area, don't use it for area calculations!**

Calculating Length

`st_length()` calculates **feature lengths** in a spatial dataframe.

Let's first load in some data on railroad locations in Michigan.

```
#https://gis-michigan.opendata.arcgis.com/datasets/Michigan::railroads-v17a/explore?location
  ## load in the railroads data
rail <- st_read("data/Railroads/")
```

```
## Reading layer `Railroads_(v17a)' from data source
##   `F:\OneDrive - Michigan State University\Teaching\MSU 2023-2024\AFRE 891 SS24\Lecture-Slides\
##   using driver `ESRI Shapefile'
## Simple feature collection with 650 features and 10 fields
## Geometry type: LINESTRING
## Dimension:     XY
## Bounding box:  xmin: -90.21952 ymin: 41.7071 xmax: -82.42273 ymax: 46.79023
## Geodetic CRS:  WGS 84
```

Calculating Length

Let's first load in some data on railroad locations in Michigan.

```
st_crs(rail)
```

```
## Coordinate Reference System:
##   User input: WGS 84
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             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",4326]]
```

Q: should we reproject the railroad data before continuing?

Calculating Length

`st_length()` calculates **feature lengths** in a spatial dataframe.

```
rail <- mutate(rail,
               len = st_length(rail))
head(rail$len)
```

```
## Units: [m]
## [1]   120.1038   142.0916 10207.5456 11527.4791 26872.0126   455.0065
```

Note the output units! (meter)

--

This mostly matches the (unitless) `LENGTH` variable that was already in the dataframe

Calculating Length

Use `set_units()` from the **units** package to change to preferred units:

```
rail <- mutate(rail,
               len_mi = units::set_units(st_length(rail), mi),
               len_km = units::set_units(st_length(rail), km)
               )

select(rail, starts_with("len")) %>% st_drop_geometry() %>% head()
```

```
##       LENGTH            len           len_mi           len_km
## 1   120.3320   120.1038 [m]  0.07462907 [mi]  0.1201038 [km]
## 2   142.1354   142.0916 [m]  0.08829164 [mi]  0.1420916 [km]
## 3 10220.1411 10207.5456 [m]  6.34267476 [mi] 10.2075456 [km]
## 4 11551.5451 11527.4791 [m]  7.16284343 [mi] 11.5274791 [km]
## 5 26889.6901 26872.0126 [m] 16.69749452 [mi] 26.8720126 [km]
## 6   456.0733   455.0065 [m]  0.28272792 [mi]  0.4550065 [km]
```

Calculating Distance

How far are the railroads from the centroid of MSU's campus?

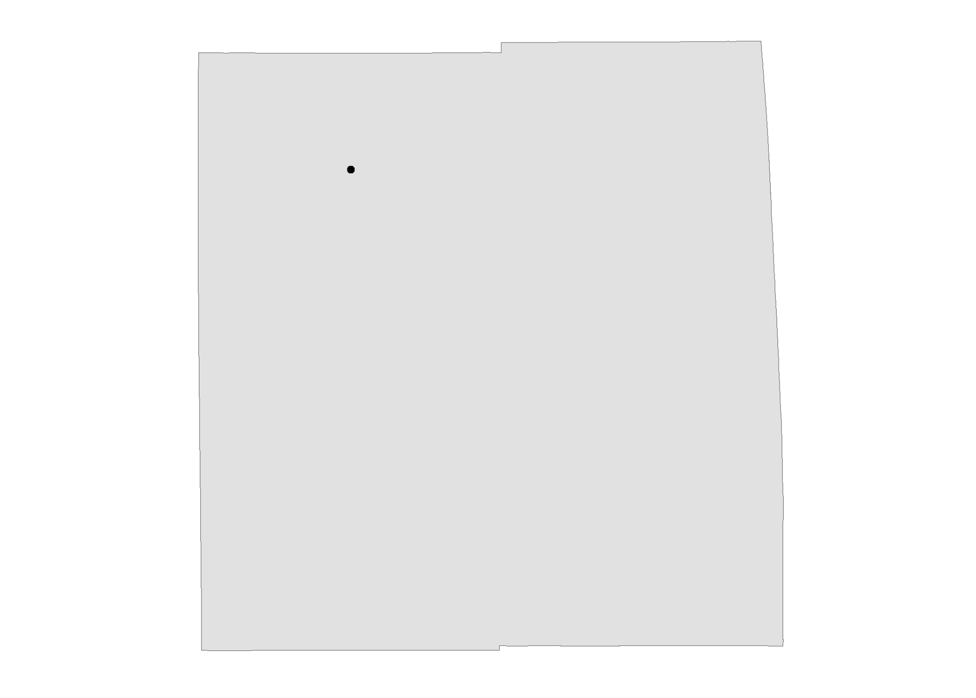
Let's create an sf object from point coordinates we **looked up on latlong.net**

```
msu <- data.frame(latitude = 42.701847,
                  longitude = -84.482170) %>% # build dataframe
  st_as_sf(coords = c("longitude", "latitude"), # specify variables with
           crs = 4326) # set WGS84 CRS
```

Calculating Distance

Plotting on top of the Ingham County polygon:

```
ggplot() +
  geom_sf(data = filter(counties, NAME == "Ingham")) +
  geom_sf(data = msu) +
  maptheme
```



Calculating Distance

`st_distance(X, Y)` calculates **spherical distances** by default

```
rail_msu_dist ← st_distance(rail, msu)
```

Calculating Distance

What does `st_distance()` return?

```
class(rail_msu_dist)
```

```
## [1] "units"
```

```
dim(rail_msu_dist)
```

```
## [1] 650   1
```

```
head(rail_msu_dist)
```

```
## Units: [m]
##          [,1]
## [1,] 577183.6
## [2,] 548383.0
## [3,] 548383.0
## [4,] 577183.6
## [5,] 523377.8
```

Calculating Distance

By default, `st_distance()` returns a `unit` object with

- One row per feature in X (650 rail lines)
- One column per feature in Y (1 point for MSU)

Calculating Distance

What about the distance to the **closest rail line** for our `counties` object?

```
rail_cnty_dist ← st_distance(counties, rail) %>% # unit object, 1 row pe
  as.data.frame() %>% # convert to data frame
  rowwise() %>% # group by row
  summarise(min = min(c_across(everything()))) # get row min across all c

counties$rail_dist ← rail_cnty_dist$min # add as column to counties
```

Which counties don't have a rail line in them?

```
counties$NAME[which(counties$rail_dist > 0)]
```

```
## Error in Ops.units(counties$rail_dist, 0): both operands of the expression s
```

That doesn't work since our variable has units!

Calculating Distance

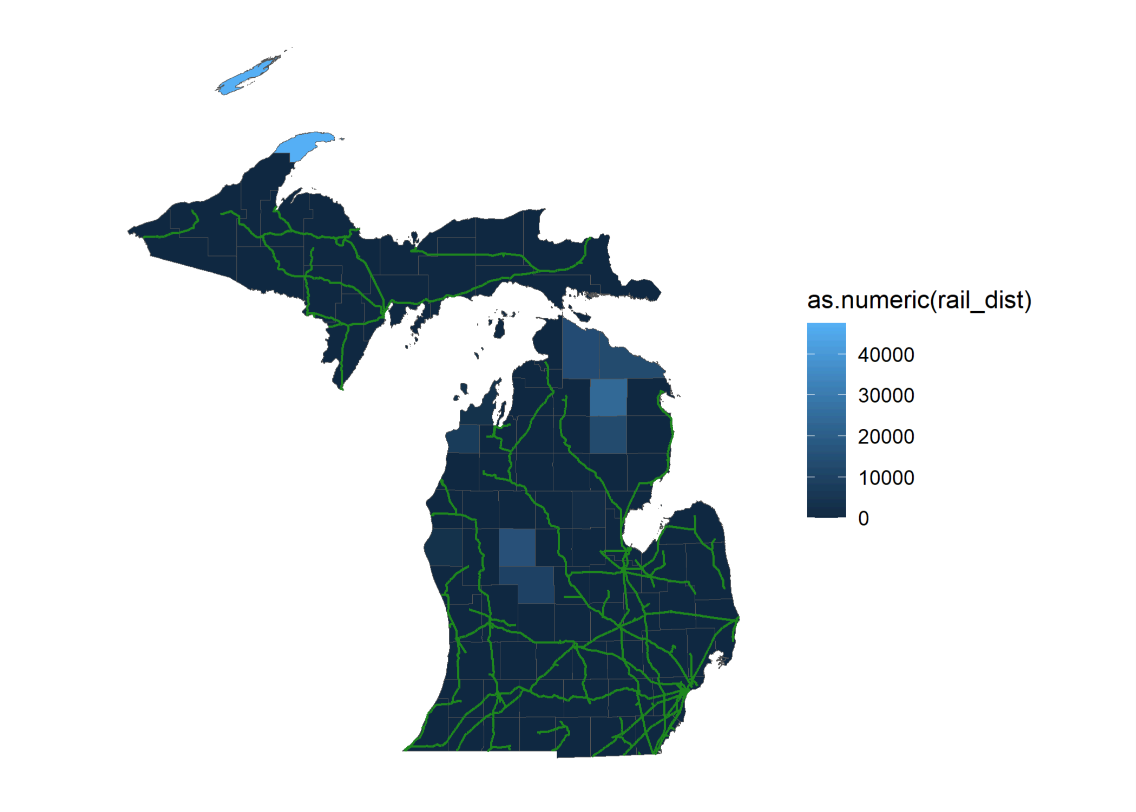
Solution: either need to compare to a units object or change the variable to numeric.

```
counties$NAME[which(counties$rail_dist > set_units(0, m))]
```

```
##  [1] "Benzie"        "Cheboygan"     "Gladwin"       "Keweenaw"      "Leelanau"
##  [6] "Mecosta"       "Montcalm"      "Montmorency"   "Oceana"        "Oscoda"
## [11] "Presque Isle"
```

Calculating Distance

```
ggplot() +
  geom_sf(aes(fill = as.numeric(rail_dist)), data = counties) +
  geom_sf(data = rail, color = "forestgreen") +
  maptheme
```



Spatial Queries: Relationship

Spatial Queries: Relationship

Often we want to understand how spatial objects relate to each other.

Binary predicates tell us whether a specified relationship holds between geometries, returning a matrix of **TRUE or FALSE**

Common binary predicates include

- `st_intersects()` (most general)
- `st_within()`
- `st_contains()` (inverse of `st_within()`)

We can also get more specific about the type of geometric confirmation

- `st_overlaps()`, `st_touches()`, `st_disjoint()`, `st_crosses()`
- `st_covered_by()`, `st_covers()`
- `st_equal()`

Spatial Queries: Relationship

Let's load in data on **protected wellhead areas** in MI to investigate

```r
whpa <- st_read("data/Wellhead_Protection_Areas_(WHPA)") %>%
  # fix invalid polygons
  st_make_valid()
```

```
## Reading layer `Wellhead_Protection_Areas_(WHPA)' from data source
##   `F:\OneDrive - Michigan State University\Teaching\MSU 2023-2024\AFRE 891 S
##   using driver `ESRI Shapefile'
## Simple feature collection with 2837 features and 12 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -90.22974 ymin: 41.71514 xmax: -82.52524 ymax: 47.4676
## Geodetic CRS:  WGS 84
```

```r
# Filter county shapefile to just Ingham
ingham <- filter(counties, NAME == "Ingham")
```

Spatial Queries: Relationship

And plot Ingham county with the wellhead protection areas…

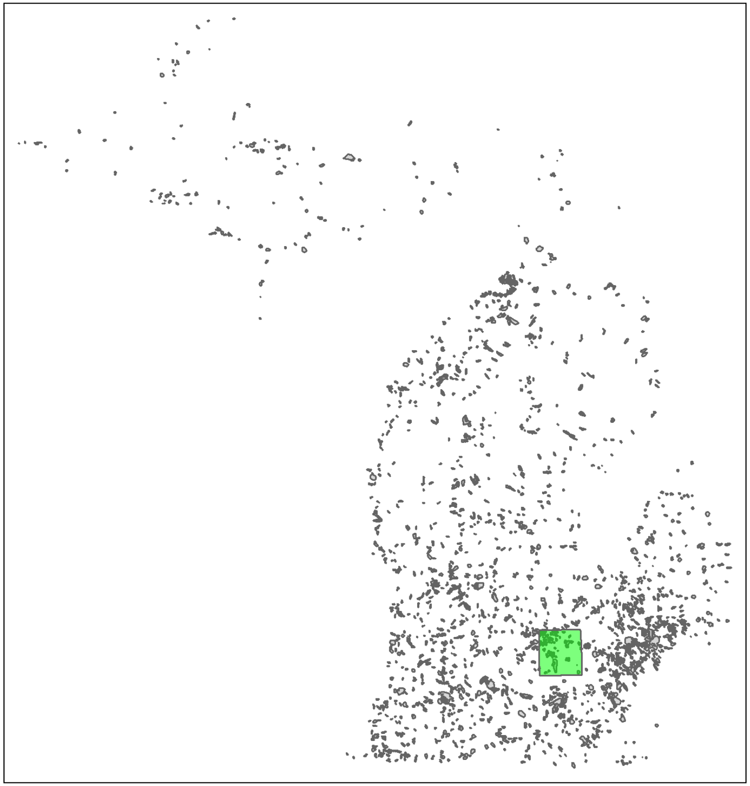
… but first verify that they're on the same CRS!

```
# Verify that CRS is the same as the rail line
stopifnot(st_crs(ingham) == st_crs(whpa))
```

Spatial Queries: Relationship

And *actually* plotting Ingham county with the wellhead protection areas

```
tm_shape(whpa) + tm_polygons() +
  tm_shape(ingham) + tm_polygons(col = "green", alpha = 0.5)
```



Spatial Queries: Relationship

Some protected areas are completely **within** the county, some **overlap**, and some are completely **outside**.

Our choice of geometric predicate will therefore influence *which* wellhead areas we select

Let's work through them and see how they differ.

Relationship Predicates

`st_intersects()` is the **most general** of confirmation functions

- returns `TRUE` if x and y geometry share **any space**, `FALSE` if **absolutely no shared space**

We could also be more specific with the form of space that's shared

- `st_crosses()` looks for **any commonalities** between x and y.
- `st_touches()` looks if x and y **share a common point** but **interiors don't intersect**
- `st_overlaps()` looks if x and y **share space**, are **of the same dimension**, but **aren't completely contained by one another**

Relationship Predicates

We could also be more specific with the form of space that's shared

- `st_within()` looks if x is **completely within** y
- `st_is_within_distance()` looks if x is **within a specified distance** of y
- `st_contains()` looks if x **completely contains** y.
- `st_covered_by()` looks if x is **completely covered by** y.
- `st_covers()` looks if x **completely covers** y.
- `st_disjoint()` looks if x **doesn't share space** with y.

Intersects

`st_intersects()` is the **most general** of confirmation functions

- returns `TRUE` if x and y geometry share **any space**, `FALSE` if **absolutely no shared space**

Let's use it to retrieve the counties adjacent to Ingham.

```
ingh_int <- st_intersects(counties, ingham)
class(ingh_int)
```

```
## [1] "sgbp" "list"
```

Intersects

`st_intersects()` is the **most general** of confirmation functions

- returns `TRUE` if x and y geometry share **any space**, `FALSE` if **absolutely no shared space**

By default, `st_intersects()` returns a **"sparse geometric binary predicate"** (`sgbp`) list

- 1 list element per geometry in x (wellheads)
- Each =1 if fully within y (ingham county)
- =0 if not fully within

Sparse vs. Non-Sparse

Setting `sparse = FALSE` yields the **entire TRUE/FALSE matrix**

- 1 row per row of X
- 1 column per row of Y
- element `i,j` whether geometry row `i` of X intersects geometry row `j` of Y

```
ingh_int_mat <- st_intersects(counties, ingham, sparse = FALSE)
class(ingh_int_mat)
```

```
## [1] "matrix" "array"
```

```
head(ingh_int_mat)
```

```
##        [,1]
## [1,] FALSE
## [2,] FALSE
## [3,] FALSE
## [4,] FALSE
```

st_filter

We could use index positions and `which` to extract matching geometries from the spatial dataframe…

But `st_filter()` makes the process of retrieving the matches even easier

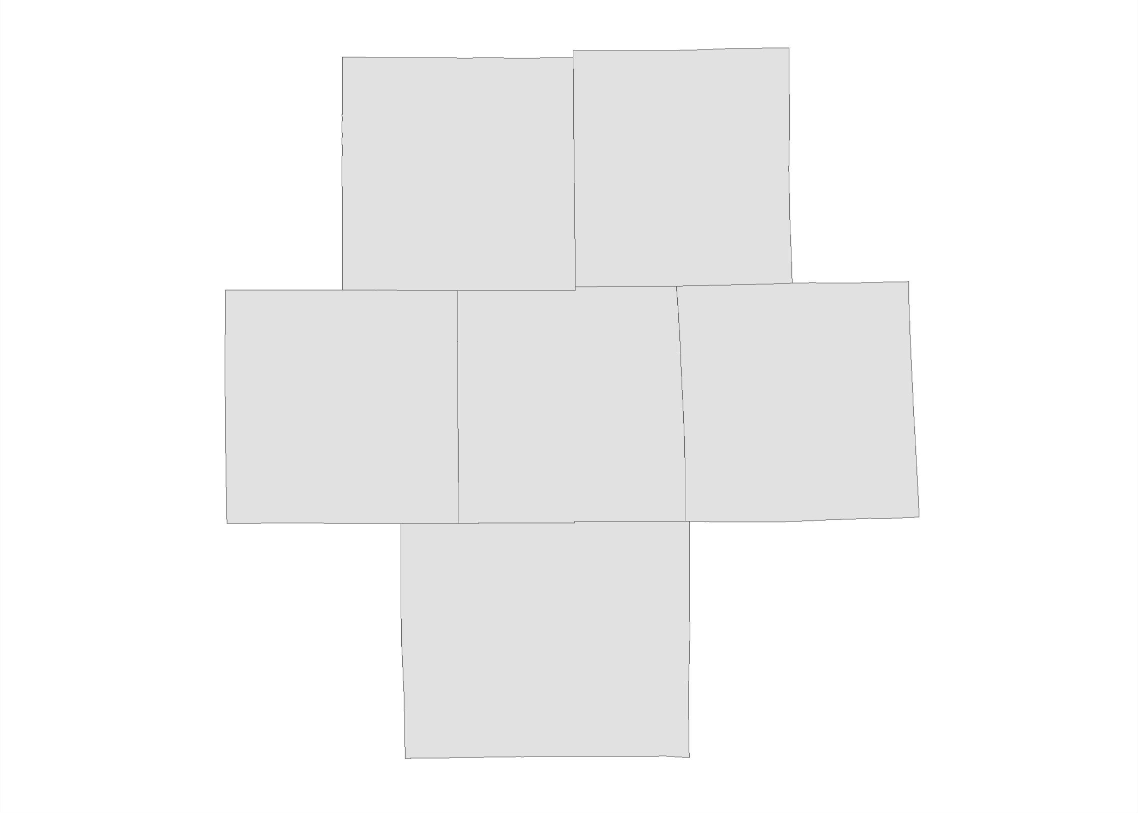
- returns a **spatial dataframe** with the geometries that return TRUE using the chosen predicate
- default predicate: `st_intersects()`
- change predicate with the `.predicate` optional argument

```
ingh_int_sf <- st_filter(counties, ingham)
ingh_int_sf$NAME
```

```
## [1] "Clinton"    "Eaton"      "Ingham"     "Jackson"    "Livingston"
## [6] "Shiawassee"
```

st_filter

```
ggplot() +
  geom_sf(data = ingh_int_sf) +
  maptheme
```



Intersect vs. Within

Within is a **stricter** operation than **intersection**

- `st_intersects()` returns both the "within" and "overlap" cases
- `st_within()` returns only the "completely within" cases

Let's find which protected areas lie **fully within** Ingham county:

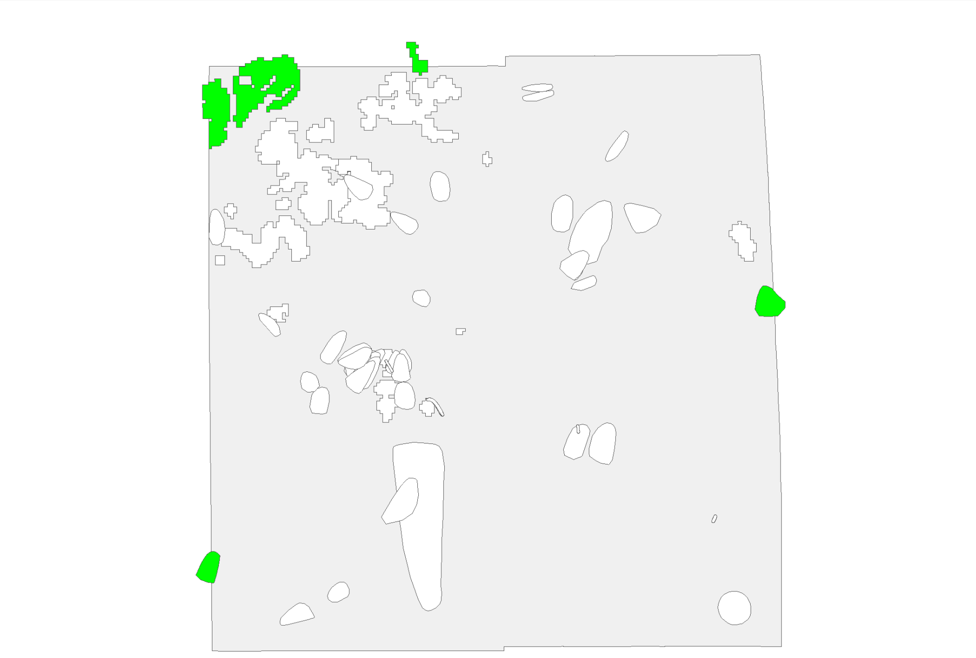
```
# Verify that CRS is the same as the rail line
stopifnot(st_crs(ingham) == st_crs(whpa))

whpa_within <- st_filter(whpa, ingham, .predicate = st_within)
```

Intersect vs. Within

Comparing **within** to **intersects** highlights the difference:

```
whpa_int <- st_filter(whpa, ingham)
ggplot() +
  geom_sf(data = ingham, alpha = 0.5) +
  geom_sf(data = whpa_int, fill = "green") +
  geom_sf(data = whpa_within, fill = "white") +
  maptheme
```



Within vs. Contains

`st_contains()` is the **inverse** of `st_within()`

- `st_contains(x,y)` returns the **transpose** of `st_within(y,x)`

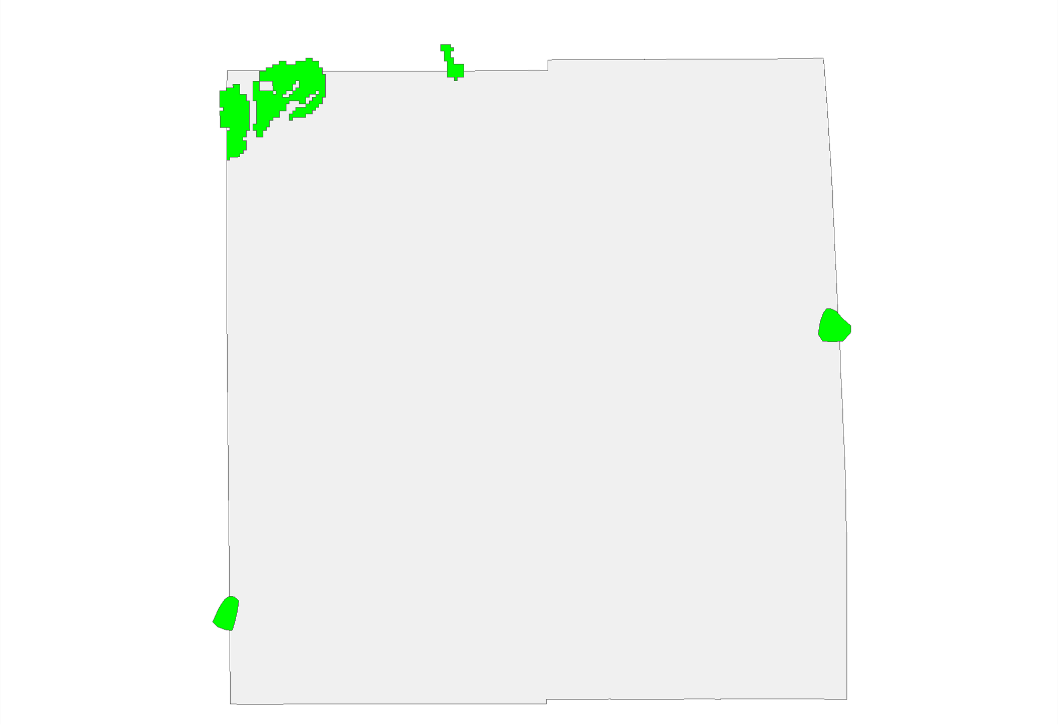
```
whpa_contain <- st_contains(ingham, whpa, sparse = FALSE) # 1 x 2837
whpa_wthn <- st_within(whpa, ingham, sparse = FALSE) # 2837 x 1
identical(whpa_contain, t(whpa_wthn))
```

```
## [1] TRUE
```

Overlaps

`st_overlaps()` yields the protected areas **partially in** Ingham County

```
ggplot() +
  geom_sf(data = ingham, alpha = 0.5) +
  geom_sf(data = st_filter(whpa, ingham, .predicate = st_overlaps), fill =
  maptheme
```



Covered by

`st_covered_by` yields the protected areas **completely covered by** Ingham County

- `st_covers` is the inverse

```
ggplot() +
  geom_sf(data = ingham, alpha = 0.5) +
  geom_sf(data = st_filter(whpa, ingham, .predicate = st_covered_by), fil
  maptheme
```

Touches

`st_touches` yields the protected areas that **only touch the boundary** of Ingham County

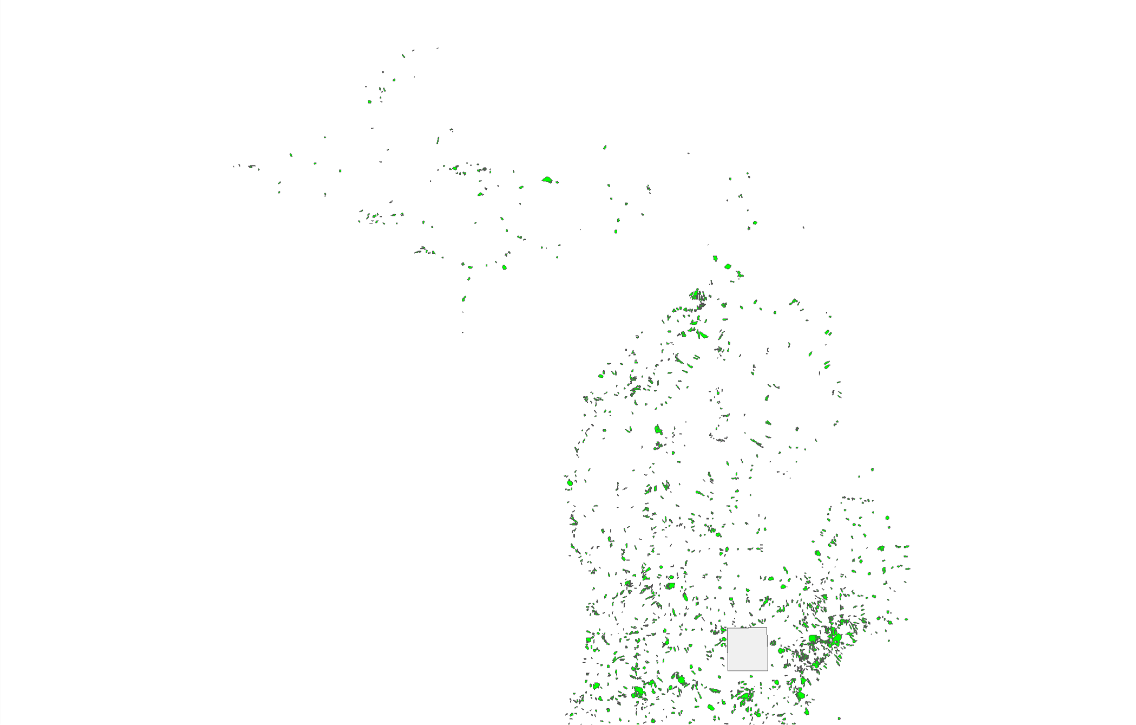
- Spoiler: there aren't any

```
ggplot() +
  geom_sf(data = ingham, alpha = 0.5) +
  geom_sf(data = st_filter(whpa, ingham, .predicate = st_touches), fill = "red")  +
  maptheme
```

Disjoint

`st_disjoint` returns the elements of x that **don't share space** with y

```
ggplot() +
  geom_sf(data = ingham, alpha = 0.5) +
  geom_sf(data = st_filter(whpa, ingham, .predicate = st_disjoint), fill =
  maptheme
```



Spatial Queries: Relationship

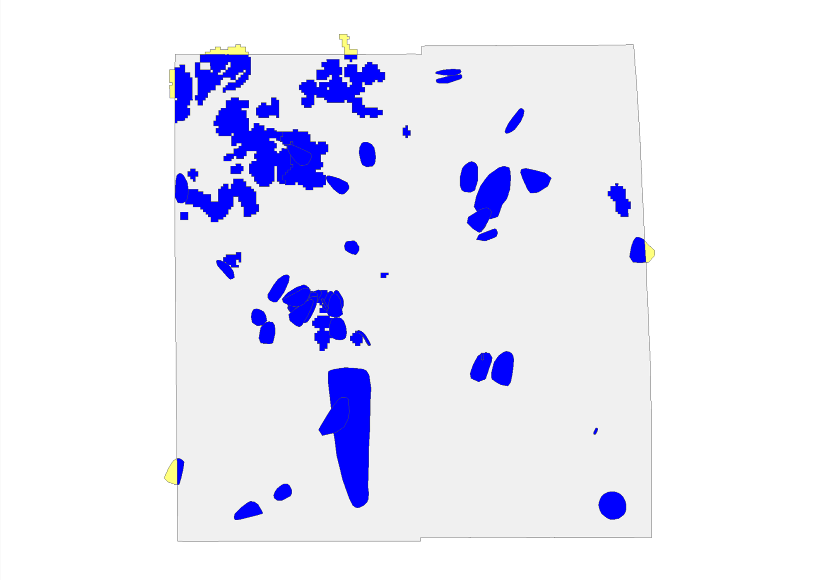
We can also **create new geometries** based on the relationship between two spatial objects

- `st_intersection()` creates geometry equal to the **shared portion** of x and y
- `st_union()` combines geometries of x and y into a **single geometry**
- `st_difference()` creates geometry equal to the **parts of x** that **don't intersect** y
- `st_sym_difference()` creates geometry equal to the **parts of x and y** that **don't intersect**
- `st_crop()` creates geometry from portion of x that **intersects a specified rectangle**

Intersection

`st_intersection()` trims the protected areas to **only the parts contained within** Ingham county

```
ggplot() +
  geom_sf(data = st_filter(whpa, ingham, .predicate = st_overlaps), fill =
  geom_sf(data = ingham, alpha = 0.5) +
  geom_sf(data = st_intersection(whpa, ingham), fill = "blue")  +
  maptheme
```



Intersection

Note that `st_intersection()` returns attributes from **both intersecting features**.

- Mindlessly, without checking whether they still makes sense for the **intersection** geometry.
- Now we have two different sets of area/length variables, plus acre/mile values for **all of Ingham County**

```
colnames(st_intersection(whpa, ingham))
```

```
##  [1] "OBJECTID"     "WSSN"         "SYSTEM"       "TYPE"         "APPROVAL_D
##  [6] "CreatedUse"   "CreatedDat"   "LastEdited"   "LastEdit_1"   "EgleSdeEGL
## [11] "ShapeSTAre"   "ShapeSTLen"   "OBJECTID.1"   "FIPSCODE"     "FIPSNUM"
## [16] "NAME"         "LABEL"        "TYPE.1"       "CNTY_CODE"    "SQKM"
## [21] "SQMILES"      "ACRES"        "VER"          "LAYOUT"       "PENINSULA"
## [26] "ShapeSTAre.1" "ShapeSTLen.1" "rail_dist"    "geometry"
```

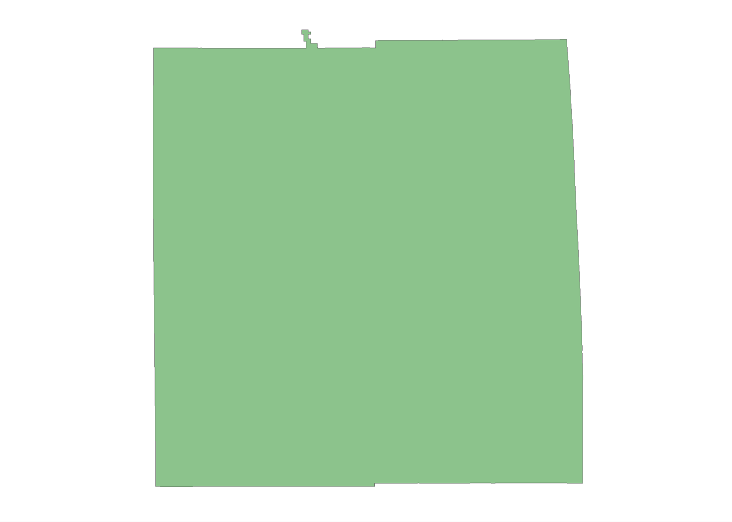
Be careful when interpreting attributes after geometry operations.

Union

`st_union()` combines **each geometry** in x with **each geometry of y**

- if a protected area falls fully within y, returns y's geometry
- if it falls partially outside, returns the conjoined geometry

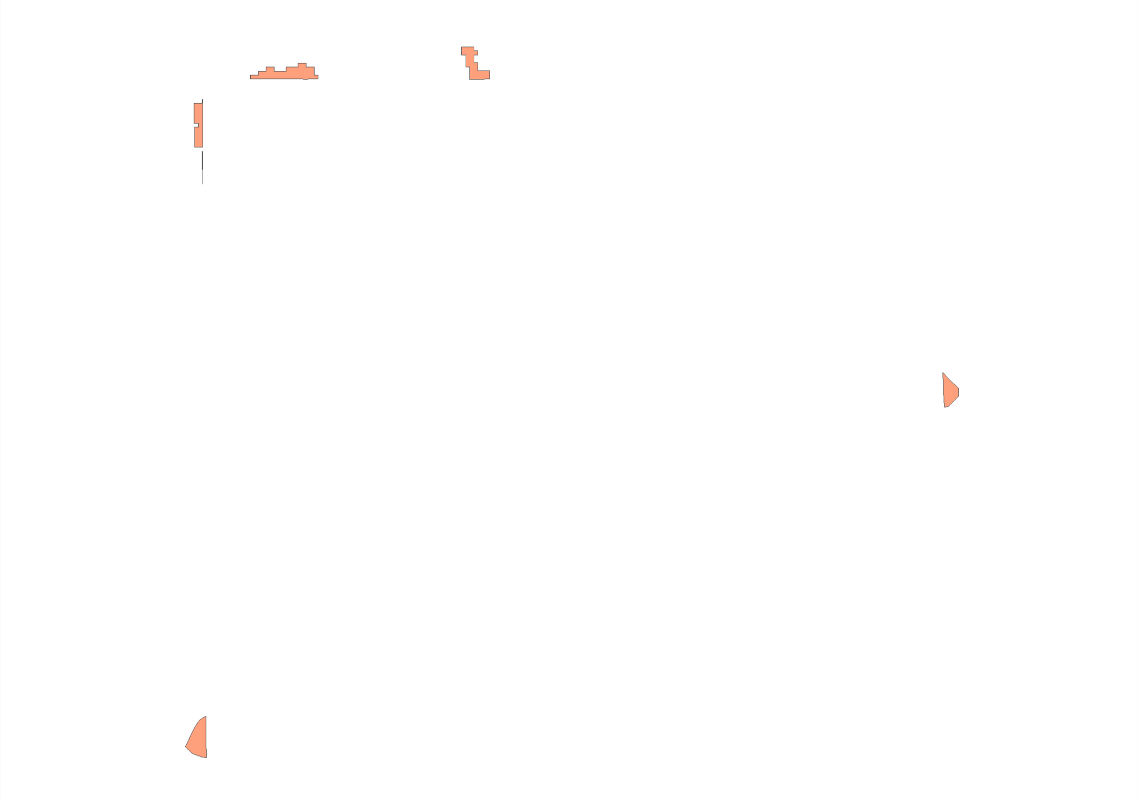
```
# calculate union
whpa_union <- st_union(whpa_int, ingham)
# plotting the new, unioned geometry in the fourth row
ggplot() +
  geom_sf(data = whpa_union[4,], alpha = 0.5, fill = "forestgreen")  + maptheme
```



Difference

`st_difference()` returns the **parts** of x (protected areas) **not in** y (Ingham)

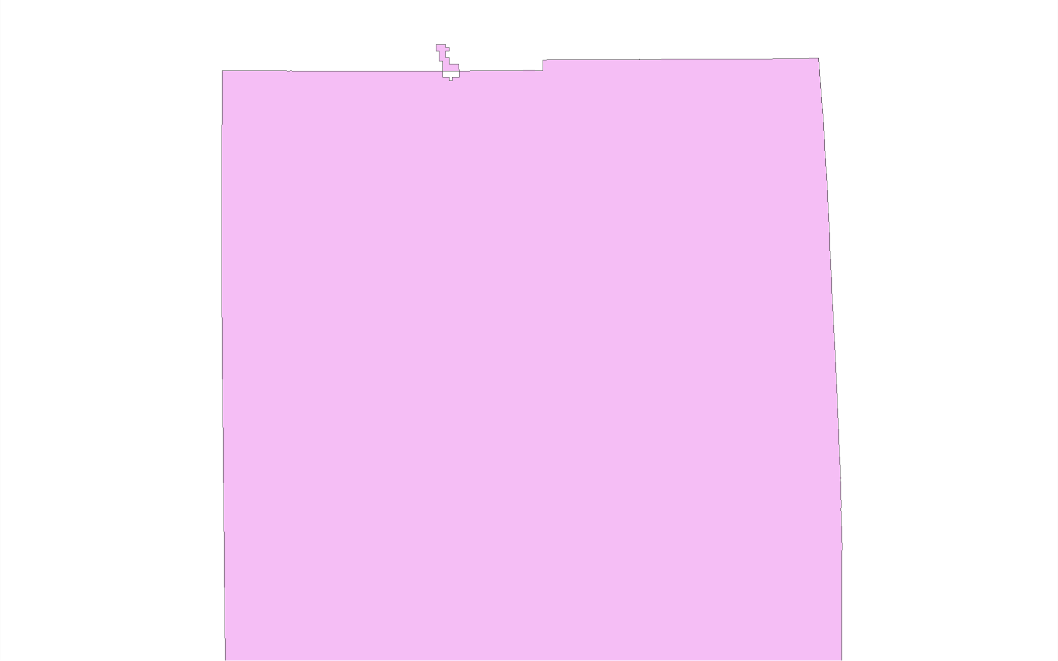
```
ggplot() +
  geom_sf(data = st_difference(whpa_int, ingham), alpha = 0.5, fill = "orangered")
  maptheme
```



Symmetric Difference

`st_sym_difference()` returns a geometry for each row of x that also includes the **parts of y** (Ingham) that are **not in** x (note the little missing overlapping piece)

```
ggplot() +
  geom_sf(data = st_sym_difference(whpa_int, ingham)[4,], alpha = 0.5, fill = "vic
  maptheme
```



Crop

`st_crop()` trims x to a **specified rectangle**

- **Easiest solution:** use `st_bbox()` as the y object
- **Less easy:** use `ymin`, `ymax`, `xmin`, `xmax` arguments to make the box manually

```
st_crop(whpa_int, st_bbox(ingham))
```

Crop

The cropping is a little hard to see with Ingham county's boundary

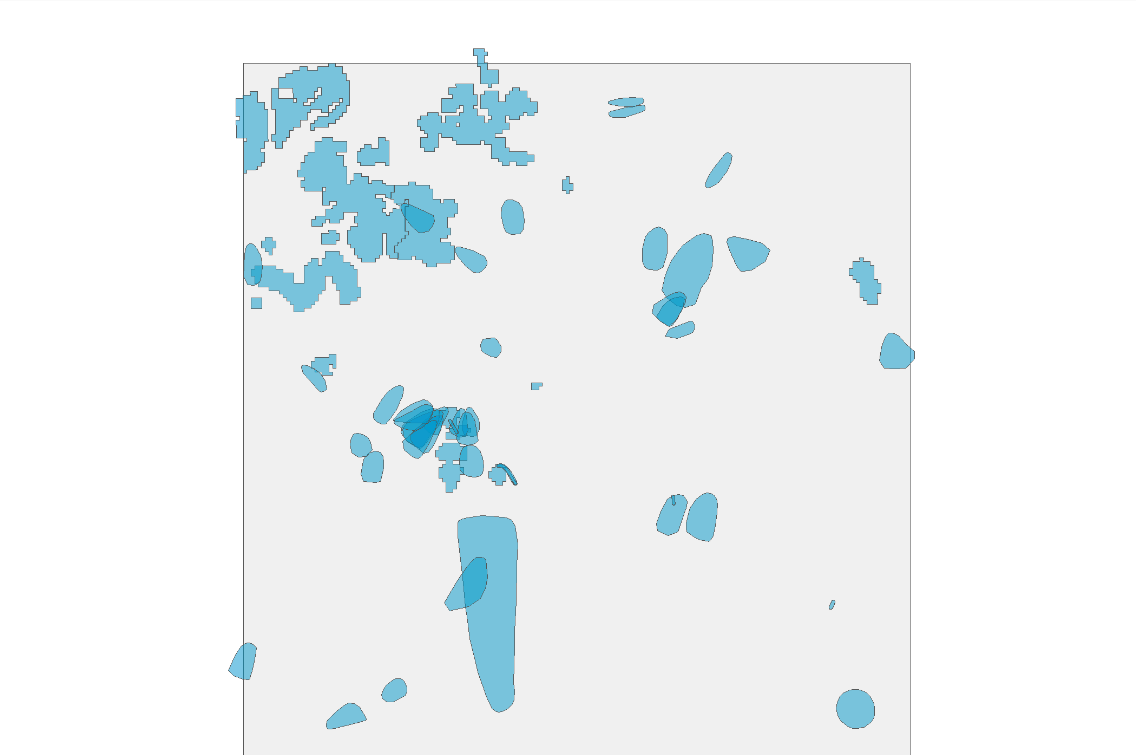
Let's convert the bounding box to an sf object so we can plot it instead.

```
# Create a dataframe with the desired point locations
box <- data.frame(lon = c(st_bbox(ingham)["xmin"], st_bbox(ingham)["xmax"]
                          st_bbox(ingham)["xmax"], st_bbox(ingham)["xmin"]
                          ,st_bbox(ingham)["xmin"] ),
                  lat = c(st_bbox(ingham)["ymin"], st_bbox(ingham)["ymin"]
                          st_bbox(ingham)["ymax"],st_bbox(ingham)["ymax"]
                          st_bbox(ingham)["ymin"])) %>%
  # convert to sf object
  st_as_sf(coords = c("lon", "lat"), crs = st_crs(ingham))%>%
  # combine the geometry and cast into a single polygon
  summarise((geometry = st_combine(geometry))) %>%
  st_cast("POLYGON")
```

Crop

Plotting the **uncropped** protected areas and the bounding box:

```
ggplot() +
  geom_sf(data = box, alpha = 0.5) +
  geom_sf(data = whpa_int, alpha = 0.5, fill = "deepskyblue3") +
  maptheme
```



Crop

Plotting the **cropped** protected areas and the bounding box:

```
ggplot() +
  geom_sf(data = box, alpha = 0.5) +
  geom_sf(data = st_crop(whpa_int, st_bbox(ingham)), alpha = 0.5, fill = "aquamari
  maptheme
```

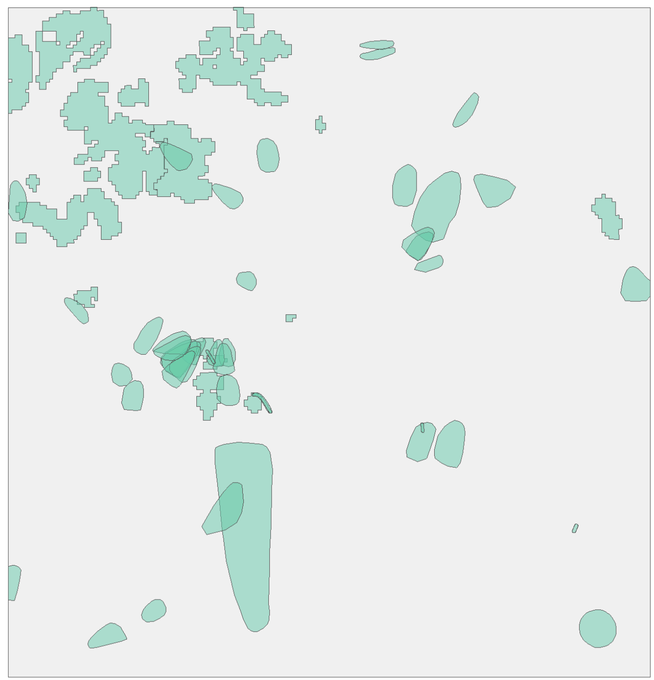


Table of Contents: