Geospatial Fundamentals in R with sf, Part 2

Patty Frontiera and Drew Hart, UC Berkeley D-Lab

August 2019

Part II Prep

1. Open the repo at https://github.com/dlab-berkeley/Geospatial-Fundamentals-in-R-with-sf
   - Download and unzip the zip file
   - Take note of where the folder is located
2. Start RStudio and open a **new script**, or **./docs/02-spatial_analysis.Rmd**
3. Set your working directory to the folder you unzipped
4. Install the required libraries in RStudio, ONLY IF YOU DO NOT HAVE THEM ALREADY!

```
our_packages<- c("ggplot2", "dplyr", "sf", "units", "tmap")
for (i in our_packages) {
  if ( i %in% rownames(installed.packages()) == FALSE) {
    install.packages(i)
  }
}
```

5. Open the slides, **./docs/02-spatial-analysis.html**, in your browser (or click the "Part 2 Slides" link the repo).

Part II Overview

Recap Part I

Tour of Spatial Analysis

Part I Recap

In Part I, we:

- Loaded geospatial data from CSV files
- Mapped data with `ggplot`
- Promoted data frames to `sf` objects with `sf::st_as_sf`
- Loaded geodata from shapefiles with `sf::st_read`
- Explored `CRSs` with `sf::st_crs`
- Transformed CRSs with `sf::st_transform`
- Mapped data with `tmap`

R Spatial Libraries

Let's load the libraries we will use

```
library(sf)      # spatial objects and methods
library(tmap)    # mapping spatial objects
```

Set your working directory

Use `setwd` to set your working directory to the location of the tutorial files.

For example:

```
setwd("~/Documents/Dlab/workshops/2018/rgeo/r-geospatial-workshop/r-geospatial-workshop")
```

Reload Part I data

You may want to reload the data that we had in our workspace at the end of Part I.

We've provided a little script for doing that, which you can run using the following line of code:

```
source('./docs/reload_part_01_data.R')
```

Spatial Analysis

The Spatial Analysis Workflow

1. Mapping / plotting to see location and distribution
2. Asking questions of, or querying, your data
3. Cleaning & reshaping the data
4. Applying analysis methods
5. Mapping analysis results
6. Repeat as needed

Transform data to common CRS

In order to perform spatial analysis we need to first convert all data objects to a common CRS.

Which type? Projected or Geographic CRS?

Geographic vs. Projected CRS

If my goal is to create maps, I may convert all data to a geographic CRS.

- Why? Which one?

If my goal is to do spatial analysis, I will convert to a projected CRS.

- Why? Which one?

Common CRS EPSG Codes

Geographic CRSs

- `4326` Geographic, WGS84 (default for lon/lat)
- `4269` Geographic, NAD83 (USA Fed agencies like Census)

Projected CRSs

- `5070` USA Contiguous Albers Equal Area Conic
- `3310` CA ALbers Equal Area
- `26910` UTM Zone 10, NAD83 (Northern Cal)
- `3857` Web Mercator (web maps)

Transform all layers to UTM 10N, NAD83

Use `st_transform` to transform `SFhomes15_sp` and `bart` to `UTM 10N, NAD83`

- `SFhighways` and `SFboundary` already have this CRS

Recall, this transformation is called `projecting` or `reprojecting`

The `EPSG` code is **26910**, units are meters.

Transform all layers to UTM 10, NAD83

First, transform `SFhomes15_sp`

(*Remember, this is also called* `reprojecting`*.*)

Note the two methods for doing same thing:

```r
#highways are already in 26910!
st_crs(SFhighways)
```

```
## Coordinate Reference System:
##   EPSG: 26910
##   proj4string: "+proj=utm +zone=10 +datum=NAD83 +units=m +no_defs"
```

```r
#so we can use them as the target CRS
SFhomes15_utm <- st_transform(SFhomes15_sf, st_crs(SFhighways))

#OR we could just use the EPSG code directly
#SFhomes15_utm <- st_transform(SFhomes15_sf, 26910)
```

Transform the boundary?

```r
# Check the CRS
st_crs(SFboundary) == st_crs(SFhomes15_utm)
```

```
## [1] FALSE
```

```r
# Transform
SFboundary_utm <- st_transform(SFboundary, st_crs(SFhomes15_utm))

# Check again
st_crs(SFboundary_utm) == st_crs(SFhomes15_utm)
```

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

BART data - Challenge

Transform the `bart_sf` object to UTM 10N.

Name the new object `bart_utm`

Challenge: Solution

```r
# Transform Bart to UTM
bart_utm <- st_transform(bart_sf, st_crs(SFhomes15_utm))
```

Check

Do the CRSs all match?

```
st_crs(bart_utm)$epsg
```

```
## [1] 26910
```

```
st_crs(SFboundary_utm)$epsg
```

```
## [1] 26910
```

```
st_crs(SFhighways)$epsg
```

```
## [1] 26910
```

```
st_crs(SFhomes15_utm)$epsg
```

```
## [1] 26910
```

Map all layers

Visual check

```
plot(SFboundary_utm)
lines(SFhighways, col='purple', lwd=4)
```

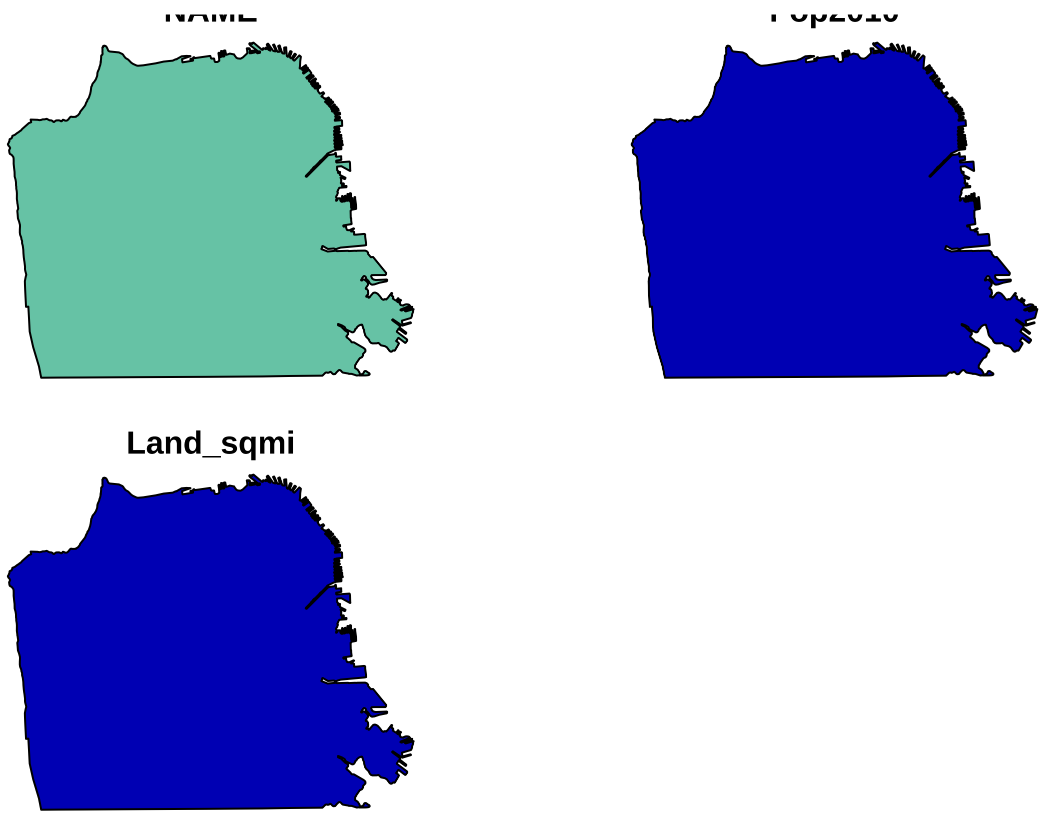
```
## Error in data.matrix(x): (list) object cannot be coerced to type 'double'
```

```
points(SFhomes15_utm)
```

```
## Warning in data.matrix(x): NAs introduced by coercion

## Warning in data.matrix(x): NAs introduced by coercion

## Warning in data.matrix(x): NAs introduced by coercion

## Warning in data.matrix(x): NAs introduced by coercion
## Error in data.matrix(x): (list) object cannot be coerced to type 'double'
```

```
plot(bart_utm, col="red", pch=15, add=T)
```

```
## Warning in plot.sf(bart_utm, col = "red", pch = 15, add = T): ignoring all
## but the first attribute
```



Map all layers

What happened?

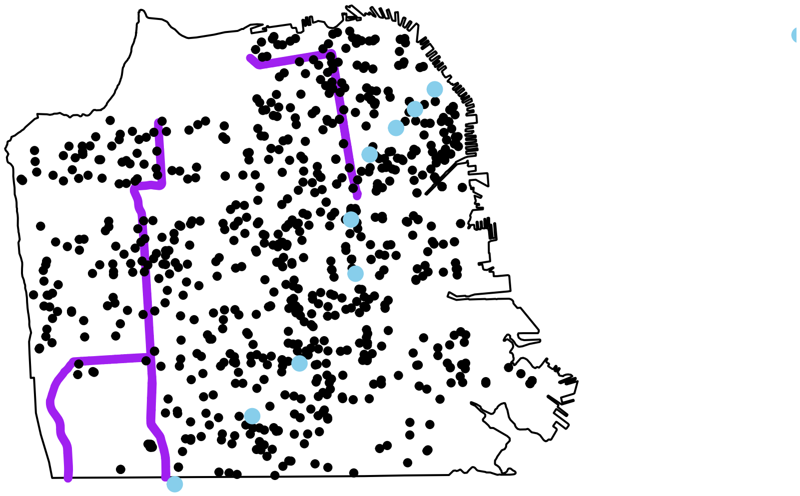
Two things:

1. Remember, by default, `sf`'s `plot` method will plot a grid of maps, one for each variable in the `data.frame`!
2. We can't just plot `sf` objects directly with calls to R's `lines` and `points` functions.

Map all layers

However, we can get what we want easily, with the help of the `st_geometry` function:

```
plot(st_geometry(SFboundary_utm))
plot(st_geometry(SFhighways), col='purple', lwd=4, add = T)
plot(st_geometry(SFhomes15_utm), add = T, pch = 19, cex = 0.5)
plot(st_geometry(bart_utm), col="skyblue", pch=19, cex = 1, add=T)
```



Challenge (Optional / time permitting)

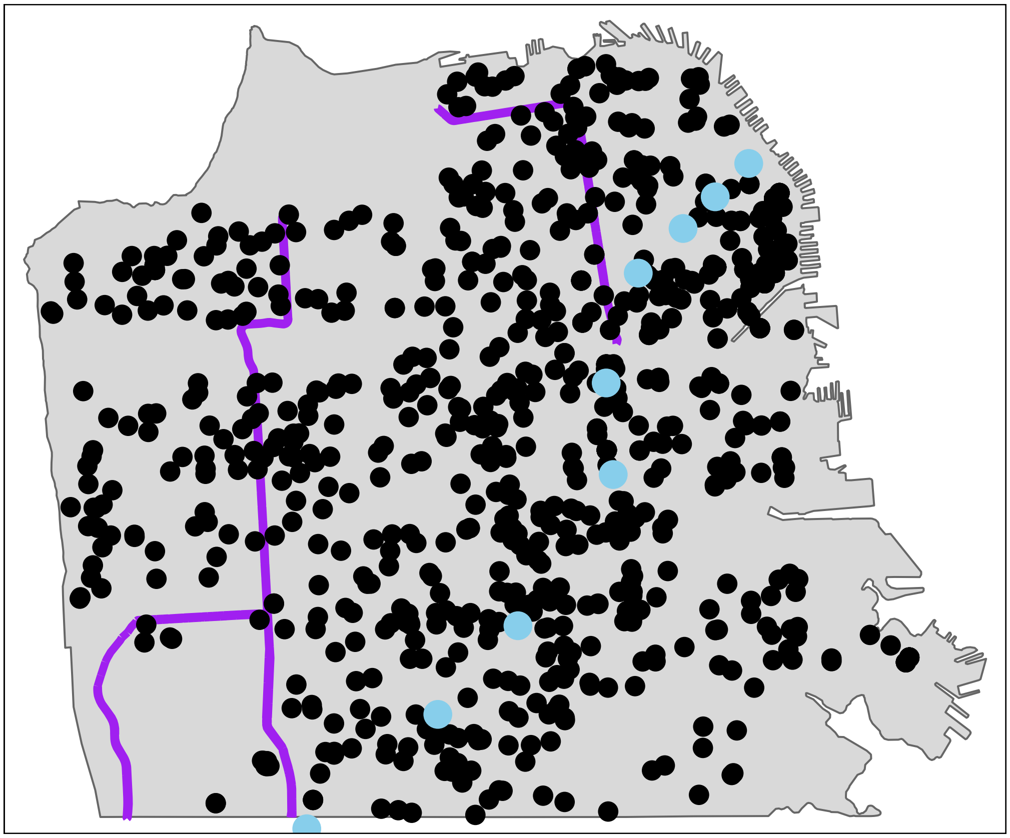
Create the same plot, as closely as possible, using `tmap`.

Challenge: Solution

```
challenge_map = tm_shape(SFboundary) +
  tm_polygons() +
tm_shape(SFhighways) +
  tm_lines(col = 'purple', lwd = 4) +
tm_shape(SFhomes15_sf) +
  tm_dots(col = 'black', size = 0.5) +
tm_shape(bart_utm) +
  tm_dots(col = 'skyblue', size = 1)
```

Challenge: Solution

```
challenge_map
```



Spatial Queries

Spatial Queries

There are two key types of spatial queries

- **spatial measurement** queries,
  - e.g. area, length, distance
- **spatial relationship** queries,
  - e.g. what locations in A are also in B.

These types are often combined, e.g.

- What is the area of region A that is within region B?

Spatial Measurement Queries

Computing Area

What is the area of San Francisco?

What data would we use to answer that question?

Area of San Francisco

- Use `sf::st_area` to compute the area of `sf` objects with polygons
- Check results against Wikipedia for SF

```r
sf_area = st_area(SFboundary_utm)
sf_area
```

```
## 119949901 [m^2]
```

Area of San Francisco

How did it manage to give us the units?

That comes from the `units` package, which `sf` imports and uses!

```r
class(sf_area)
```

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

```r
typeof(sf_area)
```

```
## [1] "double"
```

Area in sq km

Compare to the Wikipedia page's area for SF

```r
sf_area / (1000 * 1000) # Convert to square KM
```

```
## 119.9499 [m^2]
```

Area in sq km

That number is right, but now we've got an annoying little problem: Our value in square kilometers is labeled as square meters!

The `units` package, an `sf` dependency, provides a better way.

```r
library(units)
```

```
## udunits system database from /usr/share/udunits
```

```r
set_units(sf_area, km^2)
```

```
## 119.9499 [km^2]
```

Area in sq km

The function `valid_udunits` will give us a table of the valid units we could convert to:

(Note that the 'ud' comes from the `udunits` package, a dependency of the `units` package.)

```r
head(valid_udunits(), 2)
```

```
## udunits system database read from /usr/share/udunits
```

```
## # A tibble: 2 x 11
##   symbol symbol_aliases name_singular name_singular_a~ name_plural
##   <chr>  <chr>          <chr>         <chr>            <chr>
## 1 m      ""             meter         metre            ""
## 2 kg     ""             kilogram      ""               ""
## # ... with 6 more variables: name_plural_aliases <chr>, def <chr>,
## #   definition <chr>, comment <chr>, dimensionless <lgl>, source_xml <chr>
```

Area of San Francisco

What if we gave `st_area` the SF boundary in an unprojected CRS?

```
st_area(SFboundary)
```

Area of San Francisco

```
st_area(SFboundary)
```

```
## 120038745 [m^2]
```

`st_area` still gives us the measurement in a reasonable unit (rather than squared decimal degrees).

(However, this isn't a reason not to choose a reasonable, projected CRS for our data! Still best practice.

(Also notice the slight difference in our answers. This is not an equal-area projection!)

```
st_area(SFboundary_utm)
```

```
## 119949901 [m^2]
```

Length of highways

Use the function `st_length` to compute length of linear geometries.

```
st_length(SFhighways)
```

```
## Units: [m]
##   [1]  106.81285  106.81176  111.51019  111.50913  111.59378  111.59378
##   [7]  112.02125  112.01864  110.76642  111.31424  113.21844  113.63364
##  [13]  109.70803  109.79960  106.00554  106.00187  106.28392  106.29112
##  [19]  105.21542  105.22731  106.39351  106.38778  109.62289  109.62900
##  [25]  113.93788  113.93659   54.70773   55.80114   54.71353   55.79882
##  [31]   56.86037   55.79639   56.84993   55.80369   56.68567   54.90766
##  [37]   56.63758   54.94442   55.49930   56.47812  110.61722   53.84482
##  [43]   54.79581   53.63327   54.00083   53.39926   54.32512   53.38106
##  [49]   54.17142   54.12614   54.66743   53.40842   53.96184  109.13590
##  [55]   54.61882   54.37285   54.84968   54.77273  108.97194   54.33636
##  [61]   57.63089  110.55822  208.94243  209.05216   56.59410   53.67826
##  [67]  110.31555  108.50992  108.50737  140.98684   54.22959   53.02420
##  [73]   34.22720   71.54877   69.91093   69.90407  153.78462  214.54450
##  [79]   97.90823   98.13696  241.35294  240.57488  200.78614   93.35164
##  [85]   95.85929   83.63511  122.56650   79.05691   91.61146   81.18008
##  [91]   97.64588   84.83871   84.53116  138.24165  136.15610  272.46325
##  [97]   72.06197   77.57336   78.17869   43.96122   88.45841   79.16623
## [103]   30.10884   37.52394   76.91117   81.12105   53.90307   76.37346
## [109]   34.27302   80.96283   81.27529   51.27864  180.65158   80.83910
## [115]   77.97029   73.19650  219.33740  132.40054   82.67087  181.75832
## [121]   84.60513   95.72339  623.40101  624.13256 1250.25180  764.47517
## [127]  766.94192  160.89800  204.46082  428.26660  113.70954  212.18632
## [133]  103.59201   36.06458  117.74497   21.14108  132.30204  137.46452
## [139]  819.56269  816.21275  211.16053  209.84847  210.83762  210.30138
## [145]  210.89008  210.78982  217.65637  217.59643  216.63497  216.65846
## [151]  211.83011  211.42684  212.15515  209.11608   47.56151   53.09755
## [157]  153.68153  153.34954  153.01433  152.32205  150.89362  150.33399
## [163]  150.61101  150.63365  151.20176  151.20023  150.60030  150.57768
## [169]  150.84118  150.81207  150.91343  150.91135  150.84600  150.89499
```

```
## [175]  150.91640  150.97727  150.88674  150.88337  149.72000  149.73431
## [181]  376.11733   98.16757  154.78278   43.28478  338.17768   96.13831
## [187]  173.47910  195.88397   65.32746  167.10764  312.72440   70.90073
## [193]   71.23036  614.68089  618.12994  369.24483   80.26625  325.36115
## [199]  110.06959   74.52087   79.25213  518.91854  149.83566  231.09698
## [205]  202.09511  237.65820  230.24111  230.32594  385.18077   80.66169
## [211]  156.86509  156.81320  243.26701  243.50972  317.30484  317.20170
## [217]  210.81728  211.14779  210.19899  210.06277  210.59325  209.85169
## [223]  209.37551  209.47373  209.18052  209.08728  209.69002  209.74257
## [229]  209.45700  209.61743  209.42360  209.60218  209.60701  209.56462
## [235]  210.87848  210.80542  211.21313  212.05797  210.47259  210.41286
## [241]  210.54065  210.93940  209.85168  209.11913  212.40005  211.85840
```

Length of highways

Oh! We got the length of every segment, in meters.

How do we get the total length of highways, in km?

Challenge

Calculate the total length of SF highways in our dataset, in km.

Challenge: solution

```
tot_length = set_units(sum(st_length(SFhighways)), km)
tot_length
```

```
## 39.83624 [km]
```

Perimeter

We can also calculate the perimeter of polygons, should we need it (though this is implemented in the `lwgeom` package, an `sf` dependency, rather than in `sf` itself.)

```
perim = lwgeom::st_perimeter(tracts)
head(perim, 10)
```

```
## Units: [m]
##  [1] 3095.519 2771.519 4053.398 2960.833 5846.816 6509.614 2498.040
##  [8] 3030.683 3744.900 4150.271
```

Distance

The `st_distance` will return the min distance between two geometries.

Compute the distance in kilometers between Embarcadero & Powell St Bart stations

(**NOTE**: You can always spot-check on Google Maps.)

```
emb_pow_dist = st_distance(bart_utm[bart_utm$STATION == 'EMBARCADERO',],
                           bart_utm[bart_utm$STATION == 'POWELL STREET',])
emb_pow_dist = set_units(emb_pow_dist, km)
emb_pow_dist
```

```
## Units: [km]
##          [,1]
## [1,] 1.334997
```

Distance

Take note of the print-out. What's up with the `[1,]` and `[,1]` around the value?

`st_distance` is going to calculate a matrix of pairwise distances, by default! (We just happened to subset our `sf` object to two new objects, each with a single feature, i.e. row.)

Read the docs:

```
?st_distance
```

Challenge

That means we can easily calculate the distance between all SF properties and Embarcadero station. So go ahead and do that!

Challenge: solution

```
dist2emb <- st_distance(bart_utm[bart_utm$STATION == 'EMBARCADERO',],
                        SFhomes15_utm)
dist2emb <- set_units(dist2emb, km)

# check output
length(dist2emb)
```

```
## [1] 835
```

```
nrow(SFhomes15_utm)
```

```
## [1] 835
```

```
head(dist2emb, 10)
```

```
## Units: [km]
##  [1]  9.3560500  3.7753035  2.3817736 11.1305421  4.4098830  1.6601826
##  [7]  6.9005955  4.2245048 11.1512413  0.6239989
```

Challenge: solution

Different syntax, equivalent result:

You could just nest your calls, if you'd like.

```
dist2emb <- set_units(st_distance(bart_utm[bart_utm$STATION == 'EMBARCADERO',],
SFhomes15_utm), km)

# check output
head(dist2emb, 10)
```

```
## Units: [km]
##  [1]  9.3560500  3.7753035  2.3817736 11.1305421  4.4098830  1.6601826
##  [7]  6.9005955  4.2245048 11.1512413  0.6239989
```

Challenge: solution

Different syntax, equivalent result:

You could also use the 'tidy' syntax, if you're into that!

```
dist2emb <- st_distance(bart_utm[bart_utm$STATION == 'EMBARCADERO',],
                        SFhomes15_utm) %>% set_units(km)
# check output
head(dist2emb, 10)
```

```
## Units: [km]
##  [1]  9.3560500  3.7753035  2.3817736 11.1305421  4.4098830  1.6601826
##  [7]  6.9005955  4.2245048 11.1512413  0.6239989
```

Spatial Relationship Queries

Spatial Relationship queries

Spatial relationship queries compare the geometries of two spatial objects in the same coordinate space (CRS).

Some example relationships:

Spatial Relationship queries

There are many, often similar, functions to perform spatial relationship queries (can be confusing!).

These operations may return logical values, lists, matrices, dataframes, geometries or spatial objects

- you need to check what type of object is returned
- you need to check what values are returned to make sure they make sense

BART stations in SF?

This is a very common type of spatial query called a `point-in-polygon` query.

We can use the `st_within` function to answer this.

We'll start with the simplest question: **Are there BART stations in SF?**

We already know the answer, but let's see how it's done.

Are there any BART stations in SF?

What does it return by default?

```
bart_stations_in_sf <-st_within(bart_utm, SFboundary_utm)

head(bart_stations_in_sf)
```

```
## [[1]]
## integer(0)
##
## [[2]]
## integer(0)
##
## [[3]]
## integer(0)
##
## [[4]]
## integer(0)
##
```

```
## [[5]]
## integer(0)
##
## [[6]]
## integer(0)
```

BART stations in SF?

The docs for the function (?st_within) explain that it returns a sparse-matrix object by default. This is more efficient, but more complicated to work with. For our purposes, let's disable this behavior:

```r
bart_stations_in_sf <-st_within(bart_utm, SFboundary_utm, sparse=F)

head(bart_stations_in_sf)
```

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

BART stations in SF?

That's a bit more obvious! Looks like we got a logical value for each BART station.

Let's check the object's size:

```r
dim(bart_stations_in_sf)
```

```
## [1] 44  1
```

```r
dim(bart_utm)
```

```
## [1] 44  5
```

BART stations in SF?

So, to answer the simple question, we just need to know if there's at least one TRUE in that list.

```r
T %in% bart_stations_in_sf
```

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

Which Bart stations are in SF?

What about this question?

We can use the same output, but now leverage its station-by-station structure!

Challenge

Return the names of the BART stations that are within SF.

Challenge: solution

```r
bart_utm[bart_stations_in_sf, ]$STATION
```

```
## [1] "EMBARCADERO"            "MONTGOMERY STREET"
## [3] "POWELL STREET"          "CIVIC CENTER/ UN PLAZA"
## [5] "16TH STREET & MISSION"  "24TH STREET & MISSION"
## [7] "GLEN PARK"              "BALBOA PARK"
```

Which Bart stations are in SF?

And of course, there are multiple ways to do a thing!

We could also use the `st_intersection` function to get similar results.

```
sfbart_utm = st_intersection(bart_utm, SFboundary_utm)
```

```
## Warning: attribute variables are assumed to be spatially constant
## throughout all geometries
```

```
sfbart_utm
```

```
## Simple feature collection with 8 features and 7 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: 548723.7 ymin: 4175071 xmax: 553175.5 ymax: 4183045
## epsg (SRID):    26910
## proj4string:    +proj=utm +zone=10 +datum=NAD83 +units=m +no_defs
##                   STATION OPERATOR DIST CO          NAME Pop2010 Land_sqmi
## 30            EMBARCADERO     BART    4 SF San Francisco  805235     46.87
## 31      MONTGOMERY STREET     BART    4 SF San Francisco  805235     46.87
## 32          POWELL STREET     BART    4 SF San Francisco  805235     46.87
## 33 CIVIC CENTER/ UN PLAZA     BART    4 SF San Francisco  805235     46.87
## 34  16TH STREET & MISSION     BART    4 SF San Francisco  805235     46.87
## 35  24TH STREET & MISSION     BART    4 SF San Francisco  805235     46.87
## 36              GLEN PARK     BART    4 SF San Francisco  805235     46.87
## 37            BALBOA PARK     BART    4 SF San Francisco  805235     46.87
##                    geometry
## 30 POINT (553175.5 4183045)
## 31 POINT (552693.7 4182561)
## 32 POINT (552231.4 4182101)
## 33 POINT (551587.2 4181453)
## 34 POINT (551132.8 4179866)
## 35 POINT (551242.5 4178546)
## 36 POINT (549876.6 4176357)
## 37 POINT (548723.7 4175071)
```

Map the SF BART stations

```
tmap_mode("view")

tm_shape(SFboundary_utm) +
  tm_polygons(col="beige", border.col="black") +
tm_shape(sfbart_utm) +
  tm_dots(col="red")
```

Map the SF BART stations

Reset `tmap` to plot mode

```r
tmap_mode("plot")
```

```
## tmap mode set to plotting
```

st_within vs st_intersects vs st_intersection

Devil in the details...

`st_within` returns TRUE/FALSE, testing if one geometry is *completely* within another.

`st_intersects` returns TRUE/FALSE, testing if two geometries have any points in common.

`st_intersection` returns the geometry that intersects.

`st_within`, `st_intersects`, `st_intersection`, and friends

- These were just a couple examples of common geometric queries used in spatial analysis.
- These, and other similar operations are neatly summarized on this great `sf` cheatsheet (also available in the `./docs` subdirectory of our workshop repo):

SF Census Tracts

Let's consider the `SFhomes15_utm` data along with the *SF census tract* data that we saw on day 1.

However, we are going to work with another version of the tract data, one that includes the population for each tract.

Challenge

Read in the SF Census Tracts with pop data and call it `sftracts`

- The filename is `sftracts_wpop.shp`.
- The file is located in `./data`.

Then, create a population `choropleth map`.

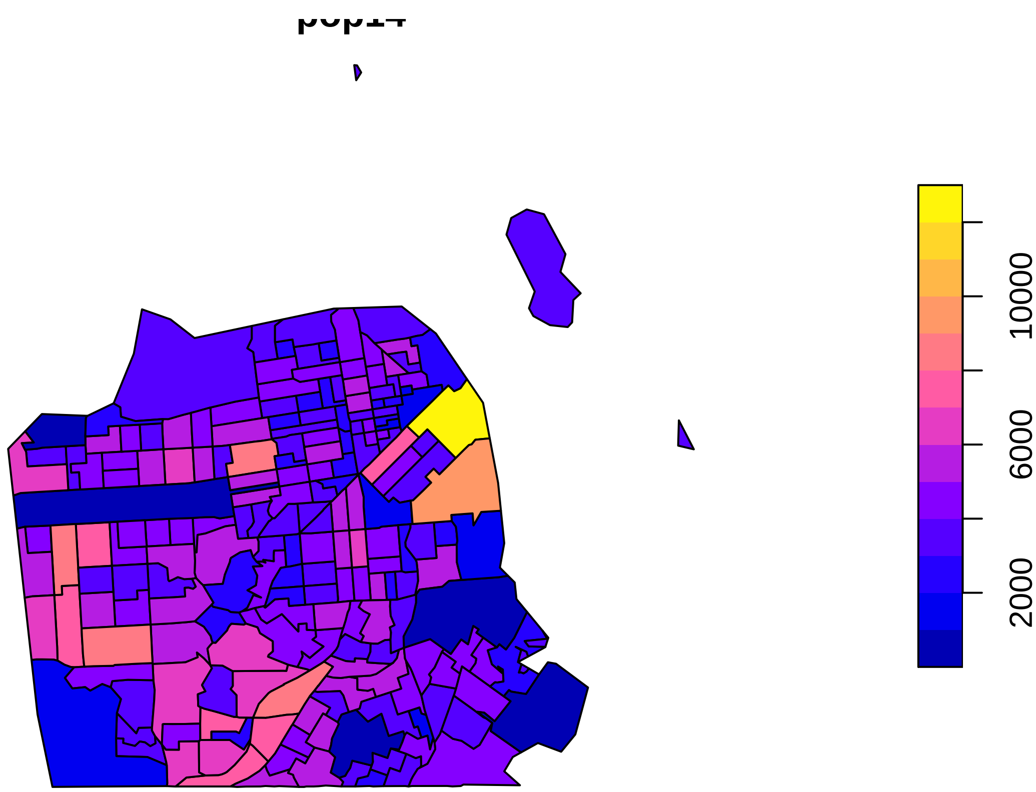
Challenge: solution

```r
#read in tracts
sftracts <- st_read("./data", "sftracts_wpop")
```

```
## Reading layer `sftracts_wpop' from data source `/home/drew/Desktop/stuff/berk/dlab/Geospatial-Fundame
## Simple feature collection with 195 features and 10 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -122.5145 ymin: 37.70813 xmax: -122.328 ymax: 37.86334
## epsg (SRID):    4269
## proj4string:    +proj=longlat +datum=NAD83 +no_defs
```

Challenge: solution

```r
#plot
plot(sftracts['pop14'])
```



Composite operations

Composite operations

The remaining material will work through some common spatial anlysis tasks.

Each workflow will feature some combination spatial measurement operations, spatial relationship operations, and other non-spatial operations.

Joins and Aggregation

Spatial join

A spatial join associates rows of data in one object with rows in another object based on the spatial relationship between the two objects.

A spatial join is based on the comparison of two sets of geometries in the same coordinate space.

- This is also called a **spatial overlay**.

Spatial join

We could use any of a family of spatial relationships that all return matrices of logical values.

`sf` refers to these as 'geometric binary predicates', and collects all their documentation into one document, which we've already seen:

```
?st_within
```

In what census tract is each property located?

We need to **spatially join** the `sftracts` and `SFhomes15_utm` to answer this.

What spatial object are we joining data from? to?

Spatial join

We have points, which are pretty much certain to be either inside or outside polygons. So we'll use `st_within` again as our spatial relationship.

We want to associate with each home the name of the census tract within which it falls.

So here goes...

In what census tract is each SF property located?

```
homes_with_tracts <- st_within(SFhomes15_utm, sftracts)
```

Did it work?

If not, why not?

CRSs must be the same

The `st_within` function, like almost all spatial analysis functions, requires that both data sets be spatial objects (they are) with the same coordinate reference system (CRS). Let's investigate

```
# What is the CRS of the property data?
st_crs(SFhomes15_utm)

# What is the CRS of the census tracts?
st_crs(sftracts)
```

Transform the CRS

```
#transform to UTM
sftracts_utm = st_transform(sftracts, st_crs(SFhomes15_utm))

# make sure the CRSs are the same
st_crs(sftracts_utm) == st_crs(SFhomes15_utm)
```

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

Now let's try that overlay operation again

Try 2

In what tract is each SF property is located?

```
homes_with_tracts <- st_within(SFhomes15_utm, sftracts_utm)
```

Review the st_within output

What is our output? Does it answer our question?

What type of data object did the over function return?

```
homes_with_tracts <- st_within(SFhomes15_utm, sftracts_utm)

class(homes_with_tracts)
length(homes_with_tracts)
```

```
nrow(sftracts_utm)
nrow(SFhomes15_utm)
```

Review the `st_within` output

What do we have here?

```
homes_with_tracts <- st_within(SFhomes15_utm, sftracts_utm)
class(homes_with_tracts)
```

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

```
length(homes_with_tracts)
```

```
## [1] 835
```

```
nrow(sftracts_utm)
```

```
## [1] 195
```

```
nrow(SFhomes15_utm)
```

```
## [1] 835
```

Review the `st_within` output

What the heck is an object of the class `sgbp`?

Read the docs!

(It's basically just a special sparse-matrix structure designed to hold the results returned from these binary-predicate functions.)

```
?sgbp
```

Review the `st_within` output

What data does the output object *store*?

```
head(homes_with_tracts)
```

```
## [[1]]
## [1] 92
##
## [[2]]
## [1] 38
##
## [[3]]
## [1] 193
##
## [[4]]
## [1] 34
##
## [[5]]
## [1] 153
##
## [[6]]
## [1] 64
```

Review the st_within output

We have a `list`, where each item's *index* is a `SFhomes15_utm` property's index, and each *value* is the index of the `sftracts_utm` census tract within which it is found.

We're halfway there!

Spatial join

We can now finish the operation by:

1. using that `st_within` output object to subset the `sftracts_utm` `data.frame`;
2. grabbing the desired columns from that subsetted `data.frame` and adding them to our `SFhomes15_utm` `data.frame`.

In our case, the desired column will just be the `GEOID` column (a standardized ID that we can then use to link up to non-spatial census data).

Add the GEOID column

CAUTION: this only works because the data are in the right order!

```
SFhomes15_utm$home_geoid <- sftracts_utm[unlist(homes_with_tracts),]$GEOID
```

Check the result

```
head(SFhomes15_utm, 2)
```

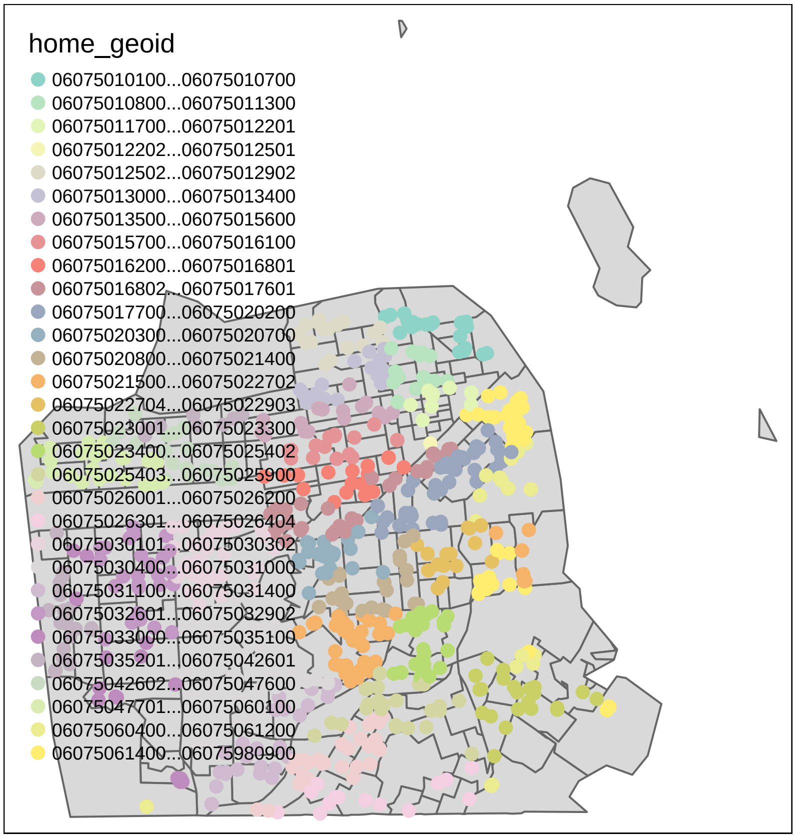
```
## Simple feature collection with 2 features and 18 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: 546340.7 ymin: 4176656 xmax: 553157.3 ymax: 4179269
## epsg (SRID):    26910
## proj4string:    +proj=utm +zone=10 +datum=NAD83 +units=m +no_defs
##    FiscalYear  SalesDate                             Address YearBuilt
## 24       2015 2015-08-21 0000 2760 19TH                AV0015      1979
## 35       2015 2015-08-13 0000 0560AMISSOURI            ST0000      2003
##    NumBedrooms NumBathrooms NumRooms NumStories NumUnits AreaSquareFeet
## 24           2            2        5          0        1           1595
## 35           2            2        5          1        1           1191
##    ImprovementValue LandValue       Neighborhood
## 24           432500    432500 West of Twin Peaks
## 35           701280    701280       Potrero Hill
##                                 Location SupeDistrict totvalue SalesYear
## 24 (37.7360097396496, -122.474067310226)            7   865000      2015
## 35  (37.759197817252, -122.396516184449)           10  1402560      2015
##                    geometry  home_geoid
## 24 POINT (546340.7 4176656) 06075030800
## 35 POINT (553157.3 4179269) 06075061400
```

Check the result

```
join_map = tm_shape(sftracts_utm) +
  tm_polygons() +
tm_shape(SFhomes15_utm) +
  tm_dots(col = 'home_geoid', size = 0.25)
```

Check the result

```
#Note that tmap bins our tracts because we have so many
join_map
```



WOW

Data linkage via space!

The `st_within` operation gave us the census tract data info for each point in `SFhomes15_utm`

We added the `GEOID` for each point to the `SFhomes15_utm` sf object.

We can now join `SFhomes15_utm` points by `GEOID` to any census variable, eg median household income, and then do an analysis of the relationship between, for example, property value and that variable.

How would we do that?

Attribute Joins

Attribute Joins

`Attribute joins` merge data in two tables based on matching data values contained in a column in each table.

For example we could join a table of student grades with a table of student names and addresses if both tables contain a column with student id.

Read in the census data

Let's read in a CSV file of median househould income for SF tracts.

The `sf_med_hh_income2015.csv` file only has two columns: `GEOID` and `medhhinc`.

Because `GEOIDs` can have leading zeros, we set the `colClasses` to make sure they are not stripped.

```
med_hh_inc <- read.csv("data/sf_med_hh_income2015.csv", stringsAsFactors = F,
                       colClasses = c("character","numeric"))

head(med_hh_inc)
```

```
##         GEOID medhhinc
## 1 06075980401        0
## 2 06075990100        0
## 3 06075012502    11925
## 4 06075012301    13909
## 5 06075061100    16545
## 6 06075980501    16638
```

Joining a regular `data.frame` to an `sf` `data.frame`

We can use `merge` to join the `med_hh_inc` DF to the `SFhomes15_utm` `sf` object.

We should make sure that they share a column of common values - GEOID / home_geoid

Joining a regular `data.frame` to an `sf` `data.frame`

Join two data objects based on common values in a column.

Use `merge` to join two `data.frames`.

(Notice, again, that our `sf data.frame` will conveniently behave just like regular old `data.frame` in this way.)

```
#make sure we're using `base` `merge` (because multiple other packages
#that you might have read in also have a `merge` function)
SFhomes15_utm <- base::merge(SFhomes15_utm,
                       med_hh_inc, by.x="home_geoid", by.y="GEOID")
```

Take a look at output

```
head(SFhomes15_utm, 2) # Look for the col medhhinc
```

```
## Simple feature collection with 2 features and 19 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: 551575.9 ymin: 4184223 xmax: 551672.2 ymax: 4184228
## epsg (SRID):    26910
## proj4string:    +proj=utm +zone=10 +datum=NAD83 +units=m +no_defs
##    home_geoid FiscalYear  SalesDate                              Address
## 1 06075010100       2015 2015-06-04 0000 0650 CHESTNUT            ST0204
## 2 06075010100       2015 2015-01-08 0592 0588 CHESTNUT            ST0000
##   YearBuilt NumBedrooms NumBathrooms NumRooms NumStories NumUnits
## 1      1995           2            2        0          0        1
## 2      1907           0            0       17          3        1
##   AreaSquareFeet ImprovementValue LandValue Neighborhood
```

```
## 1           1103           571078    571078  North Beach
## 2           3264           654836   1527951  North Beach
##                                   Location SupeDistrict totvalue SalesYear
## 1  (37.8039342366397, -122.41411670973)            3  1142156      2015
## 2 (37.8039733892367, -122.413021836652)            3  2182787      2015
##   medhhinc                  geometry
## 1    61442 POINT (551575.9 4184223)
## 2    61442 POINT (551672.2 4184228)
```

Check the `merge` results

```r
tmap_mode("view")
tm_shape(SFhomes15_utm) + tm_dots(col="medhhinc")
```

The Census Tract Perspective

We now know the census tract for each property.

Now let's think about this question from the tract perspective.

Let's ask the question

- What is the average propety value per tract?

Non-Spatial Aggregation

Since we joined GEOID to each property we can use the non-spatial `aggregate` function to compute the mean of totvalues for each GEOID.

But we'll use `sf`'s spatial implementation of aggregate.

We'll start by…

Reading the docs!

```r
?sf::aggregate.sf
```

sf::aggregate.sf

We see that we can provide arguments:

- `x`: `sf` object to be aggregated
- `by`: can be another `sf` object whose geometries will generate the groupings
- `FUN`: function to be used to summarize the grouped values

What is the mean home value in each census tract?

```r
tracts_with_mean_val <- aggregate(x = SFhomes15_utm["totvalue"],
                                  by = sftracts_utm,
                                  FUN = mean)
```

Wow, so simple. What does that give us?

Examine output of `sf::aggregate.sf`

```r
class(tracts_with_mean_val)
```

```
## [1] "sf"         "data.frame"
```

```r
head(tracts_with_mean_val, 2)
```

```
## Simple feature collection with 2 features and 1 field
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 551221.8 ymin: 4182036 xmax: 552338.1 ymax: 4184030
## epsg (SRID):    26910
## proj4string:    +proj=utm +zone=10 +datum=NAD83 +units=m +no_defs
##   totvalue                       geometry
## 1       NA MULTIPOLYGON (((551683.5 41...
## 2   482000 MULTIPOLYGON (((551221.8 41...
```

```r
nrow(tracts_with_mean_val) == nrow(sftracts_utm)
```

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

sf::aggregate.sf output

`sf::aggregate.sf` returned a new `sf data.frame`.

The new `data.frame` has the same geometry as `sftracts_utm`

But it only contains one column, with the mean `totvalue` for each tract.

To make these data more useful, let's add this value to `sftracts_utm`!

Note: This only works because there are the same number of elements in both `data.frame`s and they are in the same order!

```r
sftracts_utm$mean_totvalue <- tracts_with_mean_val$totvalue

head(sftracts_utm, 2) # check it
```

```
## Simple feature collection with 2 features and 11 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 551221.8 ymin: 4182036 xmax: 552338.1 ymax: 4184030
## epsg (SRID):    26910
## proj4string:    +proj=utm +zone=10 +datum=NAD83 +units=m +no_defs
##   STATEFP COUNTYFP TRACTCE             AFFGEOID       GEOID   NAME LSAD
## 1      06      075  010700 1400000US06075010700 06075010700    107   CT
## 2      06      075  012201 1400000US06075012201 06075012201 122.01   CT
##    ALAND AWATER pop14                       geometry mean_totvalue
## 1 183170      0  5311 MULTIPOLYGON (((551683.5 41...            NA
## 2  92048      0  4576 MULTIPOLYGON (((551221.8 41...        482000
```

Map it

Map the results to make sure they seem reasonable.

(**NOTE**: This is called a 'choropleth' map.)

```r
choropleth =
tm_shape(sftracts_utm) +
  tm_polygons(col="mean_totvalue", border.col=NA)
```

Map it

```
choropleth
```

Why no values for some tracts?

```
choropleth + tm_shape(SFhomes15_utm) + tm_dots(size = 0.01)
```

Proximity Analysis

Many methods of spatial analysis use distance to select features. For example...

What properties are within walking distance of BART?

In order to select properties with 1KM of BART, we can:

1. create a 1km-radius buffer polygon around each BART point
2. do a point-in-polygon operation to either count the number of properties within the buffer or compute mean values.

Create the buffers

For this, we'll use—surprise, suprise—`st_buffer`.

But first, we'll...

Read the docs!

```
?st_buffer
```

Create the buffers

It takes as input:

- **x**: an `sf*` object or objects to be buffered;
- **dist**: a buffer distance.

Create the buffers

Let's assume 1km is our 'standard walking distance'.

```
#remember: our units are meters!
bart_1km_buffer <- st_buffer(sfbart_utm, dist=1000)
```

Map the buffers

```
tm_shape(bart_1km_buffer) + tm_polygons(col="red") +
tm_shape(sfbart_utm) + tm_dots()
```

What properties are within 1km of a bart station?

What operation can we use here?

Once again, we can use `st_intersects` or `st_intersection`

What properties are within 1km of a bart station?

```
SFhomes_near_bart <-st_intersection(SFhomes15_utm, bart_1km_buffer)
```

```
## Warning: attribute variables are assumed to be spatially constant
## throughout all geometries
```

```
# Take a look
head(SFhomes_near_bart)
```

```
## Simple feature collection with 6 features and 26 fields
## geometry type:  POINT
## dimension:      XY
## bbox:           xmin: 552402.6 ymin: 4182488 xmax: 552895.5 ymax: 4183658
## epsg (SRID):    26910
## proj4string:    +proj=utm +zone=10 +datum=NAD83 +units=m +no_defs
##     home_geoid FiscalYear  SalesDate                             Address
## 19 06075010500       2015 2015-08-14 0000 0016 FRONT              ST0000
## 21 06075010500       2015 2015-09-08 0000 0733 FRONT              ST0707
## 24 06075010600       2015 2015-12-16 0000 0455 VALLEJO            ST0311
## 42 06075011700       2015 2015-04-17 0000 0690 MARKET             ST1905
## 43 06075011700       2015 2015-06-05 0000 0333 BUSH               ST3804
## 44 06075011700       2015 2015-03-12 0000 0333 BUSH               ST3904
##    YearBuilt NumBedrooms NumBathrooms NumRooms NumStories NumUnits
## 19      1986           3            3        7          2        1
## 21      2007           2            2        5          1        1
## 24      1973           0            0        3          0        1
## 42      2007           1            1        4          0        1
## 43      1987           2            2        5          0        1
## 44      1987           2            2        5          2        1
##    AreaSquareFeet ImprovementValue LandValue
## 19           2251          1475000   1475000
## 21           1378          1200000   1200000
## 24            825           437500    437500
## 42            952           477167    715751
## 43           1510           812200    812200
## 44           1510           761361    761361
##                   Neighborhood                              Location
## 19 Financial District/South Beach (37.7982533719796, -122.399172133445)
## 21 Financial District/South Beach (37.7980267794367, -122.400091849915)
## 24                   North Beach (37.7987906647799, -122.404766465894)
## 42 Financial District/South Beach (37.7882423441494, -122.403200587421)
## 43 Financial District/South Beach (37.7905798690843, -122.403108701388)
## 44 Financial District/South Beach (37.7905798690843, -122.403108701388)
##    SupeDistrict totvalue SalesYear medhhinc     STATION OPERATOR DIST CO
## 19            3  2950000      2015   105000 EMBARCADERO     BART    4 SF
## 21            3  2400000      2015   105000 EMBARCADERO     BART    4 SF
## 24            3   875000      2015    34808 EMBARCADERO     BART    4 SF
## 42            3  1192918      2015    34914 EMBARCADERO     BART    4 SF
## 43            3  1624400      2015    34914 EMBARCADERO     BART    4 SF
## 44            3  1522722      2015    34914 EMBARCADERO     BART    4 SF
##             NAME Pop2010 Land_sqmi                  geometry
## 19 San Francisco  805235     46.87 POINT (552895.5 4183601)
## 21 San Francisco  805235     46.87 POINT (552814.7 4183576)
## 24 San Francisco  805235     46.87 POINT (552402.6 4183658)
```

```
## 42 San Francisco  805235     46.87 POINT (552547.9 4182488)
## 43 San Francisco  805235     46.87 POINT (552554.4 4182748)
## 44 San Francisco  805235     46.87 POINT (552554.4 4182748)
```

Plot it

```r
tmap_mode('view')
```

```
## tmap mode set to interactive viewing
```

```r
tm_shape(bart_1km_buffer) + tm_borders(col="red") +
tm_shape(sfbart_utm) + tm_dots() +
tm_shape(SFhomes_near_bart) +
tm_dots(col = 'green', size = 0.03)
```

Any Questions?

Summary

That was a whirlwind tour of just some of the methods of spatial analysis.

There was of course a lot we didn't and can't cover.

Selected References & Tutorials

Here's that great `sf` cheatsheet (also available in the `./docs` subdirectory of this repo).

Introductory tutorials

- Spatial Data in R tutorial
- NEON Spatial Data tutorials
- GIS in R

Selected References & Tutorials

Emphasis on geodata visualization

- Tmap in a Nutshell
- Intro to visualizing Spatial Data in R
- RStudio Leaflet in R tutorial
- Blog on mapping census data in R

Selected references & tutorials

Deep dive Tutorials that include spatial analysis

- Geocomputation in R
- Intro to GIS and Spatial Analysis (see appendices)
- An Introduction to Spatial Data Analysis and Visualisation in R

CRAN Spatial Packages

- CRAN Task View: Analysis of Spatial Data