Session 6 - Spatial Data

R for Stata Users

Luiza Andrade, Rob Marty, Rony Rodriguez-Ramirez, Luis Eduardo San Martin, Leonardo Viotti
The World Bank – DIME | WB Github

March 2024

Table of contents

Overview of GIS conceps

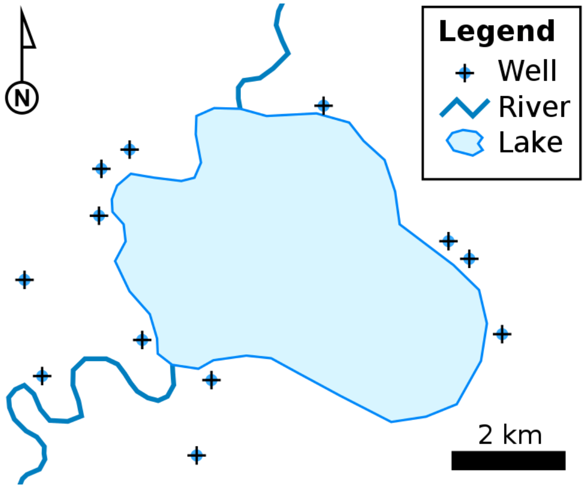
Spatial data: The two main types of spatial data are **vector data** and **raster data**

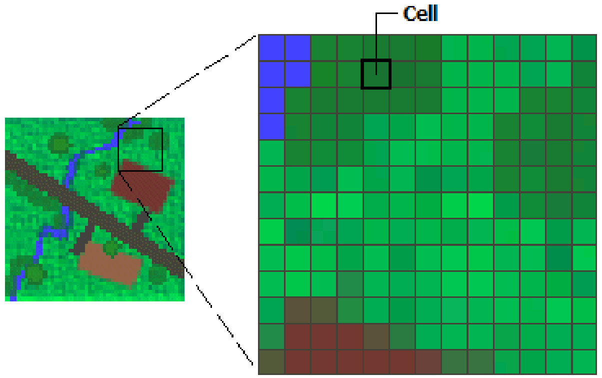
Vector data

- Points, lines, or polygons
- Common file formats include shapefiles (.shp) and geojsons (.geojson)
- Examples: polygons on countries, polylines of roads, points of schools

Raster data

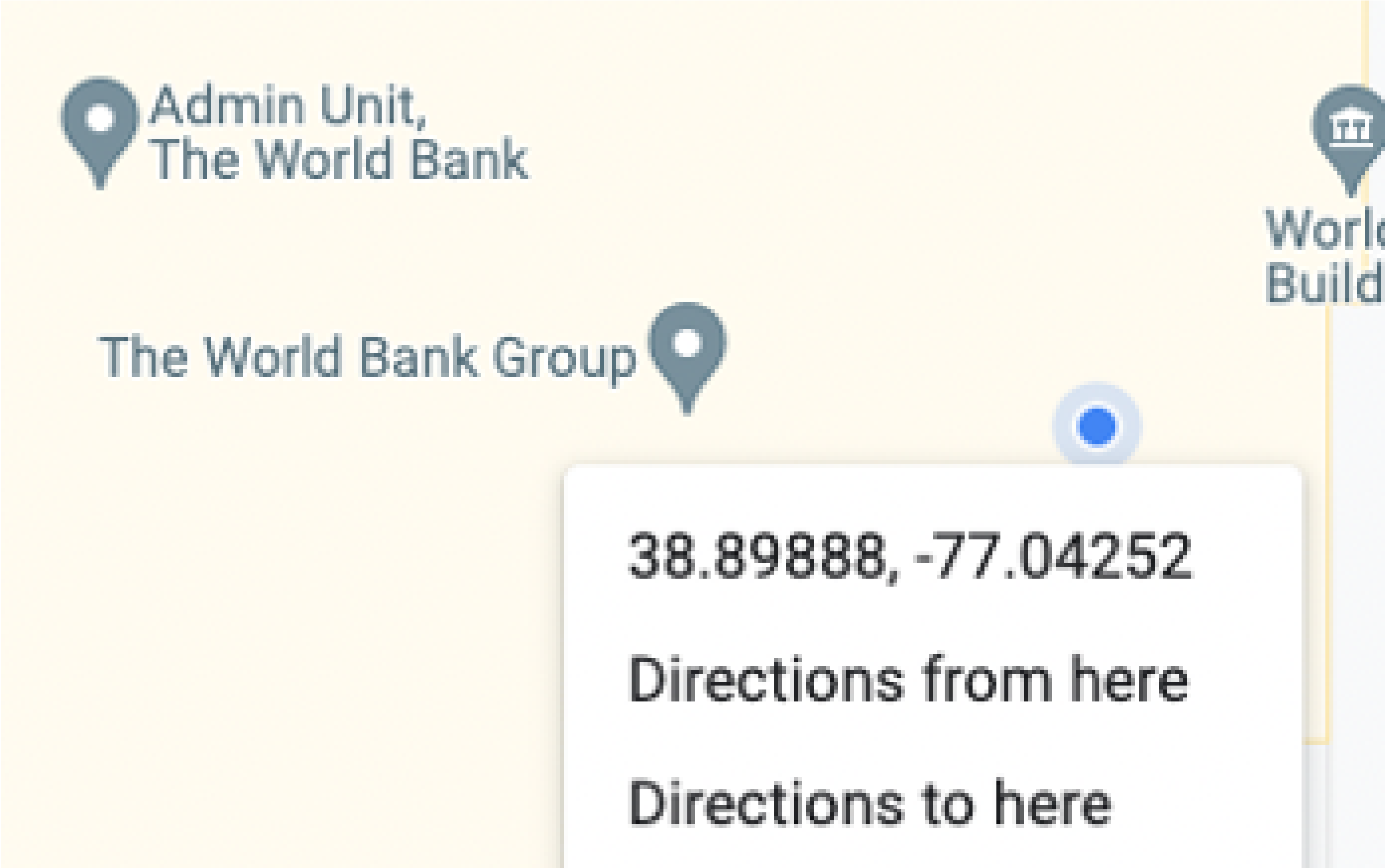
- Spatially referenced grid
- Common file format is a geotif (.tif)
- Example: Satellite imagery of nighttime lights





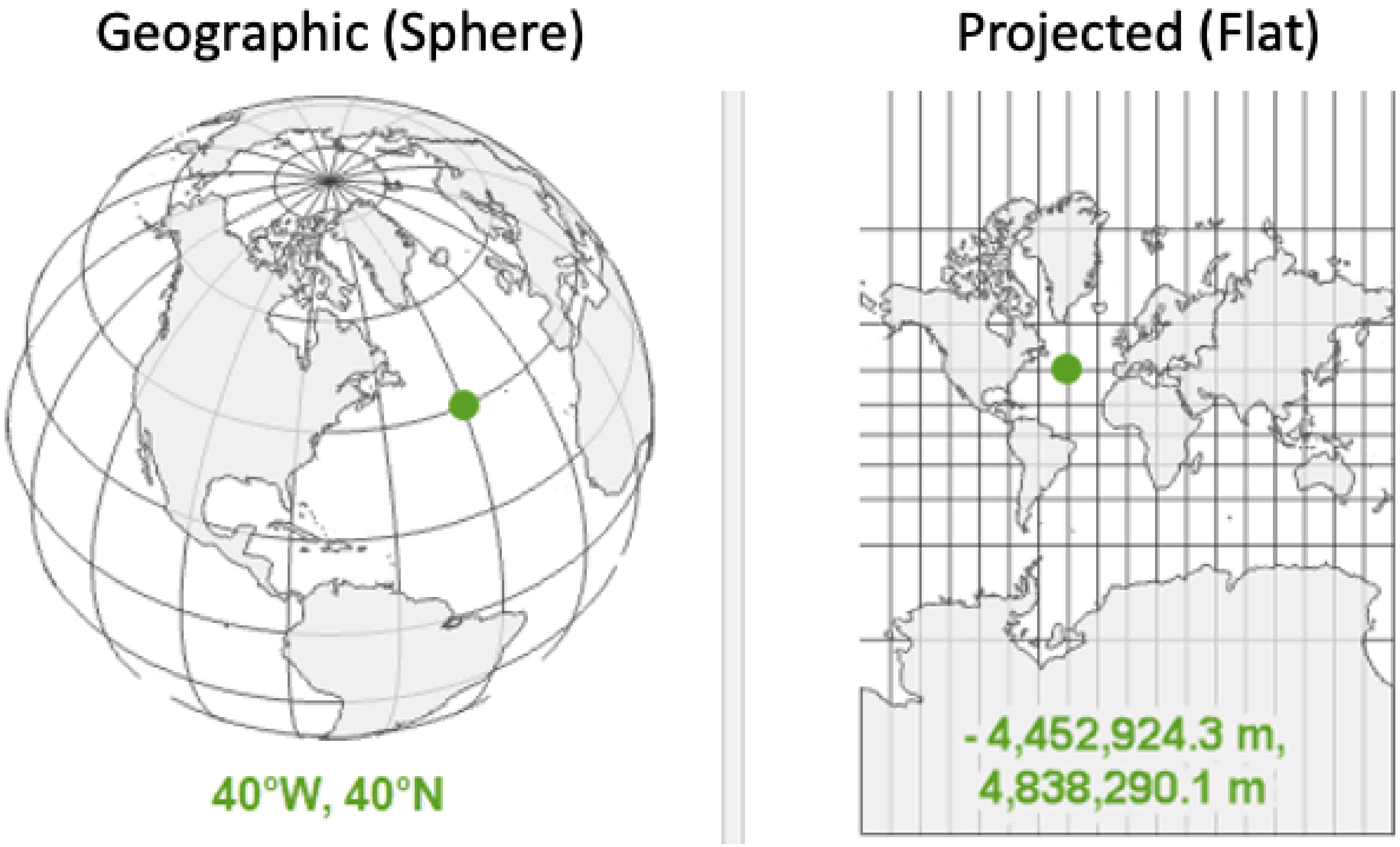
Coordinate Reference Systems (CRS)

- Coordinate reference systems use pairs of numbers to define a location on the earth
- For example, the World Bank is at a latitude of 38.89 and a longitude of -77.04



Coordinate Reference Systems (CRS)

There are many different coordinate reference systems, which can be grouped into **geographic** and **projected** coordinate reference systems. Geographic systems live on a sphere, while projected systems are “projected” onto a flat surface.

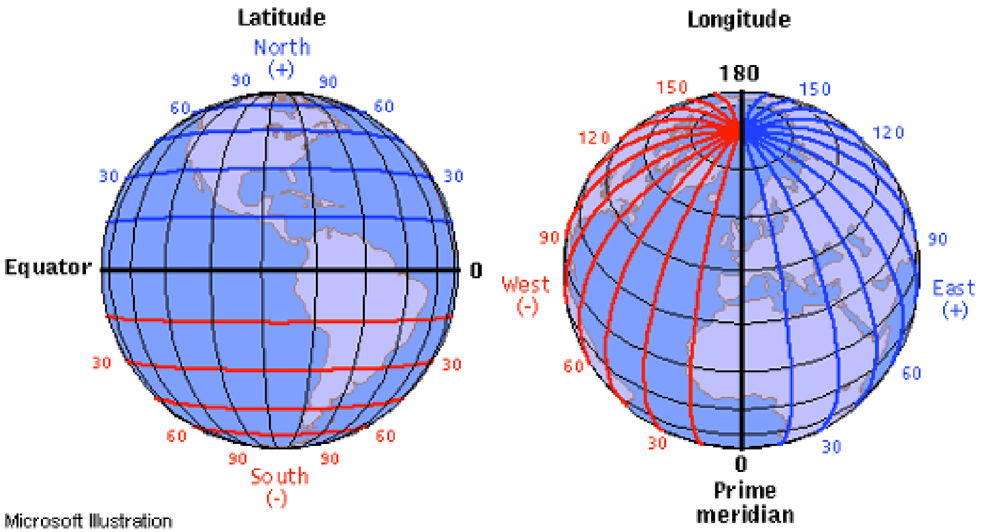


Geographic Coordinate Systems

Units: Defined by latitude and longitude, which measure angles and units are typically in decimal degrees. (Eg, angle is latitude from the equator).

Latitude & Longitude:

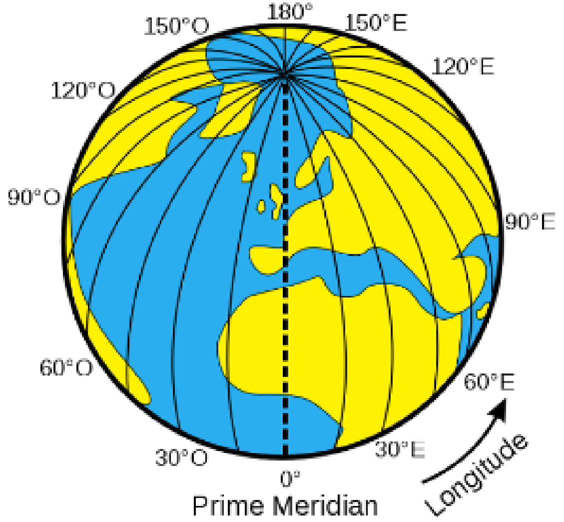
- On a grid X = longitude, Y = latitude; sometimes represented as (longitude, latitude).
- Also has become convention to report them in alphabetical order: (latitude, longitude) — such as in Google Maps.
- Valid range of latitude: -90 to 90
- Valid range of longitude: -180 to 180
- **{Tip}** Latitude sounds (and looks!) like latter.

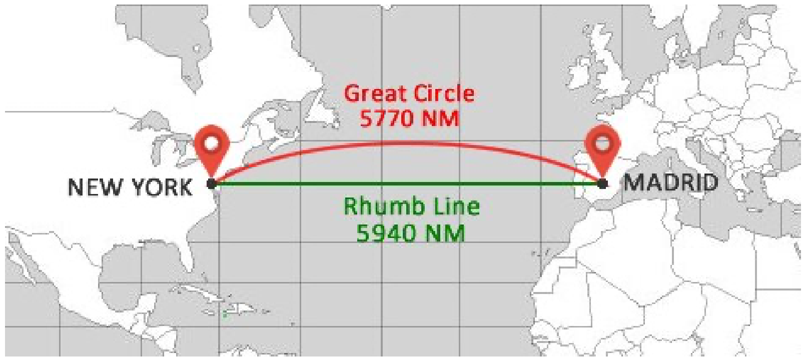


Geographic Coordinate Systems

Distance on a sphere

- At the equator (latitude = 0), a 1 decimal degree longitude distance is about 111km; towards the poles (latitude = -90 or 90), a 1 decimal degree longitude distance converges to 0 km.
- We must be careful (ie, use algorithms that account for a spherical earth) to calculate distances! The distance along a sphere is referred to as a great circle distance.
- Multiple options for spherical distance calculations, with trade-off between accuracy & complexity. (See distance section for details).



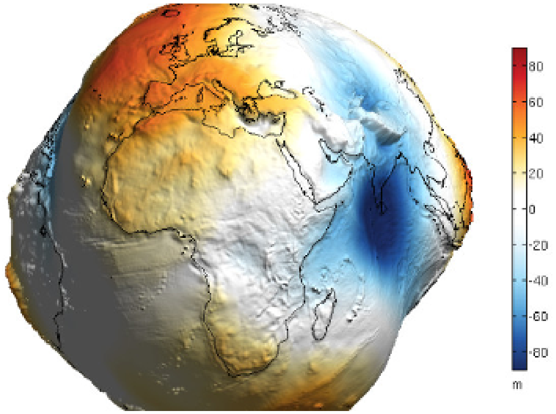


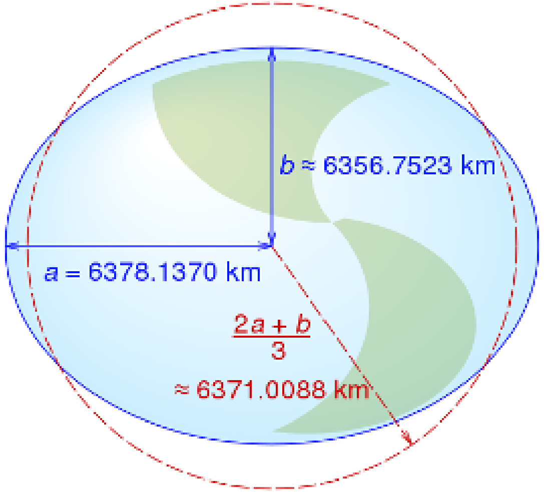
Datums

- **Is the earth flat?** No!
- **Is the earth a sphere?** No!
- **Is the earth a lumpy ellipsoid?** Yes!

The earth is a lumpy ellipsoid, a bit flattened at the poles.

- A datum is a model of the earth that is used in mapping. One of the most common datums is WGS 84, which is used by the Global Positional System (GPS).
- A datum is a reference ellipsoid that approximates the shape of the earth.
- Other datums exist, and the latitude and longitude values for a specific location will be different depending on the datum.





Projected Coordinate Systems

Projected coordinate systems project spatial data from a 3D to 2D surface.

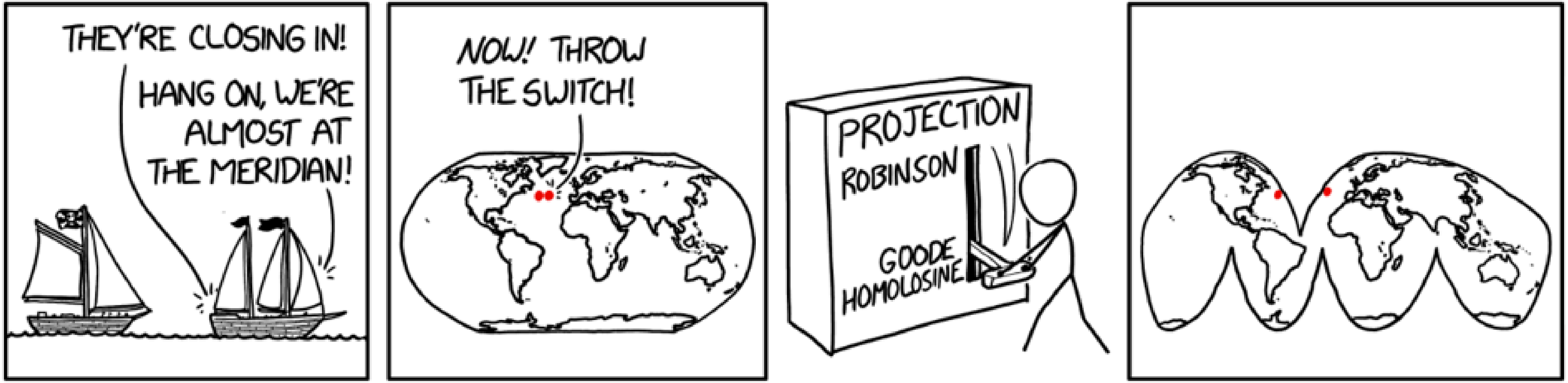
Distortions: Projections will distort some combination of distance, area, shape or direction. Different projections can minimize distorting some aspect at the expense of others.

Units: When projected, points are represented as "northings" and "eastings." Values are often represented in meters, where northings/eastings are the meter distance from some reference point. Consequently, values can be very large!

Datums still relevant: Projections start from some representation of the earth. Many projections (eg, UTM) use the WGS84 datum as a starting point (ie, reference datum), then project it onto a flat surface.

Click here to see why Toby & CJ are confused (hint: projections!)

Projected Coordinate Systems



Referencing coordinate reference systems

- There are many ways to reference coordinate systems, some of which are verbose.
- **PROJ** (Library for projections) way of referencing WGS84 `+proj=longlat +datum=WGS84 +no_defs +type=crs`
- **EPSG** Assigns numeric code to CRSs to make it easier to reference. Here, WGS84 is `4326`.

Coordinate Reference Systems

Whenever have spatial data, need to know which coordinate reference system (CRS) the data is in.

- You wouldn't say **"I am 5 away"**
- You would say **"I am 5 [miles / kilometers / minutes / hours] away"** (units!)
- Similarly, a "complete" way to describe location would be: I am at **6.51 latitude, 3.52 longitude using the WGS 84 CRS**

Introduction

- This session could be a whole course on its own, but we only have an hour and half.
- To narrow our subject, we will focus on only one type of spatial data, vector data.
- This is the most common type of spatial data that non-GIS experts will encounter in their work.
- We will use the `sf` package, which is the tidyverse-compatible package for geospatial data in R.
- For visualizing, we'll rely on `ggplot2` for static maps and `leaflet` for interactive maps

Setup

1. Copy/paste the following code into a new RStudio script, **replacing "YOURFOLDERPATHHERE" with the folder within which you'll place this R project**:

```
library(usethis)
use_course(
  url = "https://github.com/worldbank/dime-r-training/archive/main.zip",
  destdir = "YOURFOLDERPATHHERE"
)
```

2. In the console, type in the requisite number to delete the .zip file (we don't need it anymore).

3. A new RStudio environment will open. Use this for the session today.

Setup

Install new packages

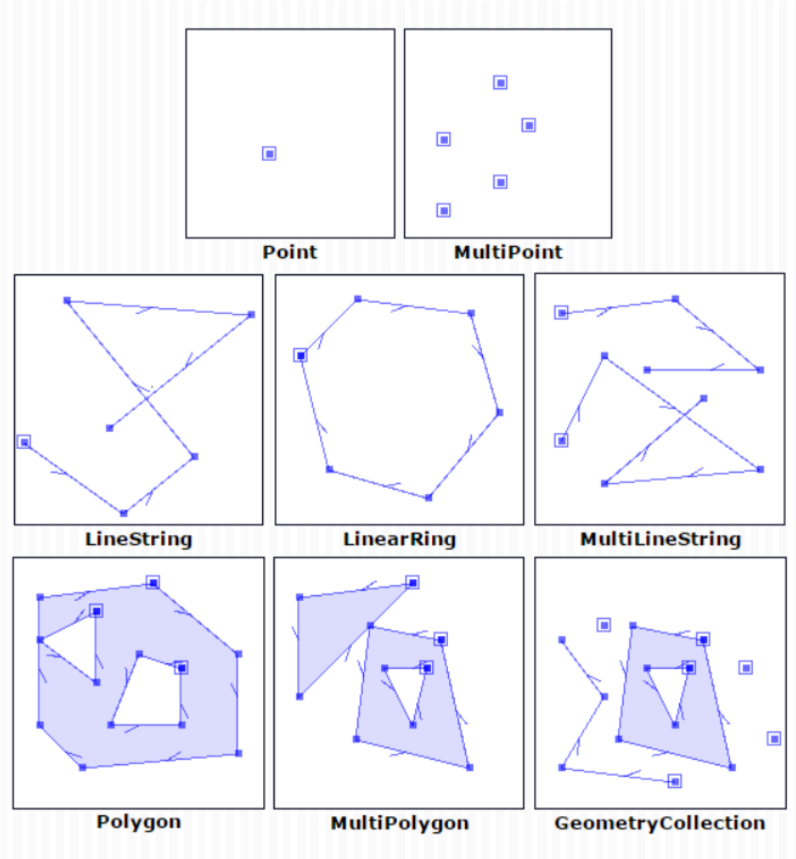
```
install.packages(c("sf",
                   "leaflet",
                   "geosphere"),
                 dependencies = TRUE)
```

And load them

```
library(here)
library(tidyverse)
library(sf)        # Simple features
library(leaflet)   # Interactive map
library(geosphere) # Great circle distances
```

Load and explore polylines, polylines, and points

The main package we'll rely on is the `sf` (simple features) package. With `sf`, spatial data is structured similarly to a **dataframe**; however, each row is associated with a **geometry**. Geometries can be one of the below types.



Load and explore polygon

The first thing we will do in this session is to recreate this data set:

```
country_sf <-
  st_read(here("DataWork",
               "DataSets",
               "Final",
               "country.geojson"))
```

```
## Reading layer `country' from data source
##   `/Users/robmarty/Documents/Github/dime-r-training/DataWork/DataSets/Final/country.geojson'
##   using driver `GeoJSON'
## Simple feature collection with 300 features and 13 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 33.90959 ymin: -4.720417 xmax: 41.92622 ymax: 5.061166
## Geodetic CRS:  WGS 84
```

Exploring the data

Look at first few observations

```
head(country_sf)
```

```
## Simple feature collection with 6 features and 13 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 35.52292 ymin: -0.198901 xmax: 36.29659 ymax: 0.990413
## Geodetic CRS:  WGS 84
##       GID_2 GID_0 COUNTRY   GID_1  NAME_1 NL_NAME_1          NAME_2 VARNAME_2
## 1 KEN.1.1_1   KEN   Kenya KEN.1_1 Baringo      <NA>             805      <NA>
## 2 KEN.1.2_1   KEN   Kenya KEN.1_1 Baringo      <NA> Baringo Central      <NA>
## 3 KEN.1.3_1   KEN   Kenya KEN.1_1 Baringo      <NA>   Baringo North      <NA>
## 4 KEN.1.4_1   KEN   Kenya KEN.1_1 Baringo      <NA>   Baringo South      <NA>
## 5 KEN.1.5_1   KEN   Kenya KEN.1_1 Baringo      <NA>   Eldama Ravine      <NA>
## 6 KEN.1.6_1   KEN   Kenya KEN.1_1 Baringo      <NA>         Mogotio      <NA>
##   NL_NAME_2       TYPE_2    ENGTYPE_2 CC_2 HASC_2
## 1      <NA> Constituency Constituency  162   <NA>
## 2      <NA> Constituency Constituency  159   <NA>
## 3      <NA> Constituency Constituency  158   <NA>
## 4      <NA> Constituency Constituency  160   <NA>
```

Exploring the data

Number of rows

```
nrow(country_sf)
```

```
## [1] 300
```

Exploring the data

Check coordinate reference system

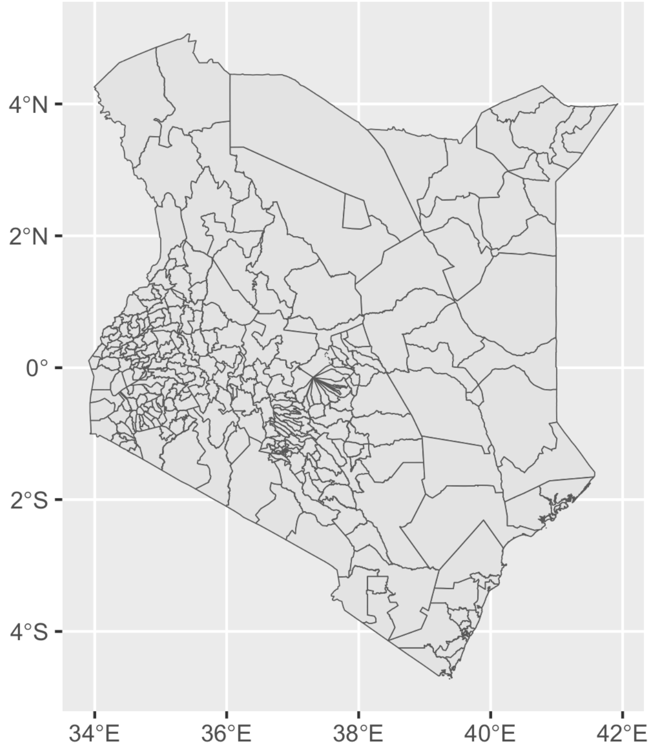
```
st_crs(country_sf)
```

```
## 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["geodetic latitude (Lat)",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["geodetic longitude (Lon)",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
##     ID["EPSG",4326]]
```

Exploring the data

Plot the data. To plot using `ggplot2`, we use the `geom_sf` geometry.

```
ggplot() +
  geom_sf(data = country_sf)
```



Attributes of data

We want the area of each location, but we don't have a variable for area

```
names(country_sf)
```

```
##  [1] "GID_2"     "GID_0"     "COUNTRY"   "GID_1"     "NAME_1"    "NL_NAME_1"
##  [7] "NAME_2"    "VARNAME_2" "NL_NAME_2" "TYPE_2"    "ENGTYPE_2" "CC_2"
## [13] "HASC_2"    "geometry"
```

Attributes of data

Determine area. Note the CRS is spherical (WGS84), but `st_area` gives area in meters squared. R uses s2 geomety for this.

```
st_area(country_sf)
```

```
## Units: [m^2]
##   [1]   174612548   664171032  1640235288  1893541200   909726196  1162391322
##   [7]  4446442673   244343146   319588761   546475821   791703635   487034695
##  [13]   347077424   234870080   320531683   178295354   302805234   267396640
##  [19]   950722974   208724111   377069605   235040170   166654802   311166355
##  [25]   244842707   270856300   195816268   234613620   250070745   300226047
##  [31]   544675636   890441628   819095221   562423505   473818437   774008353
##  [37]  1329273270   340468166  4248579388  7557295525 16712310204   603028994
##  [43]  8442761951  6483073596   270237144   247974418   626536513   259329880
##  [49]  1032562720   699140857   292011673  1147576864   124531035 15430646699
##  [55] 10122778443  4333824029  3283326674   112849413  6385975697  7975956018
##  [61]   210204075   144513355   145118575   102257259   162397218   424754069
##  [67]   278291541   175812288   138974871   258559234   412385494   246642948
##  [73]   447240937   317315791   755656427   354432319   466560197   244100322
##  [79]   214221442   184422278   336670907    59209638    79588059   106276706
##  [85]   172477093   468135485   283853034   204126728   232219565  2950195157
##  [91]   703903982   832269572   436237078  6970056180   614090663   201415848
```

Operations similar to dataframes

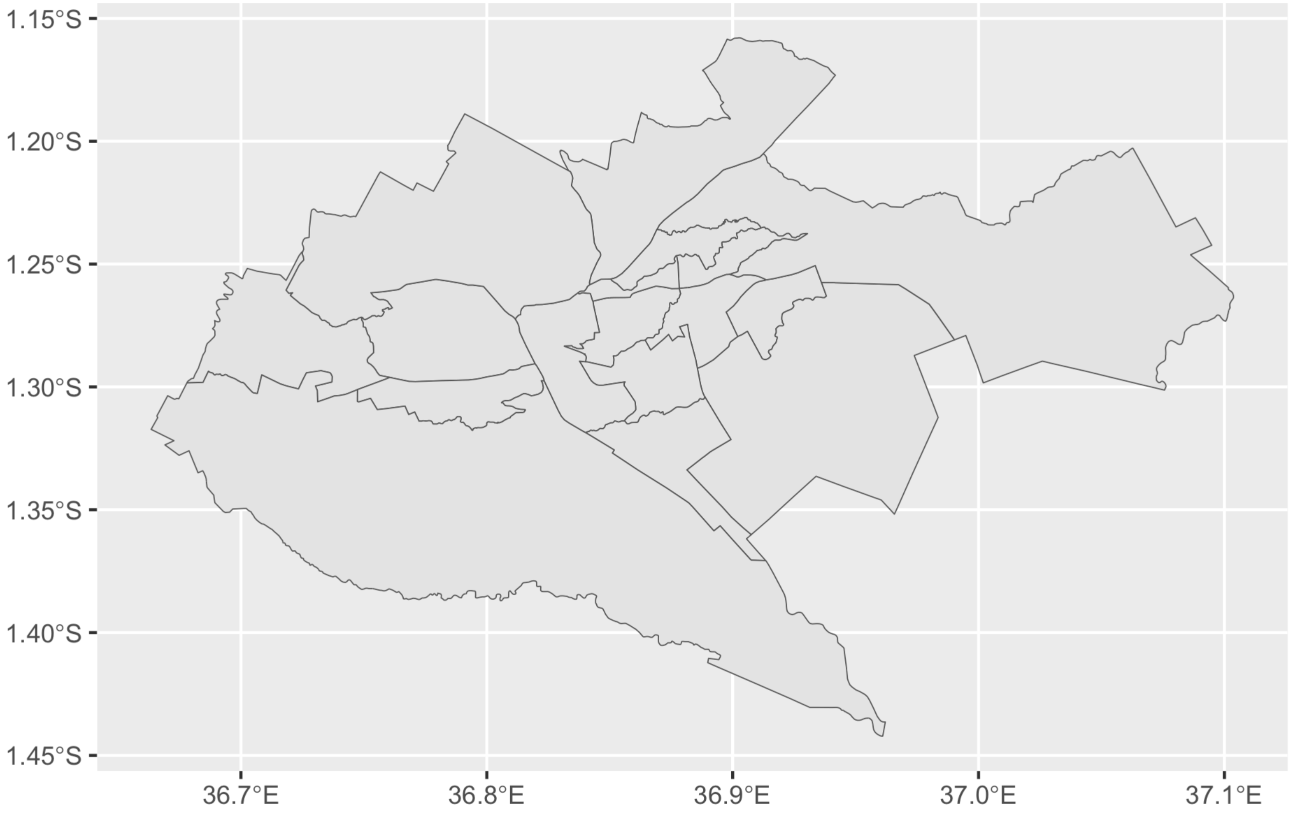
Create new dataset that captures locations for one administrative region

```r
city_sf <- country_sf %>%
  filter(NAME_1 == "Nairobi")
```

Operations similar to dataframes

Plot the dataframe

```
ggplot() +
  geom_sf(data = city_sf)
```



Load and explore polyline

Exercise:

- Load the roads data `roads.geojson` and name the object `roads_sf`
- Look at the first few observations
- Check the coordinate reference system
- Map the polyline

Solution:

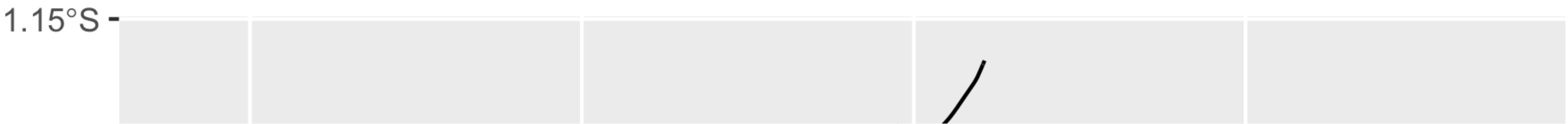
```
roads_sf <- st_read(here("DataWork", "DataSets", "Final", "roads.geojson"))

head(roads_sf)

st_crs(roads_sf)

ggplot() +
  geom_sf(data = roads_sf)
```

Load and explore polyline

```r
roads_sf <-
  st_read(here("DataWork",
               "DataSets",
               "Final",
               "roads.geojson"))
```

```
## Reading layer `roads' from data source
##   `/Users/robmarty/Documents/Github/dime-r-training/DataWork/DataSets/Final/roads.geojson'
##   using driver `GeoJSON'
## Simple feature collection with 3326 features and 3 fields
## Geometry type: MULTILINESTRING
## Dimension:     XY
## Bounding box:  xmin: 36.68034 ymin: -1.430759 xmax: 37.07664 ymax: -1.162558
## Geodetic CRS:  WGS 84
```

```r
ggplot() +
  geom_sf(data = roads_sf)
```



Load and explore polyline

Exercise: Determine length of each line (hint: use `st_length`)

Solution:

```
st_length(roads_sf)
```

```
## Units: [m]
##     [1]  901.5756687  137.8591461  166.0896978   24.2633581  174.3557406
##     [6]  482.5503262  486.2506610   64.1042587  615.4574212   16.7135334
##    [11]   19.6987155    3.8487318    4.1237787  555.4454885  551.5809255
##    [16]    7.0648280  229.5654887  588.9304516  136.9445835  579.1322038
##    [21]   58.3317564   21.7936436   90.1570913   41.2507165   81.7085529
##    [26]   68.3241690  496.3577265   14.1516418   44.0357348   45.8100172
##    [31]   41.6768610   35.1260485   40.5775920   43.1833377 1068.7247201
##    [36]  254.9862449  554.1882417  280.0103135  573.6525111  632.2063080
##    [41]  903.4320724   15.2156667   84.9916400   89.5853644   39.3628302
             656.4399014  562.9941828   53.0465233   55.0490837   22.1513814
             157.2766915  742.1361789  255.6841254  188.6658952  196.8677316
##    [56]  322.5741540  136.8794494  145.1362286 1123.8795488  449.7621126
```

01:00

Load and explore point data

We'll load a dataset of the location of schools

```
schools_df <-
  read_csv(here("DataWork",
                "DataSets",
                "Final",
                "schools.csv"))
```

```
## Rows: 3546 Columns: 5
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## chr (2): name, amenity
## dbl (3): osm_id, longitude, latitude
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Explore data

```
head(schools_df)
```

```
## # A tibble: 6 × 5
##       osm_id name                 amenity longitude latitude
##        <dbl> <chr>                <chr>       <dbl>    <dbl>
## 1   30312225 Consolata School     school       36.8    -1.27
## 2  674552830 <NA>                 <NA>         36.8    -1.26
## 3 1399125354 Galitos restaurant   school       36.8    -1.29
## 4 1764153756 Makini Schools       school       36.8    -1.30
## 5 1867185524 Bohra Primary School school       36.8    -1.26
## 6 2061462027 <NA>                 <NA>         36.8    -1.26
```

Explore data

```
names(schools_df)
```

```
## [1] "osm_id"    "name"      "amenity"   "longitude" "latitude"
```

Convert to spatial object

We define the (1) coordinates (longitude and latitude) and (2) CRS. **Note:** We must determine the CRS from the data metadata. This dataset comes from OpenStreetMaps, which uses EPSG:4326.

Assigning the incorrect CRS is one of the most common sources of issues I see with geospatial work. If something looks weird, check the CRS!

```
schools_sf <- st_as_sf(schools_df,
                       coords = c("longitude", "latitude"),
                       crs = 4326)
```

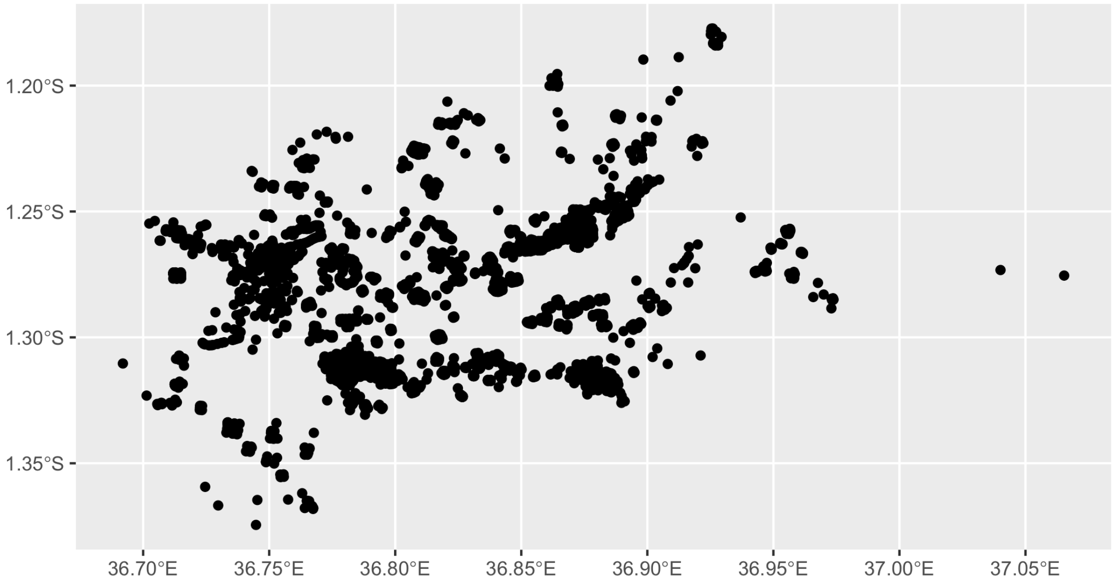
Convert to spatial object

```
head(schools_sf$geometry)
```

```
## Geometry set for 6 features
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 36.76877 ymin: -1.296051 xmax: 36.80406 ymax: -1.258515
## Geodetic CRS:  WGS 84
## First 5 geometries:

## POINT (36.80406 -1.267486)

## POINT (36.79734 -1.25969)

## POINT (36.77077 -1.290325)

## POINT (36.76877 -1.296051)

## POINT (36.79066 -1.258515)
```

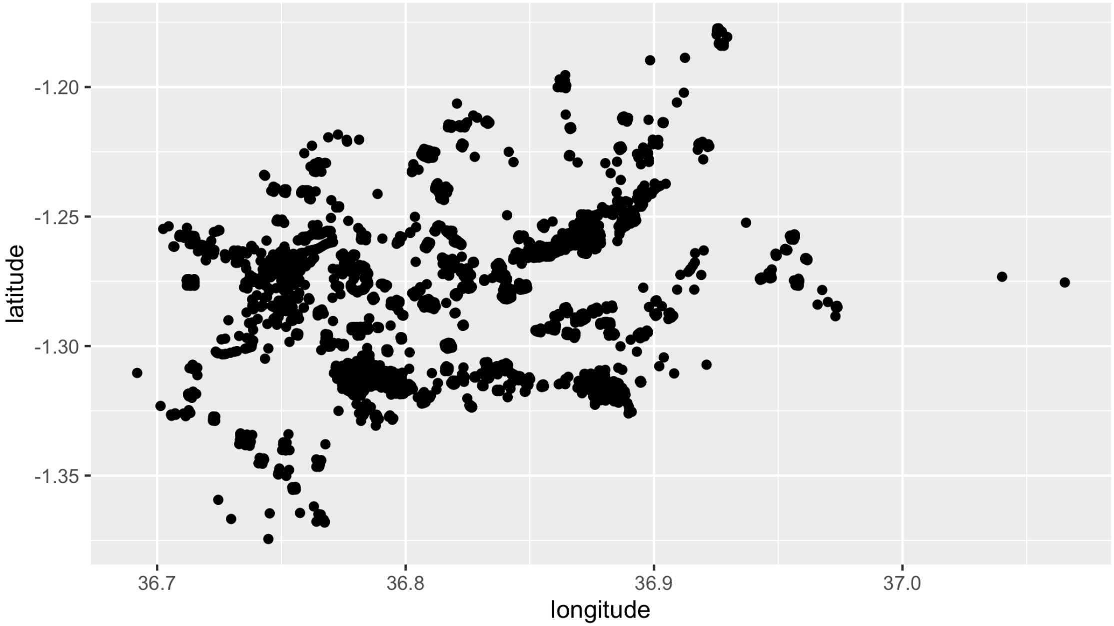
Map points object: Using sf

```
ggplot() +
  geom_sf(data = schools_sf)
```



Map points object: Using dataframe

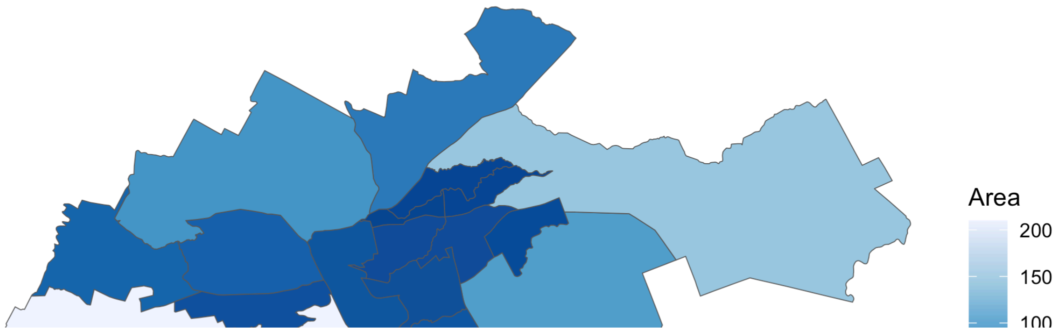
```
ggplot() +
  geom_point(data = schools_df,
             aes(x = longitude,
                 y = latitude))
```



Make better static map

Lets make a better static map.

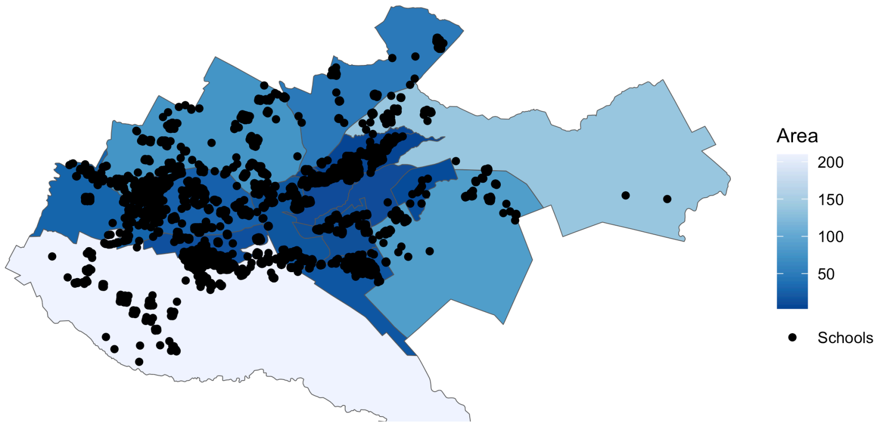
```
# Adding a variable with squared km
city_sf <- city_sf %>%
  mutate(area_m = city_sf %>% st_area() %>% as.numeric(),
         area_km = area_m / 1000^2)

# Plotting
ggplot() +
  geom_sf(data = city_sf,
          aes(fill = area_km)) +
  labs(fill = "Area") +
  scale_fill_distiller(palette = "Blues") +
  theme_void()
```



Make better static map

Lets add another spatial layer

```
ggplot() +
  geom_sf(data = city_sf,
          aes(fill = area_km)) +
  geom_sf(data = schools_sf,
          aes(color = "Schools")) +
  labs(fill = "Area",
       color = NULL) +
  scale_fill_distiller(palette = "Blues") +
  scale_color_manual(values = "black") +
  theme_void()
```



Another static map

Exercise: Make a static map of roads, coloring each road by its type. (**Hint:** The `highway` variable indicates the type).

Solution:

```
ggplot() +
  geom_sf(data = roads_sf,
          aes(color = highway)) +
  theme_void() +
  labs(color = "Road Type")
```

01:00

Another static map

```r
ggplot() +
  geom_sf(data = roads_sf,
          aes(color = highway)) +
  theme_void() +
  labs(color = "Road Type")
```

Interactive map

We use the `leaflet` package to make interactive maps. Leaflet is a JavaScript library, but the `leaflet` R package allows making interactive maps using R. Use of leaflet somewhat mimics how we use ggplot.

- Start with `leaflet()` (instead of `ggplot()`)
- Add spatial layers, defining type of layer (similar to geometries)

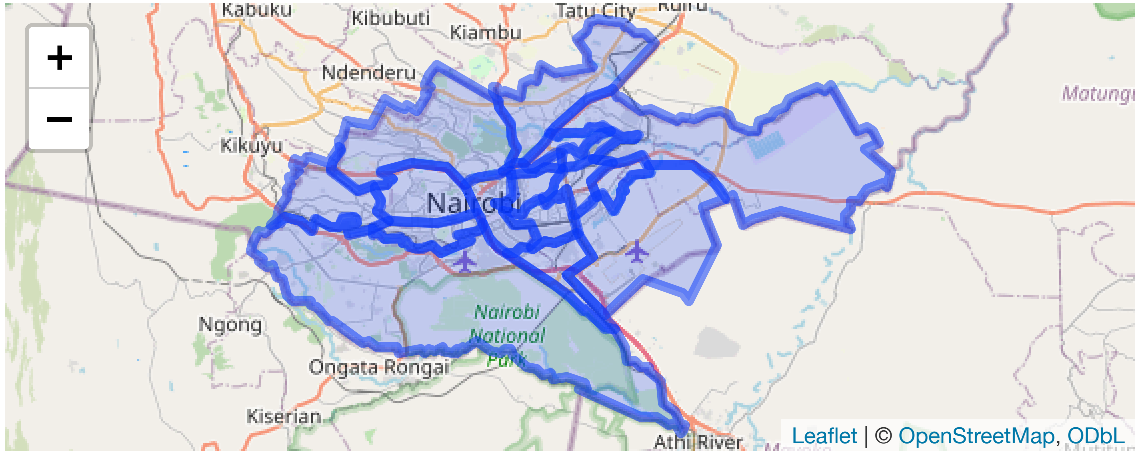
```
leaflet() %>%
  addTiles() # Basemap
```

Interactive map

We use the `leaflet` package to make interactive maps. Leaflet is a JavaScript library, but the `leaflet` R package allows making interactive maps using R. Use of leaflet somewhat mimics how we use ggplot.

- Start with `leaflet()` (instead of `ggplot()`)
- Add spatial layers, defining type of layer (similar to geometries)

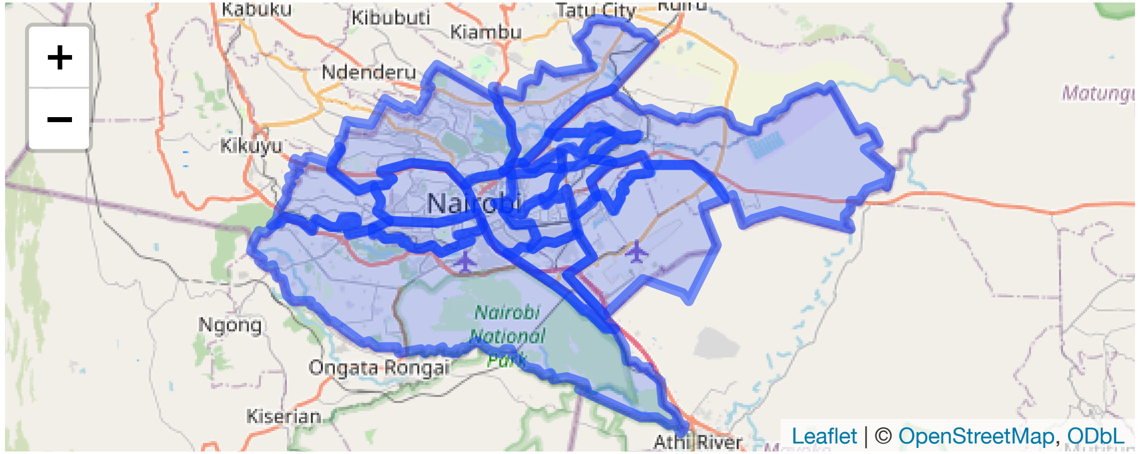
```
leaflet() %>%
  addTiles() %>%
  addPolygons(data = city_sf)
```



Interactive map

Add a pop-up

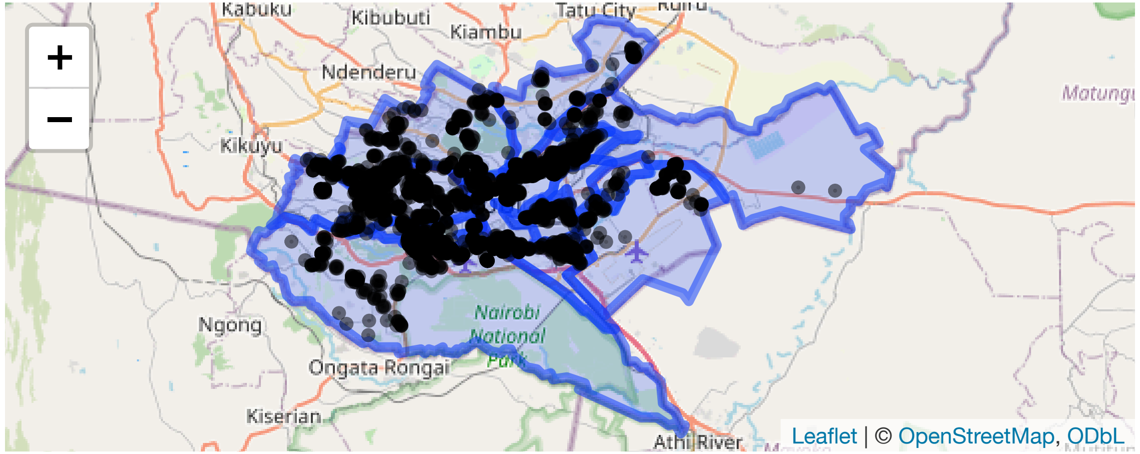
```
leaflet() %>%
  addTiles() %>%
  addPolygons(data = city_sf,
              popup = ~NAME_2)
```



Interactive map

Add more than one layer

```
leaflet() %>%
  addTiles() %>%
  addPolygons(data = city_sf,
              popup = ~NAME_2) %>%
  addCircles(data = schools_sf,
             popup = ~name,
             color = "black")
```



Interactive map of roads

Exercise: Create a leaflet map with roads, using the `roads_sf` dataset. (**Hint:** Use `addPolylines()`)

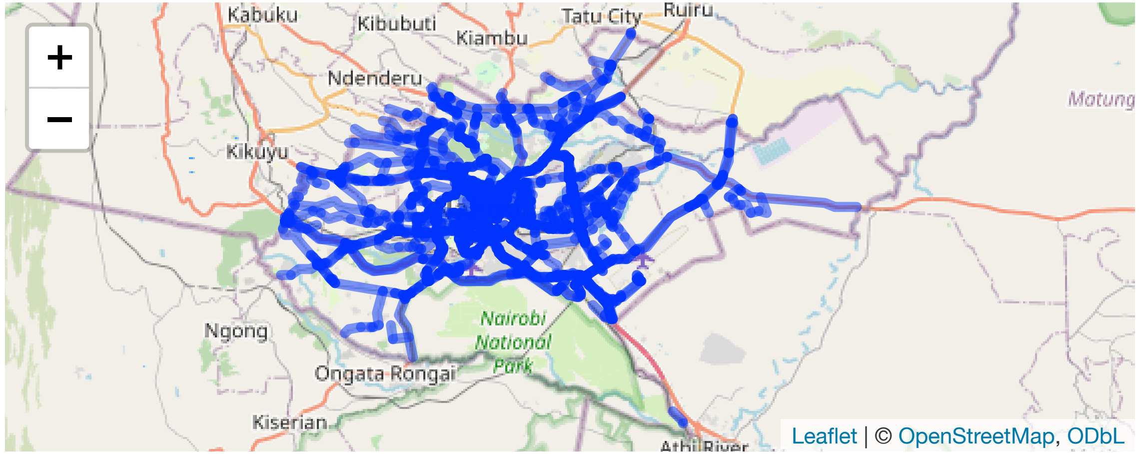
Solution:

```
leaflet() %>%
  addTiles() %>%
  addPolylines(data = roads_sf)
```

02:00

Interactive map of roads

```
leaflet() %>%
  addTiles() %>%
  addPolylines(data = roads_sf)
```



Interactive maps

We can spent lots of time going over what we can done with leaflet - but that would take up too much time. This resource provides helpful tutorials for things like:

- Changing the basemap
- Adding colors
- Adding a legend
- And much more!

Spatial operations applied on single dataset

- `st_transform`: Transform CRS
- `st_buffer`: Buffer point/line/polygon
- `st_combine`: Dissolve by attribute
- `st_convex_hull`: Create convex hull
- `st_centroid`: Create new sf object that uses the centroid
- `st_drop_geometry`: Drop geometry; convert from sf to dataframe
- `st_coordinates`: Get matrix of coordinates
- `st_bbox`: Get bounding box

Transform CRS

The schools dataset is currently in a geographic CRS (WGS84), where the units are in decimal degrees. We'll tranform the CRS to a projected CRS (EPSG:32632), and where the units will be in meters.

Note that coordinate values are large! Values are large because units are in meters. Large coordinate values suggest projected CRS; latitude is between -90 and 90 and longitude is between -180 and 180.

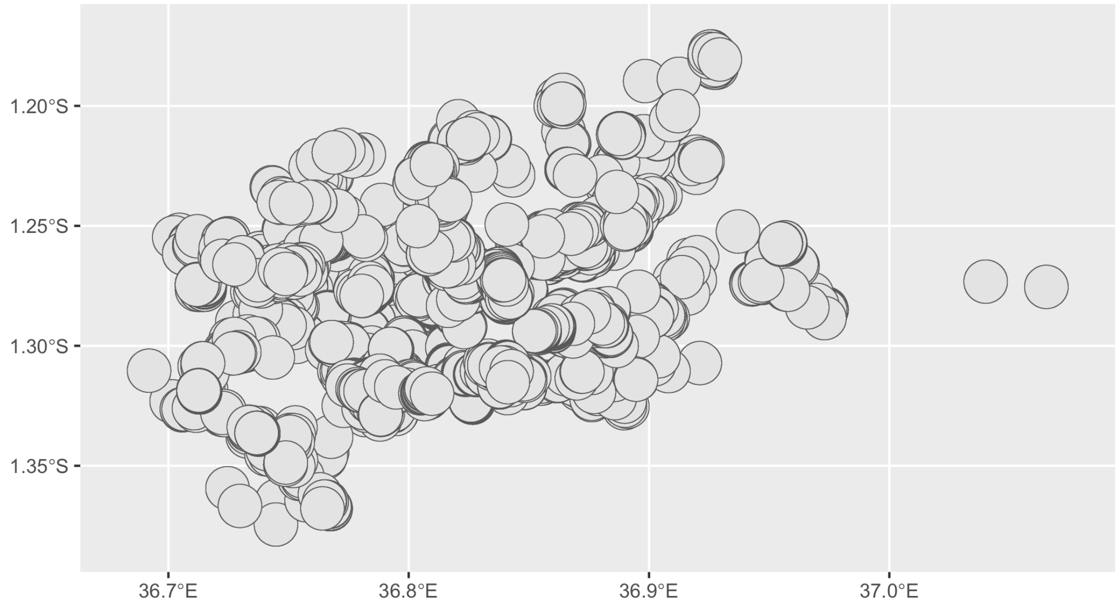
```
schools_utm_sf <- st_transform(schools_sf, 32632)

schools_utm_sf$geometry %>% head(2) %>% print()
```

```
## Geometry set for 2 features
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: 3722217 ymin: -158522.3 xmax: 3723051 ymax: -157537.6
## Projected CRS: WGS 84 / UTM zone 32N

## POINT (3723051 -158522.3)

## POINT (3722217 -157537.6)
```

Buffer

We have the points of schools. Now we create a 1km buffer around schools.

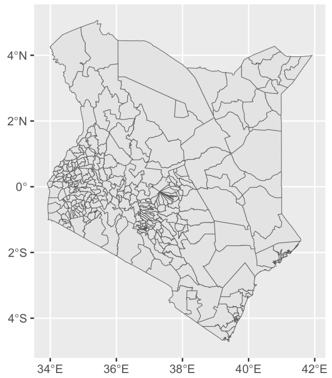
```
schools_1km_sf <- schools_sf %>%
  st_buffer(dist = 1000) # Units are in meters. Thanks s2!

ggplot() +
  geom_sf(data = schools_1km_sf)
```



Dissolve by an attribute

Below we have the second administrative regions. Using this dataset, let's create a new object at the first administrative region level.

```
country_1_sf <- country_sf %>%
  group_by(NAME_1) %>%
  summarise(geometry = st_combine(geometry)) %>%
  ungroup()

ggplot() +
  geom_sf(data = country_1_sf)
```



Exercise

Exercise: Create a polyline of all trunk roads (dissolve it using `st_combine`), and buffer the polyline by 10 meters. In `roads_sf`, the `highway` variable notes road types.

Solution:

```
roads_sf %>%
  filter(highway == "trunk") %>%
  summarise(geometry = st_combine(geometry)) %>%
  st_buffer(dist = 10)
```

```
## Simple feature collection with 1 feature and 0 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 36.68359 ymin: -1.430867 xmax: 37.07692 ymax: -1.204784
## Geodetic CRS:  WGS 84
##                         geometry
IPOLYGON (((36.76646 -1...
```

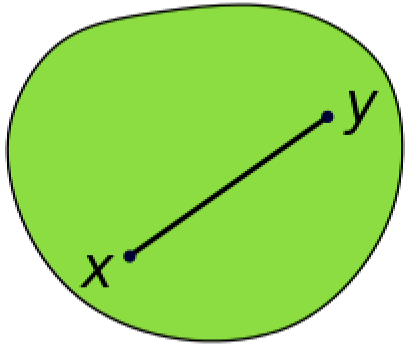
02:00

Convex Hull

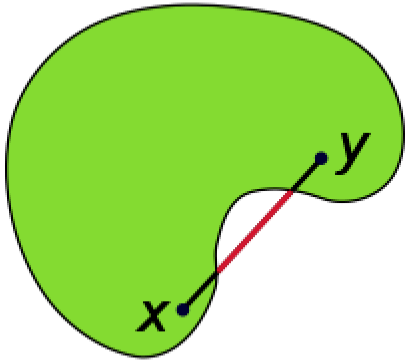
Simple definition: Get the outer-most coordinates of a shape and connect-the-dots.

Formal definition: A convex hull of a shape the smallest "convex set" that contains it. (A convex set is where a straight line can be drawn anywhere in the space and the space fully contains the line).

Convex



Not convex



Source: Wikipedia

Convex hull

In the below example, we create a conex hull around schools; creating a polygon that includes all schools.

Incorrect attempt

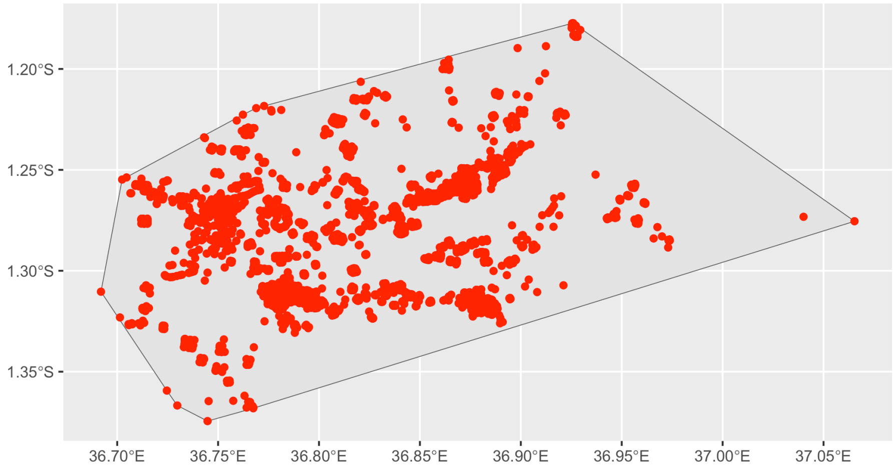
```
schools_chull1_sf <- schools_sf %>%
  st_convex_hull()

nrow(schools_chull1_sf)
```

```
## [1] 3546
```

Convex hull

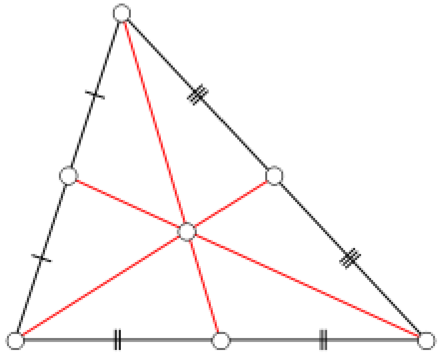
Correct

```
schools_chull2_sf <- schools_sf %>%
  summarise(geometry = st_combine(geometry)) %>%
  st_convex_hull()

ggplot() +
  geom_sf(data = schools_chull2_sf) +
  geom_sf(data = schools_sf, color = "red")
```



Determine centroid

Sometimes we want to represent a polygon or polyline as a single point. For this, we can compute the centroid (ie, geographic center) of a polygon/polyline.



Source: Wikipedia

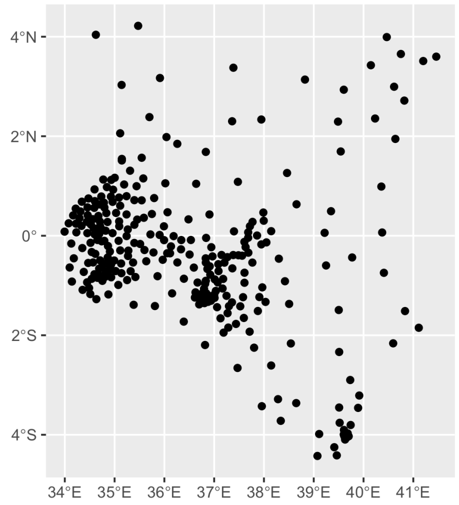
Determine centroid

Determine centroid of second administrative regions

```
country_c_sf <- st_centroid(country_sf)
```

```
## Warning: st_centroid assumes attributes are constant over geometries
```

```
ggplot() +
  geom_sf(data = country_c_sf)
```



Remove geometry

Incorrect approach

```
city_sf %>%
  select(-geometry) %>%
  head()
```

```
## Simple feature collection with 6 features and 15 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: 36.67803 ymin: -1.370704 xmax: 36.99025 ymax: -1.234921
## Geodetic CRS:  WGS 84
##        GID_2 GID_0 COUNTRY    GID_1  NAME_1 NL_NAME_1           NAME_2
## 1 KEN.30.1_1   KEN   Kenya KEN.30_1 Nairobi      <NA>  Dagoretti North
## 2 KEN.30.2_1   KEN   Kenya KEN.30_1 Nairobi      <NA>  Dagoretti South
## 3 KEN.30.3_1   KEN   Kenya KEN.30_1 Nairobi      <NA> Embakasi Central
## 4 KEN.30.4_1   KEN   Kenya KEN.30_1 Nairobi      <NA>    Embakasi East
## 5 KEN.30.5_1   KEN   Kenya KEN.30_1 Nairobi      <NA>   Embakasi North
## 6 KEN.30.6_1   KEN   Kenya KEN.30_1 Nairobi      <NA>   Embakasi South
##   VARNAME_2 NL_NAME_2       TYPE_2    ENGTYPE_2 CC_2 HASC_2   area_m   area_km
## 1      <NA>      <NA> Constituency Constituency  275   <NA> 26850519 26.850519
## 2      <NA>      <NA> Constituency Constituency  276   <NA> 28881788 28.881788
```

Remove geometry

Correct

```
city_sf %>%
  st_drop_geometry() %>%
  head()
```

```
##         GID_2 GID_0 COUNTRY    GID_1  NAME_1 NL_NAME_1          NAME_2
## 1 KEN.30.1_1   KEN   Kenya KEN.30_1 Nairobi      <NA>  Dagoretti North
## 2 KEN.30.2_1   KEN   Kenya KEN.30_1 Nairobi      <NA>  Dagoretti South
## 3 KEN.30.3_1   KEN   Kenya KEN.30_1 Nairobi      <NA> Embakasi Central
## 4 KEN.30.4_1   KEN   Kenya KEN.30_1 Nairobi      <NA>    Embakasi East
## 5 KEN.30.5_1   KEN   Kenya KEN.30_1 Nairobi      <NA>   Embakasi North
## 6 KEN.30.6_1   KEN   Kenya KEN.30_1 Nairobi      <NA>   Embakasi South
##   VARNAME_2 NL_NAME_2       TYPE_2    ENGTYPE_2 CC_2 HASC_2   area_m   area_km
## 1      <NA>      <NA> Constituency Constituency  275   <NA> 26850519 26.850519
## 2      <NA>      <NA> Constituency Constituency  276   <NA> 28881788 28.881788
## 3      <NA>      <NA> Constituency Constituency  284   <NA>  8249195  8.249195
## 4      <NA>      <NA> Constituency Constituency  285   <NA> 86236564 86.236564
## 5      <NA>      <NA> Constituency Constituency  283   <NA>  5451808  5.451808
## 6      <NA>      <NA> Constituency Constituency  282   <NA> 17635838 17.635838
```

Grab coordinates

Create a matrix of coordinates

```
schools_sf %>%
  st_coordinates() %>%
  head()
```

```
##              X         Y
## [1,] 36.80406 -1.267486
## [2,] 36.79734 -1.259690
## [3,] 36.77077 -1.290325
## [4,] 36.76877 -1.296051
## [5,] 36.79066 -1.258515
## [6,] 36.77899 -1.264575
```

Get bounding box

```
schools_sf %>%
  st_bbox()
```

```
##      xmin      ymin      xmax      ymax
## 36.691965 -1.374473 37.065336 -1.177316
```

Spatial operations using multiple datasets

- `st_distance`: Calculate distances.
- `st_intersects`: Indicates whether simple features intersect.
- `st_intersection`: Cut one spatial object based on another.
- `st_difference`: Remove part of spatial object based on another.
- `st_join`: Spatial join (ie, add attributes of one dataframe to another based on location).

Distances

For this example, we'll compute the distance between each school to a motorway.

```
motor_sf <- roads_sf %>%
  filter(highway == "motorway")

# Matrix: distance of each school to each motorway
dist_mat <- st_distance(schools_sf, motor_sf)

# Take minimun distance for each school
dist_mat %>% apply(1, min) %>% head()
```

```
## [1]   33.78464  155.32799 4006.16459 4662.68796  176.10524 1382.28513
```

Exercise

Exercise: Calculate the distance from the centroid of each second administrtaive division to the nearest trunk road.

Solution:

```
city_cent_sf <- city_sf %>% st_centroid()
```

```
## Warning: st_centroid assumes attributes are constant over geometries
```

```
trunk_sf <- roads_sf %>%
  filter(highway == "trunk")

# Matrix: distance of each school to each motorway
dist_mat <- st_distance(city_cent_sf, trunk_sf)

# Take minimun distance for each school
dist_mat %>% apply(1, min) %>% head()
```

02:00

```
27.34583 2215.07338   929.39785 5642.60850 2015.55906   19.97698
```

Distances

```r
# s2
st_distance(schools_sf[1,], schools_sf[2,]) %>%
  as.numeric()
```

```
## [1] 1144.271
```

```r
# Nigeria-specific CRS
schools_utm_sf <- st_transform(schools_sf, 32632)
st_distance(schools_utm_sf[1,], schools_utm_sf[2,]) %>
  as.numeric()
```

```
## [1] 1290.671
```

```r
# World mercator
schools_merc_sf <- st_transform(schools_sf, 3395)
st_distance(schools_merc_sf[1,], schools_merc_sf[2,])
  as.numeric()
```

```r
# Haversine
distHaversine(
  p1 = schools_sf[1,] %>% st_coordinates,
  p2 = schools_sf[2,] %>% st_coordinates)
```

```
## [1] 1145.551
```

```r
# Vincenty's method
distVincentySphere(
  p1 = schools_sf[1,] %>% st_coordinates,
  p2 = schools_sf[2,] %>% st_coordinates)
```

```
## [1] 1145.551
```

```r
# Karney's method
distGeo(p1 = schools_sf[1,] %>% st_coordinates,
        p2 = schools_sf[2,] %>% st_coordinates)
```

```
## [1] 1141.16
```

Intersects

For this example we'll determine which second administrative divisions intersects with a motorway.

```r
# Sparse matrix
st_intersects(city_sf, motor_sf) %>% print()
```

```
## Sparse geometry binary predicate list of length 17, where the predicate
## was `intersects'
## first 10 elements:
##  1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
##  2: (empty)
##  3: (empty)
##  4: 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, ...
##  5: (empty)
##  6: 30, 32, 33, 34, 35, 36, 37, 38, 39, 40, ...
##  7: (empty)
##  8: (empty)
##  9: (empty)
##  10: 42, 43, 45, 64
```

Intersects

Take `max` (`FALSE` corresponds to 0 and `TRUE` corresponds to 1). So taking max will yeild if unit intersects with *any* motorway

```
# Matrix
st_intersects(city_sf, motor_sf, sparse = F) %>%
  apply(1, max) %>%
  head()
```

```
## [1] 1 0 0 1 0 1
```

Exercise

Exercise: Determine which motorways intersect with a trunk road

Solution:

```
trunk_sf <- roads_sf %>% filter(highway == "trunk")
motor_sf <- roads_sf %>% filter(highway == "motorway")

st_intersects(motor_sf, trunk_sf, sparse = F) %>%
    apply(1, max) %>%
    head()
```

02:00

Intersection

We have roads for the full city. Here, we want to create new roads object that **only includes** roads in one unit.

```
loc_sf <- city_sf %>%
  head(1)

roads_loc_sf <- st_intersection(roads_sf, loc_sf)
```

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

```
ggplot() +
  geom_sf(data = roads_loc_sf)
```

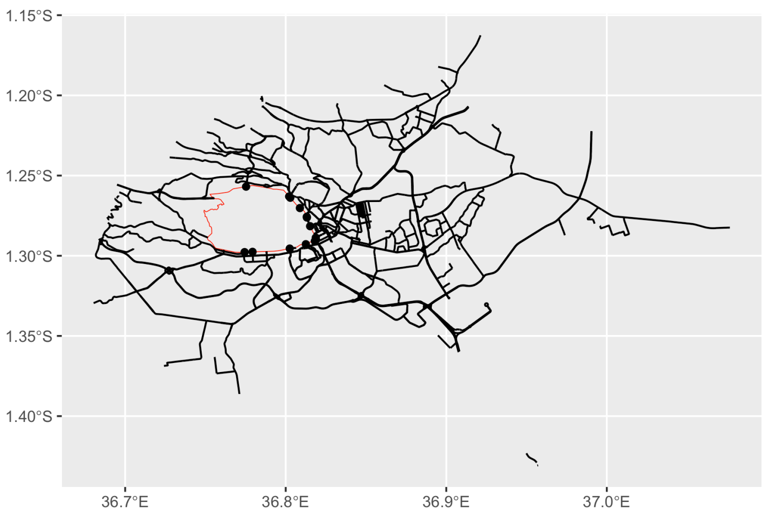
Difference

We have roads for all of the city. Here, we want to create new roads object that **excludes** roads in one unit.

```
roads_notloc_sf <- st_difference(roads_sf, loc_sf)
```

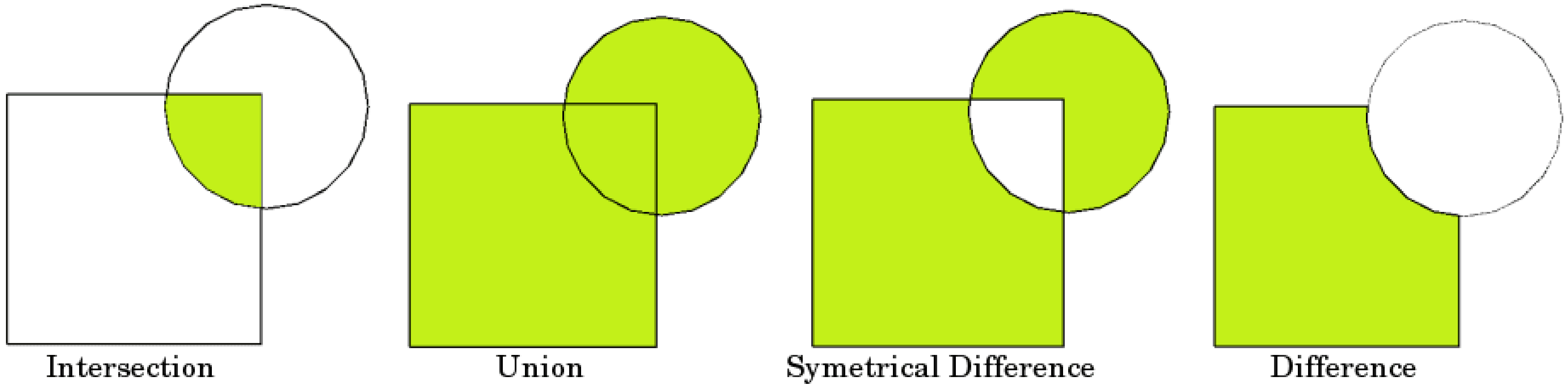
```
## Warning: attribute variables are assumed to be spatially constant throughout
## all geometries
```

```
ggplot() +
  geom_sf(data = loc_sf, fill = NA, color = "red") +
  geom_sf(data = roads_notloc_sf)
```



Overlay

Intersections and differencing are **overlay** functions



Exercise

Exercise: Create a map of schools that are within 1km of a motorway.

Solution:

```
motor_1km_sf <- roads_sf %>%
  filter(highway == "motorway") %>%
  st_buffer(dist = 1000)

schools_nr_motor_sf <- schools_sf %>%
  st_intersection(motor_1km_sf)

leaflet() %>%
  addTiles() %>%
  addCircles(data = schools_nr_motor_sf)
```

02:00

Exercise

Note that there are multiple approaches we could have used for creating a map of schools that are within 1km of a trunk road.

1. Buffer trunk roads by 1km and do a spatial intersection with schools
2. Calculate the distance of each school to the nearest trunk road, then filter schools that are within 1km of a trunk road

Spatial join

We have a dataset of schools. The school dataframe contains information such as the school name, but not on the administrative region it's in. To add data on the administrative region that the school is in, we'll perform a spatial join.

Check the variable names. No names of second administrative divison :(

```
names(schools_sf)
```

```
## [1] "osm_id"   "name"     "amenity"  "geometry"
```

Spatial join

Use `st_join` to add attributes from `city_sf` to `schools_sf`. `st_join` is similar to other join methods (eg, `left_join`); instead of joining on a varible, we join based on location.

```
schools_city_sf <- st_join(schools_sf, city_sf)

schools_city_sf %>%
  names() %>%
  print() %>%
  tail(10)
```

```
##  [1] "osm_id"     "name"       "amenity"    "geometry"   "GID_2"      "GID_0"
##  [7] "COUNTRY"    "GID_1"      "NAME_1"     "NL_NAME_1"  "NAME_2"     "VARNAME_2"
## [13] "NL_NAME_2"  "TYPE_2"     "ENGTYPE_2"  "CC_2"       "HASC_2"     "area_m"
## [19] "area_km"

##  [1] "NL_NAME_1"  "NAME_2"     "VARNAME_2"  "NL_NAME_2"  "TYPE_2"     "ENGTYPE_2"
##  [7] "CC_2"       "HASC_2"     "area_m"     "area_km"
```

Spatial join

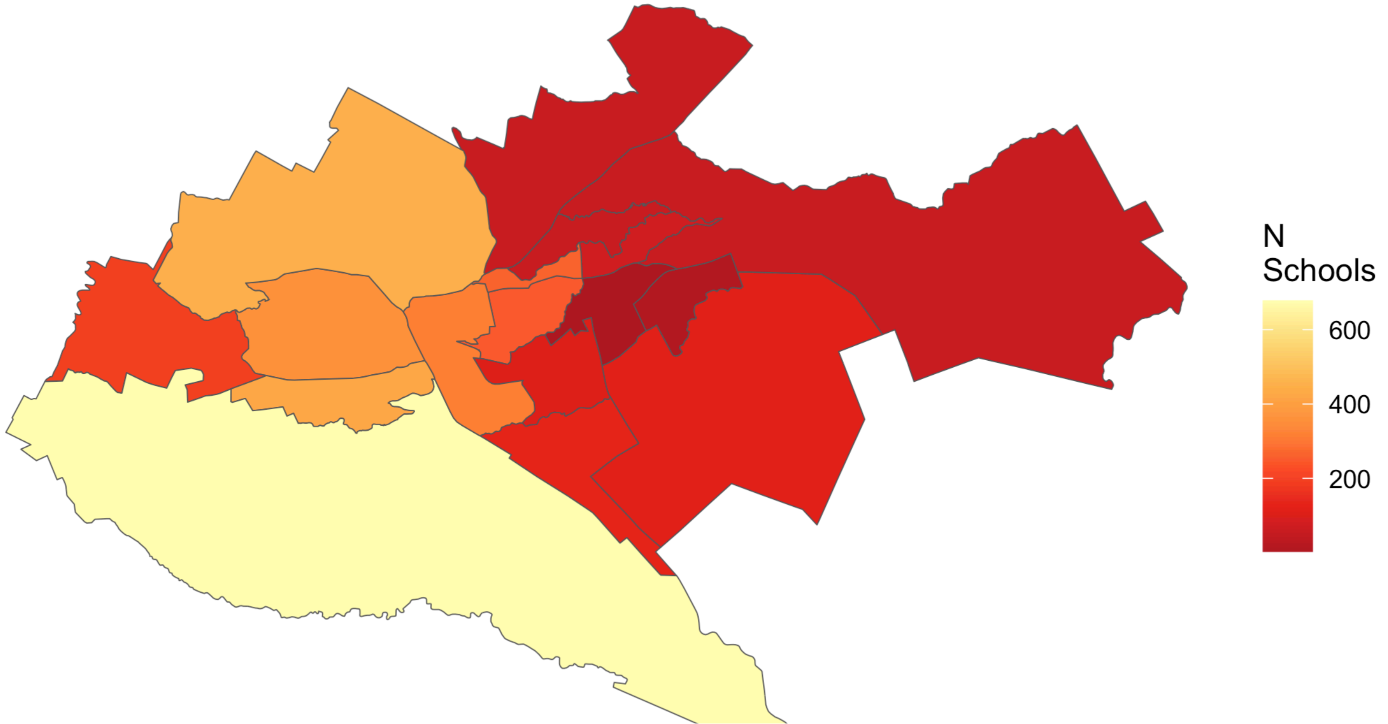
Exercise: Make a static map using of administrative areas, where each administrative area polygon displays the number of schools within the administrative area.

Solution:

```
## Dataframe of number of schools per NAME_2
n_school_df <- schools_city_sf %>%
  st_drop_geometry() %>%
  group_by(NAME_2) %>%
  summarise(n_school = n()) %>%
  ungroup()

## Merge info with city_sf
city_sch_sf <- city_sf %>% left_join(n_school_df, by = "NAME_2")

## Map
p <- ggplot() +
  geom_sf(data = city_sch_sf,
          aes(fill = n_school))
```

Spatial join

```
ggplot() +
  geom_sf(data = city_sch_sf,
          aes(fill = n_school)) +
  labs(fill = "N\nSchools") +
  scale_fill_distiller(palette = "YlOrRd") +
  theme_void()
```



Spatial join

Let's outsource to chatGPT (or gemini or your other favorite AI). Try entering the below prompt into chatGPT to see how it does. Does chatGPT give a correct answer? Do you need to modify chatGPT's output to make it work?

In R, I have an sf points object of schools called schools_sf. I also have the second administrative divisions of a city as an sf polygon called city_sf and where each location is uniquely defined by the variable NAME_2. Make a static map using of administrative areas, where each administrative area polygon displays the number of schools within the administrative area. Provide R code for this.

Resources

- sf package cheatsheet
- Spatial Data Science with Applications in R
- Geocomputation with R

Thank you!