Geographic Information Systems (GIS) in R

Jennifer Lin

2022-05-03

Overview of the Workshop

Introduction to the Material

Welcome to the Geographic Information Systems (GIS) in R Workshop! You are likely no doubt wondering why we are here and why I am teaching this in R rather than ArcGIS, or any similar software. GIS, as we will discuss later, analyzes geographical data and R provides an open source, reproducible way of doing this. In this workshop, we will cover the basics of GIS tools in R and I will leave you with some familiarity with this skill, along with recommendations for further exploration. This workshop has three goals:

1. Become familiar with how to get geographic data and how to parse that data.
2. Become familiar with shape files, how to access them and how to plot them using `ggplot2`
3. Become familiar with distance calculations and learn to add additional layers of data to an existing map.

We will cover a lot but not in great detail. This workshop, while assuming more intermediate R knowledge, is not meant to be a deep dive into all things related to GIS in R. The examples from today will be from a predominantly US context, but the skills can be applied to geographic data from almost anywhere else.

Assumptions of Skills

This is a more advanced R workshop and covers packages in the `tidyverse` universe, like `dplyr` and `ggplot2` at a more advanced level. For this workshop, I am assuming some level of working proficiency in these packages. Specifically:

1. You have broad-based familiarity with `ggplot2`[1]. There is no need to know how to write a chunk of `ggplot` code from memory, but you should have an understanding of what each line of code does.
2. You should know how to install and read in packages in R. We will cover many packages today.
3. You can load in data based in different file types wrangle data in `dplyr`[2]. Again, there is no need to know how to wrangle data off memory, but you should be familiar with functions like `filter()`,

[1] To review these basics, please refer to the materials for my Beginner's `ggplot` Workshop on GitHub: `https://github.com/NUpolisci/Fall-Module-3`. Also, refer to my Advanced `ggplot` Workshop on GitHub for ways to streamline coding in `ggplot2`: `https://github.com/NUpolisci/Adv_ggplot_W22`. These skills will come in handy here.

[2] For a quick overview of `dplyr`, see my `dplyr/ggplot` crash course workshop notes on GitHub: `https://github.com/NUpolisci/ggplot-workshop`

`select()` and `mutate()`. You should also know how to use a pipe (`%>%`).

What is GIS?

Before we get into the data and start making maps, I find it fitting to lay the foundations for what GIS is and how it is used. Here, we will also cover some basic geography to make sure that we are on the same page before jumping into the workshop.

An Overview of GIS

ESRI, the company that runs the Geographic Information Systems (GIS) defines GIS as follows[3]:

[3] See `https://www.esri.com/en-us/what-is-gis/overview`

> A geographic information system (GIS) is a system that creates, manages, analyzes, and maps all types of data. GIS connects data to a map, integrating location data (where things are) with all types of descriptive information (what things are like there). This provides a foundation for mapping and analysis that is used in science and almost every industry. GIS helps users understand patterns, relationships, and geographic context. The benefits include improved communication and efficiency as well as better management and decision making.

Let's break down this definition. A GIS is:

1. *A system that creates, manages, analyzes, and maps all types of data.*
2. *Connects data to a map, integrating location data (where things are) with all types of descriptive information (what things are like there)*

Why should we make maps?

> Mapping is important for political scientists because pinpointing where a particular political phenomenon occurred often provides clues as to why it occurred.
>
> – Cho and Gimpel, 2012, *Annual Reviews*

Politics is inherently spatial and many of the topics that we wrestle with in the discipline can be represented in geospatial terms. Here is just a short list of the examples that come to mind, for me, when thinking about how we can apply political science in a geographic context.

1. Electoral Redistricting
2. Political Representation

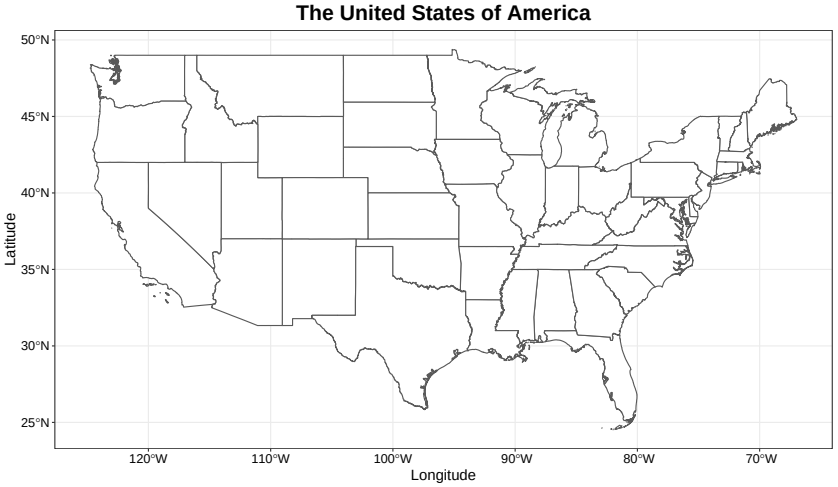
3. Political Participation
4. Political Attitudes
5. Social Interactions

A Review of Geography

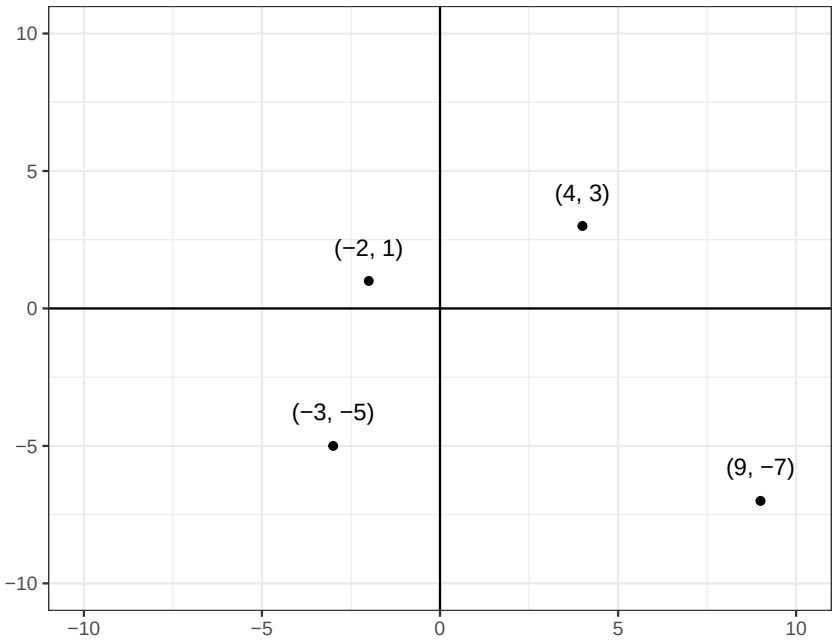
In elementary school, you might have learned how to interpret a map or a globe. There, you may have encountered the terms "latitude" and "longitude". Google defines these terms as follows:

latitude The angular distance of a place north or south of the earth's equator, or of a celestial object north or south of the celestial equator, usually expressed in degrees and minutes.

longitude The angular distance of a place east or west of the meridian at Greenwich, England, or west of the standard meridian of a celestial object, usually expressed in degrees and minutes.



During this time, you may have also learned how to plot points on a Cartesian coordinate plane, the one with the x-axis and y-axis. Each point has an x and a y coordinate and their point is where these two lines cross. For example, you likely did something like this:



In addition, you likely learned about the distance formula to calculate the distance between any two given points.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Here, the distance between the points $(-2, 1)$ and $(4, 3)$ is

$$\begin{aligned} d &= \sqrt{(4 - (-2))^2 + (3 - 1)^2} \\ d &= \sqrt{36 + 4} \\ d &= \sqrt{40} \approx 6.324555 \end{aligned}$$

I know that this all looks and seems extremely basic, much less insulting for a graduate level workshop. However, I bring this up for a reason. GIS systems seem to carry a connotation of something that is complex and adding R as a layer to it, rather than just teaching this in ArcGIS, may make it seem more so. However, GIS tools in R is as simple as the math review I just did. It is applying the understanding of these basic math concepts, with appropriate R commands that gets you very fancy visualizations and analyses.

In this workshop, we will discuss **geocoding**, which is the process of getting latitude (x) and longitude (y) points from an address. From there, we can plot it on a map. This is possible because the Earth, when placed on a map, is essentially one large coordinate plane such that the intersection of the Equator to the Prime Meridian generates the $(0, 0)$ origin. We will also discuss **shape files**, which essentially provide us the data to generate the map, so we know what to plot on this coordinate plane. Finally, we will discuss **distance** calculations between two given locations, which is an extension of the distance

formula discussed here. The difference is that we need to factor in the curvature of the Earth, and thankfully, there are canned R commands that can help with that.

Data Sources and Formats

To make any map or do any analysis using spatial data, we first need to discuss data structures and data sources. GIS-based analyses often include both the map itself and information about the area covered by the map. Here, we will discuss both components.

Geocoding of Spatial Data

Geocoding "[T]he process of attaching geographic coordinates (usually as latitude and longitude or as x–y coordinates) to data that are in more traditional geographic formats, such as, for example, street addresses, zip codes, district numbers, county identifiers, or FIPS (Federal Information Processing Standard) codes." [Cho and Gimpel, 2012]

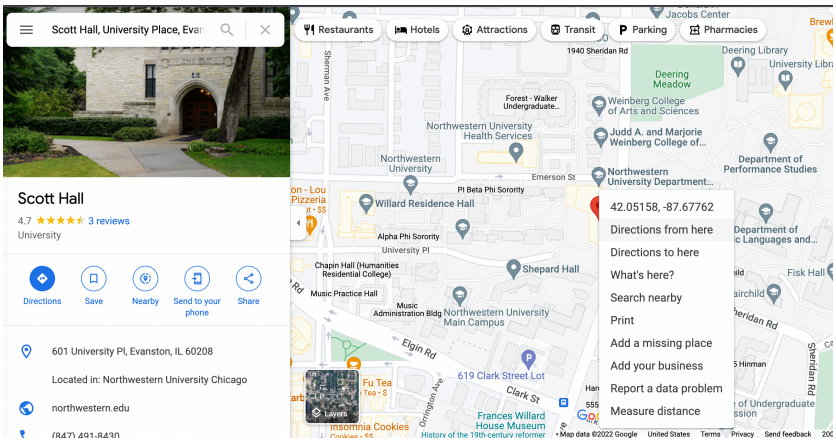
Geocoding is the process by which we append a latitude and longitude coordinate to an address, telling GIS where it is located on the Earth's coordinate plane. There are many different ways in which we can geocode a location. These include:

1. Brute Force: Google Maps
2. Brute Force: Census Geocoder API
3. By Script: `tidygeocoder`

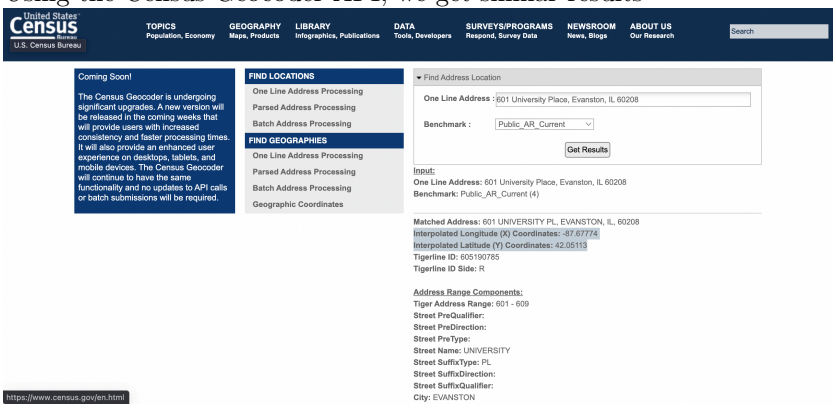
Suppose we want to geocode Scott Hall (601 University Place, Evanston, IL 60208). I will demonstrate this with three methods. But first, a dataframe, because we will need that.

```
geolocate <- data.frame(
  place = "Scott Hall",
  address = "601 University Place, Evanston, IL 60208"
)
```

First, let's look at Google Maps. Using the point and click feature (Right click to get coordinates), we see that Scott Hall is at Lat = 42.05 and Long = -87.68



Using the Census Geocoder API, we get similar results



Finally, let's look at the `tidygeocoder` package[4], and here is where we will need to dataframe from before. This package provides several API options for us to use, but many come with a fee. The best two free ones are the OpenStreetMap (`osm`) and US Census Geocoder (`census`). Let's look at the OSM first.

[4] Package documentation here: `https://jessecambon.github.io/tidygeocoder/index.html`

```
osm_geo <- geolocate %>%
  geocode(address, method = 'osm', lat = latitude , long = longitude)
osm_geo
```

```
## # A tibble: 1 x 4
##   place      address                                    latitude longitude
##   <chr>      <chr>                                         <dbl>     <dbl>
## 1 Scott Hall 601 University Place, Evanston, IL 60208       42.1     -87.7
```

And now, Census

```
census_geo <- geolocate %>%
  geocode(address, method = 'census', lat = latitude , long = longitude)
census_geo
```

```
## # A tibble: 1 x 4
```

```
##   place      address                                        latitude longitude
##   <chr>      <chr>                                             <dbl>     <dbl>
## 1 Scott Hall 601 University Place, Evanston, IL 60208           42.1     -87.7
```

How did we do? We should get the same results from all of these steps, and indeed we do.

When we want to conduct analyses with spatial data, the data need to describe a location of some size. It can be at a country, state, province, county, zip code, census tract, or census block level. Each of these describe a particular geographical area. Examples of data of this sort might include:

- Vote share for Democrats/Republicans per precinct
- Percent of people living in poverty in each census block
- Percent of people in each county who are born in that county

In the United States, each geographic area is identified using the Census GEOID, which is a 15 digit numeric string that describes the state (S), county (C), census tract (T) and census block (B) such that it is [SS][CCC][TTTTTT][BBBB]. Data to match the ID based on state and county can be located using the `tidycensus` package.

```
data("fips_codes")
```

state	state_code	state_name	county_code	county
AL	01	Alabama	001	Autauga County
AL	01	Alabama	003	Baldwin County
AL	01	Alabama	005	Barbour County
AL	01	Alabama	007	Bibb County
AL	01	Alabama	009	Blount County
AL	01	Alabama	011	Bullock County

Census GEOIDs are usually only as long as they need to be. So if you need a GEOID for a county, it will usually only be 5 digits – 2 for the state and 3 for the county. Data to match census tracts and blocks often come in the shape file itself, as it would be too hectic to put it all in a FIPS code dataset. More on shape files later.

We can also describe any particular point of geography with data. Data here often describe attributes of a particular address. Examples of data include:

- Number of protests at a given location
- Number of political rallies at a given location
- Number of residents registered to vote in a household
- COVID Testing location hours of operation (Here are data from Wisconsin).

Returning to our Scott Hall example from above, we can get a Census GEOID through a process called **reverse geocoding**. This is a process where we supply an API with latitude and longitude coordinates and have it return the address and other Census GEOID details.

We can do this using the `tidygeocoder` or `tigris` packages. First, let's look at `tidygeocoder`:

```
reverse_osm <- census_geo %>%
  reverse_geocode(lat = latitude, long = longitude, method = 'osm',
                  address = address_found, full_results = TRUE)
reverse_osm %>% t()
```

```
##                [,1]
## place          "Scott Hall"
## address        "601 University Place, Evanston, IL 60208"
## latitude       42.05113
## longitude      -87.67774
## address_found "Northwestern University, 633, Clark Street, Downtown, Evanston, Evanston Township, Co
## place_id       282635476
## licence        "Data © OpenStreetMap contributors, ODbL 1.0. https://osm.org/copyright"
## osm_type       "relation"
## osm_id         2105485
## osm_lat        "42.0551164"
## osm_lon        "-87.67581113482166"
## amenity        "Northwestern University"
## house_number   "633"
## road           "Clark Street"
## neighbourhood "Downtown"
## town           "Evanston"
## municipality   "Evanston Township"
## county         "Cook County"
## state          "Illinois"
## postcode       "60208"
## country        "United States"
## country_code   "us"
## boundingbox    character,4
```

For something that is perhaps easier, let's just look at the `tigris::call_geolocator_latlon()` that gives us a Census GEOID.

```
call_geolocator_latlon(census_geo$latitude, census_geo$longitude)
```

```
## [1] "170318087021003"
```

The results show us that Scott Hall's geolocated details are as follows:

- State: 17
- County: 17031
- Census Tract: 17031808702
- Census Block: 170318087021003

Shape Files for Maps

To draw maps in R, we will need to get *shape files*, which are data frames that include geographic information. The US Census Bureau has a database that contains shape files for all of the jurisdictions in the country, which includes states, counties, school districts and more.

We can get these shape files in two different ways. First, we can use the Census Burean Web Archive and download the file as a shape file. You can go on the website or modify the following code:

```
url <- "https://www2.census.gov/geo/tiger/"
extension <- "TIGER2021/TRACT/"
file <- "tl_2021_11_tract.zip"
shp_url <- paste0(url, extension, file)

if (! dir.exists(here::here("data/DC_Tract/"))) {
  download.file(
    shp_url,
    dest = here::here("data/DC_Tract.zip")
  )
  unzip(here::here("data/DC_Tract.zip"),
        exdir = here::here("data/DC_Tract"))
}
```

This gives us a `.zip` file that contains many files within it. Most of this is useful for ArcGIS work.

```
dir(here::here("data/DC_Tract/"))
```

```
## [1] "tl_2021_11_tract.cpg"            "tl_2021_11_tract.dbf"
## [3] "tl_2021_11_tract.prj"            "tl_2021_11_tract.shp"
## [5] "tl_2021_11_tract.shp.ea.iso.xml" "tl_2021_11_tract.shp.iso.xml"
## [7] "tl_2021_11_tract.shx"
```

```
DC_Tract_sf <- read_sf(
  here::here("data/DC_Tract/tl_2021_11_tract.shp")
  )
```

This process is useful if you have data that do not belong as part of an API. You will need this code to read in outside shape files, which we will demonstrate later. For now, the US Census shape files are also based in an API, through the `tigris` R package, which can make

loading these shape files a lot easier. Functions in this function take very intuitive names.

States `states()`
Counties `counties()`
Census Tract `tracts()`
Census Blocks `blocks()`
Zip Code `zctas()`

Each of these functions take a location general area, usually the name or FIPS code of a state. Below, we get the census tracts and counties for the DC area using its FIPS code (11).

```
DC_County <- counties("11", year = 2020)
DC_Tract <- tracts("11", year = 2020)
```

Here is the same code with the abbreviation for the area, in this case, DC. If you use this method, use the two letter abbreviation for the state of interest.

```
DC_County <- counties("DC", year = 2020)
DC_Tract <- tracts("DC", year = 2020)
```

Each shape file comes with many implicit details. This includes information on the map projection, the coordinates and other qualities that are useful if these were used in ArcGIS. This information is embedded in all the other files that are included if you downloaded the folder, but are also embedded in the data loaded on `tigris`. When you use a shape file, it is always a good idea to look at the structure of the shape file. We do this using `st_crs()`, which gives us information on the Coordinate Reference Systems (CRS).

```
st_crs(DC_County)
```

```
## Coordinate Reference System:
##   User input: NAD83
##   wkt:
## GEOGCRS["NAD83",
##     DATUM["North American Datum 1983",
##         ELLIPSOID["GRS 1980",6378137,298.257222101,
##             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",4269]]
```

When you do this, pay particular attention to `DATUM` and `ID`. In the output above, you find two important sections:

- `DATUM["North American Datum 1983",`: This suggests that the shape file is using the North American Datum 1983 (NAD83), which is perhaps the most common for US government generate shape files.
- `ID["EPSG",4269]]`: 4269 is the ID for the NAD83. This is needed for any conversions between CRS projections.

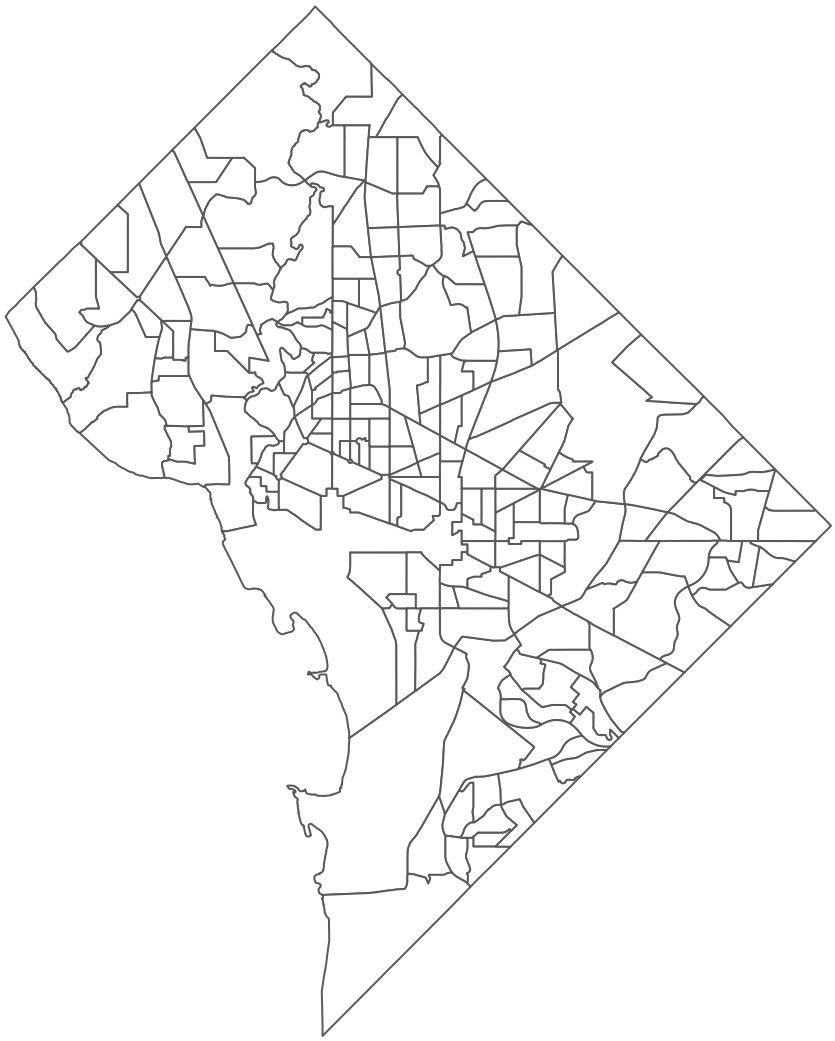
Drawing Maps with `ggplot2` and `sf`

Now, with shape files settled, let's look at how we can apply them to make useful maps. First, we will look at drawing an outline and then adding more details from a different data source.

Draw the Outline

Drawing a map outline can be done easily with the `sf` and the `ggplot2` packages. Unlike a basic plot for other types of graphs in `ggplot2`, you usually would not need to specify an `aes()` with x and y, because `grom_sf()` gets these data from the `geometry` column in your shape file dataframe. Therefore, you only need to specify the dataframe that you need, the type of plot (`geom_sf()` for maps), a fill color and a theme. Labels for titles and subtitles follow that of a standard `ggplot2` with `labs()`.

```
ggplot(DC_Tract)+
  geom_sf(fill = "white")+
  theme_void()
```



Add Details

The above map looks fine, but it is quite boring. You would want to add some more details. This can be a point with one latitude and longitude coordinate or an area, indicated by a FIPS or GEOID. In both cases, you work with shapefiles as you would with any other data frame[5].

For today, we will use the Washington DC points of interest data that come from the DC government. We can download this data from source using this code:

[5] Remember that shape files are just dataframes with geographic information.

```
url <- "https://opendata.arcgis.com/api/v3/datasets/"
extension <- "f323f677b3f34fe08956b8fcce3ace44_3/downloads/"
file <- "data?format=shp&spatialRefId=4326"
data_url <- paste0(url, extension, file)
```

```
if (! dir.exists(here::here("data/DC_POI/"))) {
  download.file(
    data_url,
    dest = here::here("data/DC_POI.zip")
  )
  unzip(here::here("data/DC_POI.zip"),
        exdir = here::here("data/DC_POI"))
}
```

If we look into the materials that are included with this download, we see quickly that we just got a shape file, hence the presence of a file that ends with `.shp`.

```
dir(here::here("data/DC_POI/"))
```

```
## [1] "Points_of_Interest.cpg" "Points_of_Interest.dbf" "Points_of_Interest.prj"
## [4] "Points_of_Interest.shp" "Points_of_Interest.shx" "Points_of_Interest.xml"
```

Therefore, let's revisit the brute force method from earlier and read it as a shape file using `read_sf()`.

```
DC_POI <- read_sf(
  here::here("data/DC_POI/Points_of_Interest.shp")
) %>%
  mutate(
    long = unlist(map(geometry,1)),
    lat = unlist(map(geometry,2))
  )
```

The contents in `mutate()` break the points in `geometry` into longitude and latitude points. This will be useful when we calculate distances later on.

There are a lot of places in this dataset. Therefore, to streamline things, I created a `.places` file to give us places that we should focus, including most of the major tourist locations in DC. We use `readLines()` since this file is just a plain text file with names of places that is intended to be used as a character string[6].

```
Select_Places <- readLines("DC.places")
```

Now that we have a few places of interest, let's revisit the original file and look at the CRS. Compare this output to the one from the `DC_Tract` CRS exploration.

```
st_crs(DC_POI)
```

```
## Coordinate Reference System:
```

[6] **Pro Tip**: Plain text on any computer usually comes with `.txt` ending. However, you can change the ending to be whatever you want. Therefore, I just implemented the `.places` custom file ending to let you know that this file contains places in DC.

```
##   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]]
```

Here, the data uses the WGS84 (`DATUM["World Geodetic System 1984",`) coordinate reference system, which has the ID 4326 (`ID["EPSG",4326]]`).

Recall that the CRS for the shape file from the US census was NAD83 (4269). Therefore, in order to do any meaningful analyses, we need to convert one of these datasets to the CRS of the other. We can do that using `st_transform()` with the CRS that we want in the parentheses. We do this in the following code, while simultaneously subsetting the data to just the locations included in the places file.

```
DC_Destinations <- DC_POI %>%
  filter(ALIASNAME %in% Select_Places) %>%
  st_transform(crs = 4269)
```

To add details to a map, let's pick a few locations from the list to pay specific attention. Here is a short list I chose:

```
vacation <- c(
  "THOMAS JEFFERSON MEMORIAL",
  "UNITED STATES HOLOCAUST MEMORIAL MUSEUM",
  "US DEPARTMENT OF THE TREASURY",
  "US NAVY MUSEUM",
  "NATIONAL ZOO",
  "US CAPITOL",
  "DUPONT CIRCLE"
)
```

Take these locations out of the data and select the place name and its geographic coordinates.

```
chosen_places <- DC_Destinations %>%
  filter(ALIASNAME %in% vacation) %>%
  select(ALIASNAME, long, lat)
```

Distance Calculations

Suppose we are interested in calculating the distance between two points. Before going to making the map, let's take a quick pit stop on calculating distances between any two locations on the list that we just subsetted.

In R, we can calculate the Haversine, "as the crow flies", distance between any two geographic points while keeping in mind the curvature of the Earth. This is the math that goes into calculating the distance as a function of the longitude and latitude points.

$$\begin{aligned}\text{haversin}\left(\frac{d}{r}\right) &= \text{haversin}(\phi_{\text{p1 lat}} - \phi_{\text{p2 lat}}) \\ &+ \cos(\phi_{\text{p2 lat}})\cos(\phi_{\text{p1 lat}})\text{haversin}(\lambda_{\text{p1 long}} - \lambda_{\text{p2 long}})\end{aligned}$$

When we rearrange this equation, we get distance, in meters, as:

$$d = 2r \arcsin\left(\sqrt{\sin^2\frac{\phi_{\text{p1 lat}} - \phi_{\text{p2 lat}}}{2} + \cos(\phi_{\text{p2 lat}})\cos(\phi_{\text{p1 lat}})\sin^2\frac{\lambda_{\text{p1 long}} - \lambda_{\text{p1 long}}}{2}}\right)$$

This is a lot of math, which is much easier to do in R. Before making these calculations, we should do some rearranging of the data. First, we need to create a list of to and from locations. The `slice()` functions under `cbind()` below does this at random. You can opt to do this by hand too. Next, we need to rename the variables to be something more meaningful.

```
chosen_places <- cbind(
  slice(chosen_places, 1:n()),
  slice(chosen_places, 2:n(), 1)
) %>%
  rename(
    from       = ALIASNAME,
    start_long = long,
    start_lat  = lat,
    to         = ALIASNAME.1,
    end_long   = long.1,
    end_lat    = lat.1
  ) %>%
  select(-contains("geometry"))
```

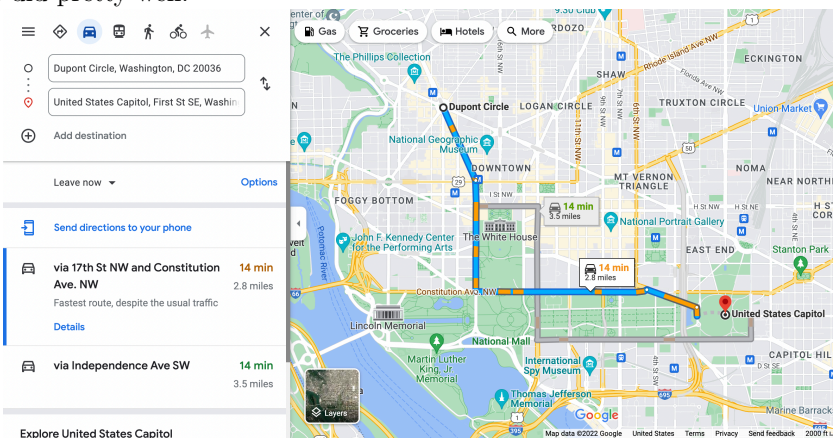
We can use `geosphere::distHaversine()` to calculate the distance. This code allows us to get results in meters, which we convert to miles and kilometers.

```
chosen_places <- chosen_places %>%
  mutate(
    distance = distHaversine(
      p1 = cbind(start_long, start_lat),
      p2 = cbind(end_long, end_lat))
    ) %>%
  group_by(from) %>%
  slice(which.min(distance)) %>%
  mutate(
    miles = distance/1609.344,
    km    = distance/1000
  )
```

Let's take a look at the results. Here, note that `st_set_geometry(NULL)` gets rid of the geometry column for this display.

```
chosen_places %>%
  select(from, to, miles) %>%
  st_set_geometry(NULL)
```

So how did we do? The results above show that the distance from the US Capitol to Dupont Circle is 2.3 miles. Let's look at Google maps, which shows that the shortest driving distance is 2.8 miles. Keeping in mind that what we have is a direct, straight line distance, we did pretty well.



Points on a Map

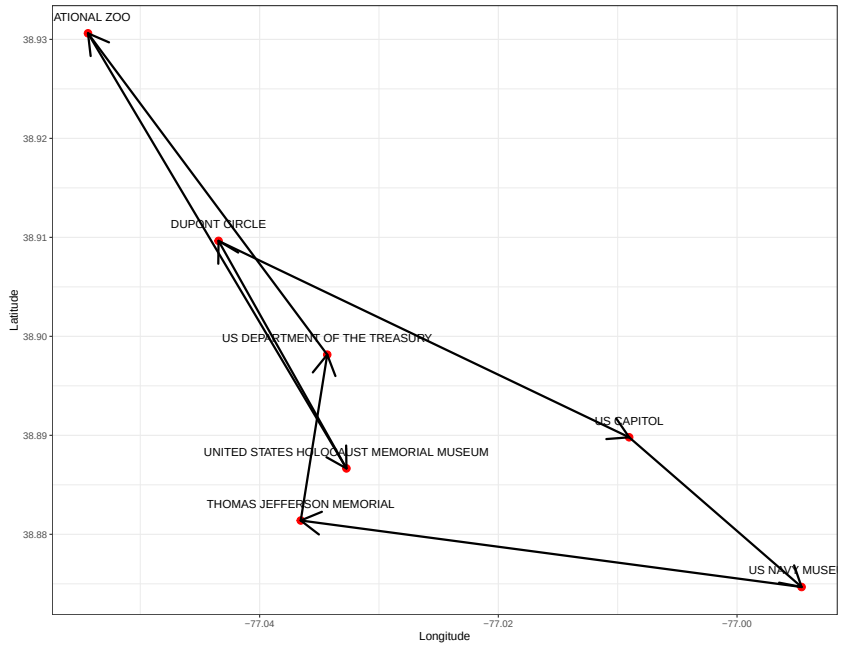
Now, let's see how these data show up on a coordinate plane. Recall that earlier in this session, I discussed how the world is basically

from	to	miles
DUPONT CIRCLE	US CAPITOL	2.305162
NATIONAL ZOO	UNITED STATES HOLOCAUST MEMORIAL MUSEUM	3.256168
THOMAS JEFFERSON MEMORIAL	US DEPARTMENT OF THE TREASURY	1.165118
UNITED STATES HOLOCAUST MEMORIAL MUSEUM	DUPONT CIRCLE	1.691316
US CAPITOL	US NAVY MUSEUM	1.303710
US DEPARTMENT OF THE TREASURY	NATIONAL ZOO	2.491286
US NAVY MUSEUM	THOMAS JEFFERSON MEMORIAL	2.306954

a large coordinate plane. Let's plot these points on a basic coordinate plane, agnostic towards coordinate reference systems. We can do this using tools that we use to make scatterplots. We can use `geom_segment()` to connect the points.

```
ggplot(chosen_places, aes(x = start_long, y = start_lat))+
  geom_point(size = 3, color = "red")+
  geom_text(aes(label = from), vjust = -1.5)+
  geom_segment(
    aes(
    x = start_long,
    y = start_lat,
    xend = end_long,
    yend = end_lat
    ),
    arrow = arrow(length = unit(0.3, "inches")),
    size = 1,
    lineend = "round",
    linejoin = "round"
  )+
  xlab("Longitude")+
  ylab("Latitude")+
```

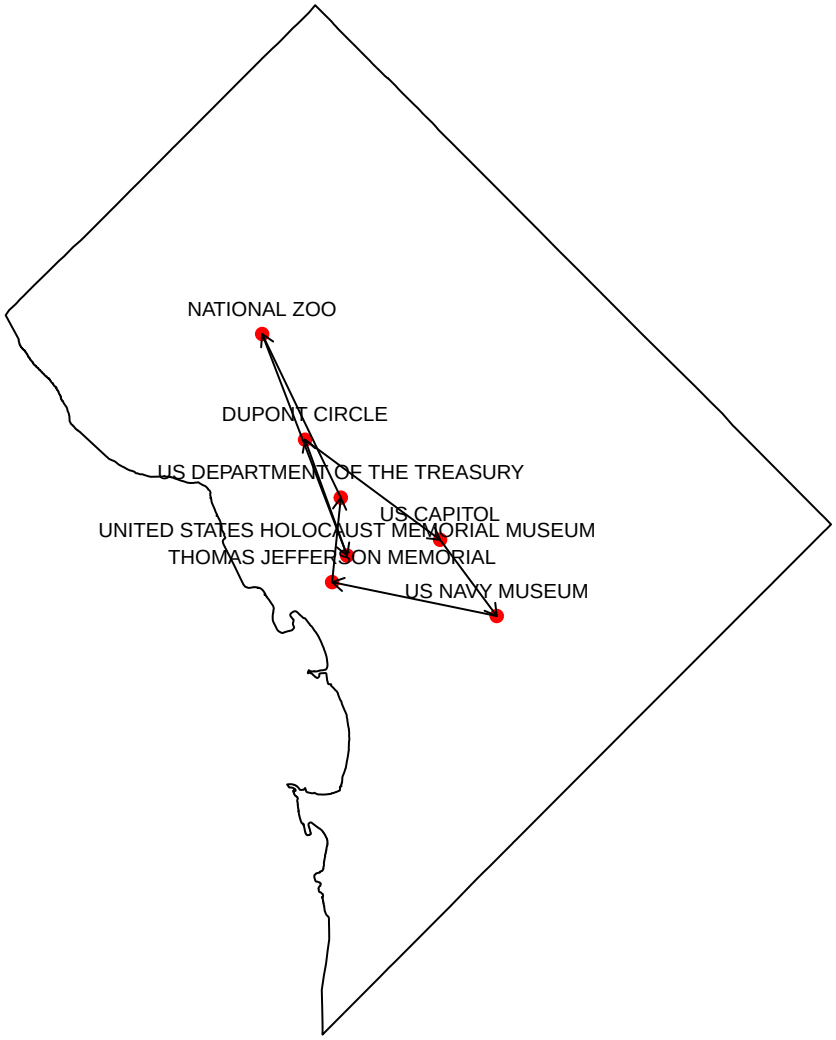
```
  theme_bw()
```



This is fine, but this is not as interesting or cool. Let's put it on a map, combining `geom_sf()` and the scatterplot from above. Note that this only works when you have two shape files that are under the same CRS.

```
ggplot(DC_County)+
  geom_sf(fill = "white", color = "black")+
  geom_point(
    chosen_places,
    mapping = aes(x = start_long, y = start_lat),
    size = 3, color = "red")+
  geom_text(
    chosen_places,
    mapping = aes(
      x = start_long, y = start_lat,
      label = from),
    vjust = -1.25)+
  geom_segment(
    chosen_places,
    mapping = aes(
      x = start_long,
      y = start_lat,
      xend = end_long,
      yend = end_lat
    ),
    arrow = arrow(length = unit(0.1, "inches")),
```

```
  size = 0.5,
  lineend = "round",
  linejoin = "round"
)+
theme_rgis_void()
```



Exercises

1. Replicate the DC shape file read in.
2. Select a state of your choice. Get the county shape file from the Census database.
3. Pick a handful of locations from the DC places file.
4. Calculate the distance between the locations.
5. Place the locations on a map, tracing a plausible path for a tourist vacation.

6. CHALLENGE: Take the state for which you got the county shape file from Step 2. Pick one county. Make a map that shades that county as a different color.

References

Wendy K Tam Cho and James G Gimpel. Geographic information systems and the spatial dimensions of american politics. *Annual Review of Political Science*, 15:443–460, 2012.