Lecture 10: Spatial Data in R

Getting Started with sf

James Sears

AFRE 891 SS 24

Michigan State University

*Parts of these slides are adapted from **"Advanced Data Analytics"** by Nick Hagerty and **"R Geospatial Fundamentals"** by the UC Berkeley D-Lab used under **CC BY-NC-SA 4.0**.

Table of Contents: Today

Table of Contents: Later

Spatial Data in R and the sf Package

GIS

The most widespread **geographic information system** is **ArcGIS.**

Advantages of ArcGIS:

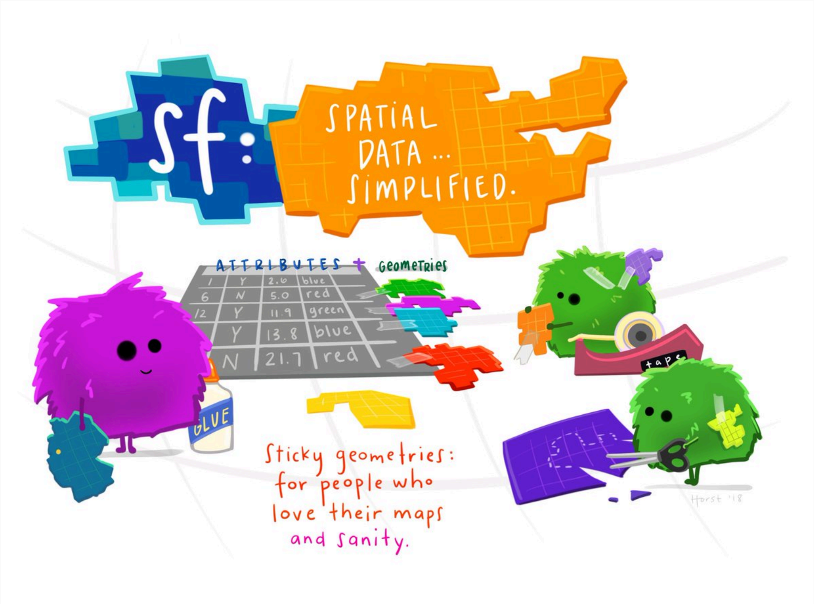
- Avoid coding.
- Interface for browsing and exploring data is incredibly comprehensive and fast.

Why we're using R instead:

- Free.
- Reproducible.
- Scriptable.
- Easily integrated with the rest of your project.
- Easy to export attractive, professional maps.
- Honestly, easier if you know some R already.

sf: Simple Features

`sf` is the main package for working with **vector** data in R.



- **features:** things with a spatial location/extent
- **simple:** linestrings and polygons are built from points (nodes) connected by straight lines (edges)
- Integrates cleanly with **tidyverse**
- Since v1.0: uses spherical geometry for transformations!

sf: Simple Features

`sf` is the main package for working with **vector** data in R.

Install and load it (and a couple other packages):

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

Shapefiles

The ESRI **shapefile** is the most widely used type of file **format for storing geospatial vector data**. A "shapefile" is actually a collection of 3+ files:

Required files:

- `shp`: The main file that stores the feature geometry
- `shx`: A positional index for locating the feature geometry in the `shp` file
- `dbf`: The data table (in dBase IV format) that stores the attribute information for each feature

Shapefiles

The ESRI **shapefile** is the most widely used type of file **format for storing geospatial vector data**. A "shapefile" is actually a collection of 3+ files:

Optional files:

- `prj`: Stores the coordinate reference system information. (**should be required!**)
- `sbn` and `sbx`: spatial index to speed up geometry operations (*used only by ESRI software*)
- `xml`: Metadata — Stores information about the shapefile.
- `cpg`: Specifies the code page for identifying the character encoding set to be used.

All files need to be kept together in the same directory.

Loading Shapefile Data

List the files:

```
dir("data/MichiganCounties")
```

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

Load the data with `st_read()`:

```
counties ← st_read("data/MichiganCounties/MichiganCounties.shp")
```

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

Loading Shapefile Data

We can also load the data by pointing to the *folder* containing the shapefile files:

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

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

Geodatabases

Shapefiles have some **severe limitations**.

- They must be less than 2 GB.
- Column names cannot be longer than 10 characters.
- The number of columns is limited to 255.

Another, newer file format is called a **geodatabase (.gdb)**.

`st_read()` can handle geodatabases with the `layer` argument.

- The important thing to keep in mind is that in your computer, the `.gdb` file *appears* to be a folder, but the individual files within it are uninterpretable.
- `st_layers()` will show you the list of layers in a geodatabase.

sf Object Contents

Looking at the structure of the data, we can see that it's an sf object, but *also* a data frame with an extra **geometry** column.

This is also easy to check in RStudio in the Environment window.

```
str(counties)
```

```
## Classes 'sf' and 'data.frame':    83 obs. of  16 variables:
##  $ OBJECTID  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ FIPSCODE  : chr  "001" "003" "005" "007" ...
##  $ FIPSNUM   : int  1 3 5 7 9 11 13 15 17 19 ...
##  $ NAME      : chr  "Alcona" "Alger" "Allegan" "Alpena" ...
##  $ LABEL     : chr  "Alcona County" "Alger County" "Allegan County" "Alpena County" ...
##  $ TYPE      : chr  "County" "County" "County" "County" ...
##  $ CNTY_CODE : chr  "001" "003" "005" "007" ...
##  $ SQKM      : num  1799 2425 2181 1539 1359 ...
##  $ SQMILES   : num  694 936 842 594 525 ...
##  $ ACRES     : num  444428 599194 538923 380383 335744 ...
##  $ VER       : chr  "17A" "17A" "17A" "17A" ...
##  $ LAYOUT    : chr  "landscape" "landscape" "landscape" "landscape" ...
##  $ PENINSULA : chr  "lower" "upper" "lower" "lower" ...
##  $ ShapeSTAre: num  3.56e+09 5.10e+09 4.03e+09 3.08e+09 2.72e+09 ...
##  $ ShapeSTLen: num  242638 567352 261622 408568 255627 ...
##  $ geometry  :sfc_MULTIPOLYGON of length 83; first list element: List of 1
##   ..$ :List of 1
##   .. ..$ : num [1:1414, 1:2] -83.9 -83.9 -83.9 -83.9 -83.9 ...
```

sf Geometry

Looking at the geometry column a bit more reveals that

- Each row contains a **feature**
- Each feature contains a `simple feature geometry list column (sfc)`
    - Which in turn contains a `simple feature geometry (sfg)`

```
class(counties$geometry)
```

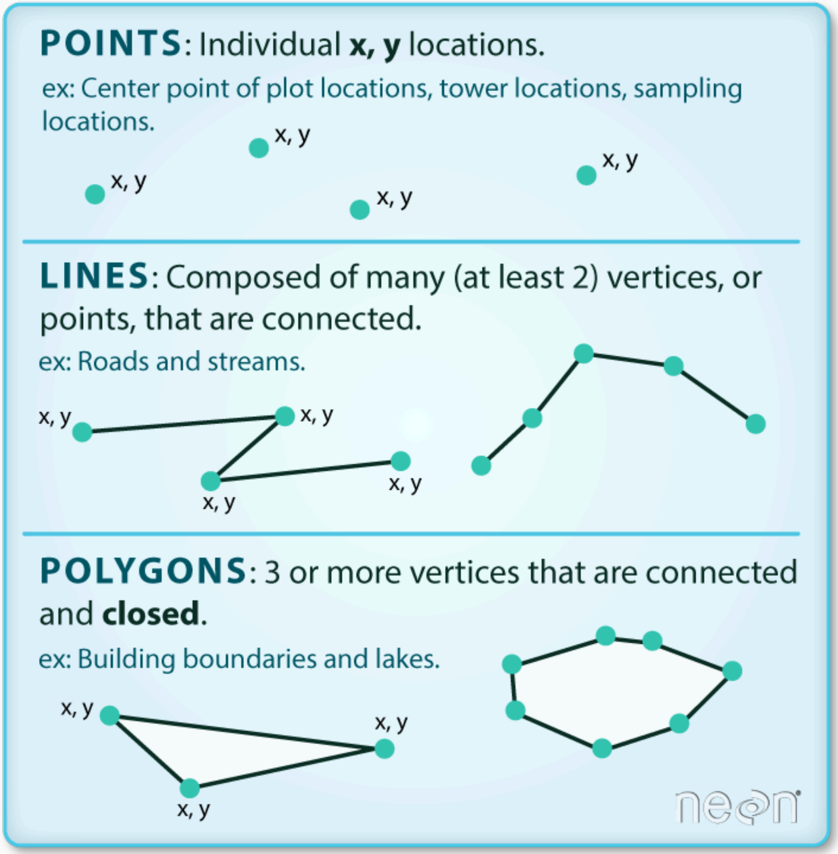
```
## [1] "sfc_MULTIPOLYGON" "sfc"
```

```
# and for the geometry of the first feature?
class(counties$geometry[[1]])
```

```
## [1] "XY"           "MULTIPOLYGON" "sfg"
```

sf Geometry Types

There are three main types of geometries that can be associated with `sf` object: points, lines and polygons:



sf Geometry Types

In an `sf data.frame` these geometries are encoded in a format known as **Well-Known Text (WKT)**.

For example:

> - POINT (30 10)
> - LINESTRING (30 10, 10 30, 40 40)
> - POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))

where X,Y coordinate values are separated by a space, coordinate pairs by a comma, and geometries by parentheses

sf Geometry Types

An `sf` object may also include the variants **multipoints**, **multilines**, and **multipolgyons** if some of the features are composed of multiple geometries:

> - MULTIPOINT ((10 40), (40 30), (20 20), (30 10))
> - MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10))
> - MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))

For example, if we had data representing US states (one per row), we could use

- POLYGON geometry for states like Utah or Colorado
- MULTIPOLYGON for a state like Hawaii, which includes many islands.

sf Geometry

Another thing to note is that an sf geometry is **sticky**

(Or like Kramer from Seinfeld: it always shows up)

```
head(counties[1:5, 1:4])
```

```
## Simple feature collection with 5 features and 4 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -87.11664 ymin: 42.41882 xmax: -83.19065 ymax: 46.6907
## Geodetic CRS:  WGS 84
##   OBJECTID FIPSCODE FIPSNUM    NAME                       geometry
## 1        1      001       1  Alcona MULTIPOLYGON (((-83.88712 4...
## 2        2      003       3   Alger MULTIPOLYGON (((-87.11602 4...
## 3        3      005       5 Allegan MULTIPOLYGON (((-85.54343 4...
## 4        4      007       7  Alpena MULTIPOLYGON (((-83.3434 44...
## 5        5      009       9  Antrim MULTIPOLYGON (((-84.84877 4...
```

sf Object to Dataframe

We can easily convert an sf object to a regular data frame by **removing the geometry**

```
counties_df <- st_drop_geometry(counties)
class(counties_df)
```

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

dataframe to sf

We can also go from a dataframe to an sf object if we have column(s) containing coordinate information.

Let's add the latitude and longitude of each county's centroid to `counties_df` and convert it to an sf object with points geometry:

```
centroids <- st_centroid(counties)
counties_df <- mutate(counties_df,
                      longitude = st_coordinates(centroids)[,1],
                      latitude = st_coordinates(centroids)[,2]) %>%
  st_as_sf(coords = c("longitude", "latitude"),
           crs = st_crs(counties))
class(counties_df)
```

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

Switching Geometry Columns

Note that an sf object can have multiple geometry columns, but only one can be **active** at a time.

We can view and switch the active geometry without deleting columns using `st_geometry()`:

```
counties_comb <- mutate(counties, points = counties_df$geometry)
st_geometry(counties_comb)
```

```
## Geometry set for 83 features
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -90.41829 ymin: 41.69613 xmax: -82.41348 ymax: 48.26269
## Geodetic CRS:  WGS 84
## First 5 geometries:
```

Switching Geometry Columns

Note that an sf object can have multiple geometry columns, but only one can be **active** at a time.

We can view and switch the active geometry without deleting columns using `st_geometry()`:

```
st_geometry(counties_comb) <- "points"
st_geometry(counties_comb)
```

```
## Geometry set for 83 features
## Geometry type: POINT
## Dimension:     XY
## Bounding box:  xmin: -89.69307 ymin: 41.88767 xmax: -82.68086 ymax: 47.6078
## Geodetic CRS:  WGS 84
## First 5 geometries:
```

Quick Mapping

Quick Mapping

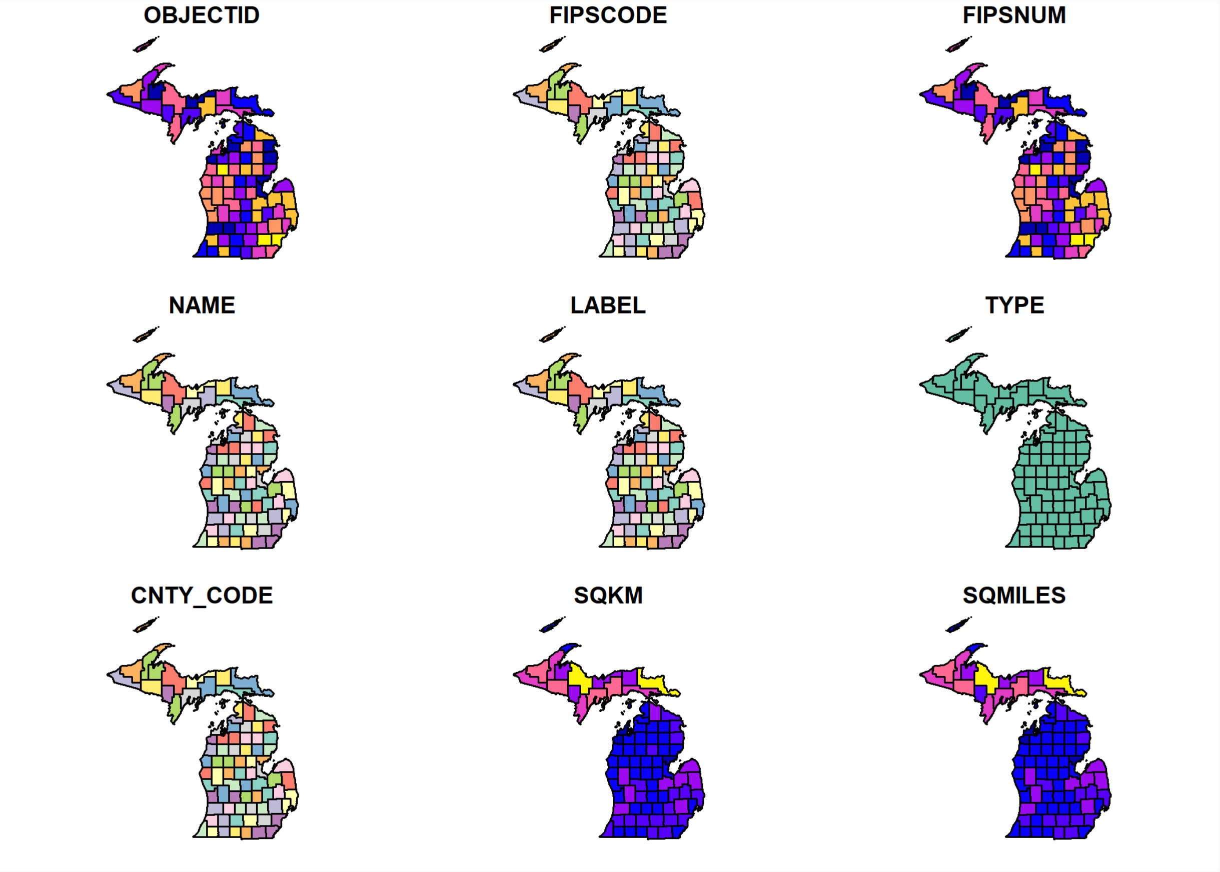
There are several ways we can quickly make maps in R

- base R's `plot()` function
- tmap package's `qtm()` function
- `ggplot()` and the `geom_sf()` function

Later on we'll go more in-depth with how to customize maps

Quick Mapping: base R

```
plot(counties)
```

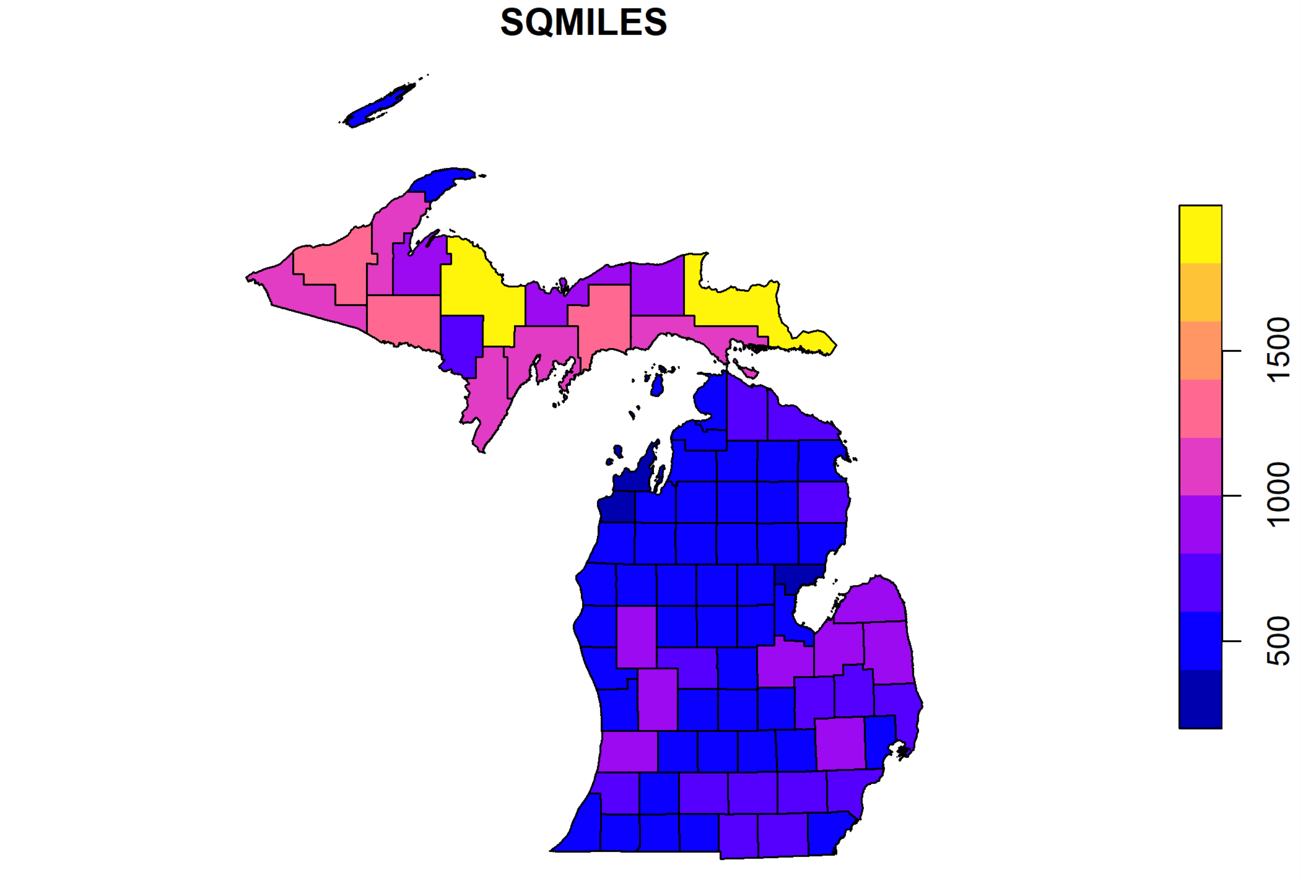


This yield a grid of maps where colors correspond to data values of the first 9 data frame columns

Quick Mapping: base R

To map just a single variable:

```
plot(counties["SQMILES"])
```



Quick Mapping: base R

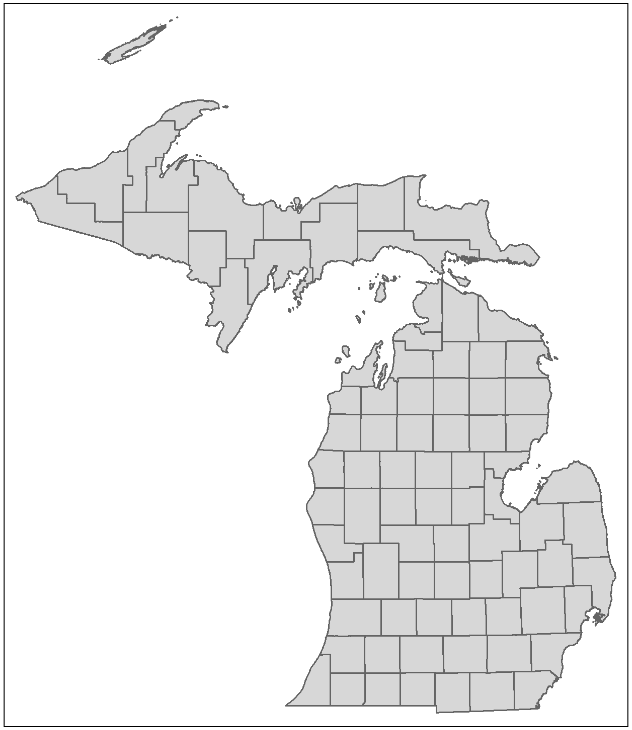
Or just the geometry:

```
plot(counties["geometry"])
```

Quick Mapping: tmap (static)

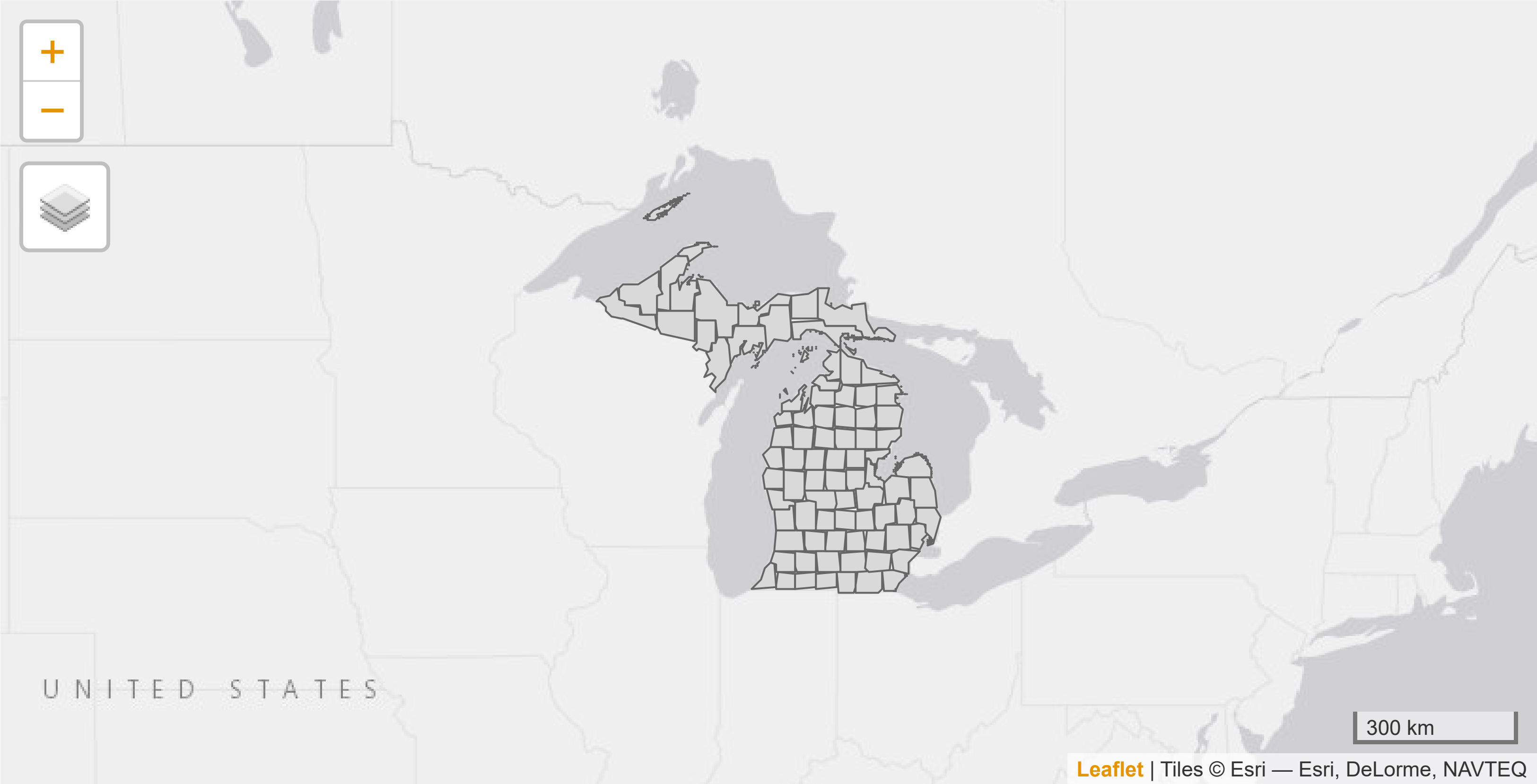
The .hi-slate[[tmap] package makes is easy to make thematic maps in R, both static and interactive

```
qtm(counties)
```



Quick Mapping: tmap (interactive)

```
tmap_mode("view")    # the default is mode = "plot"
qtm(counties)
```



Quick Mapping: Invalid Polygons errors

Sometimes you may get the error

```
## Error: Shape contains invalid polygons
```

This can usually be fixed quickly with `st_make_valid()`.

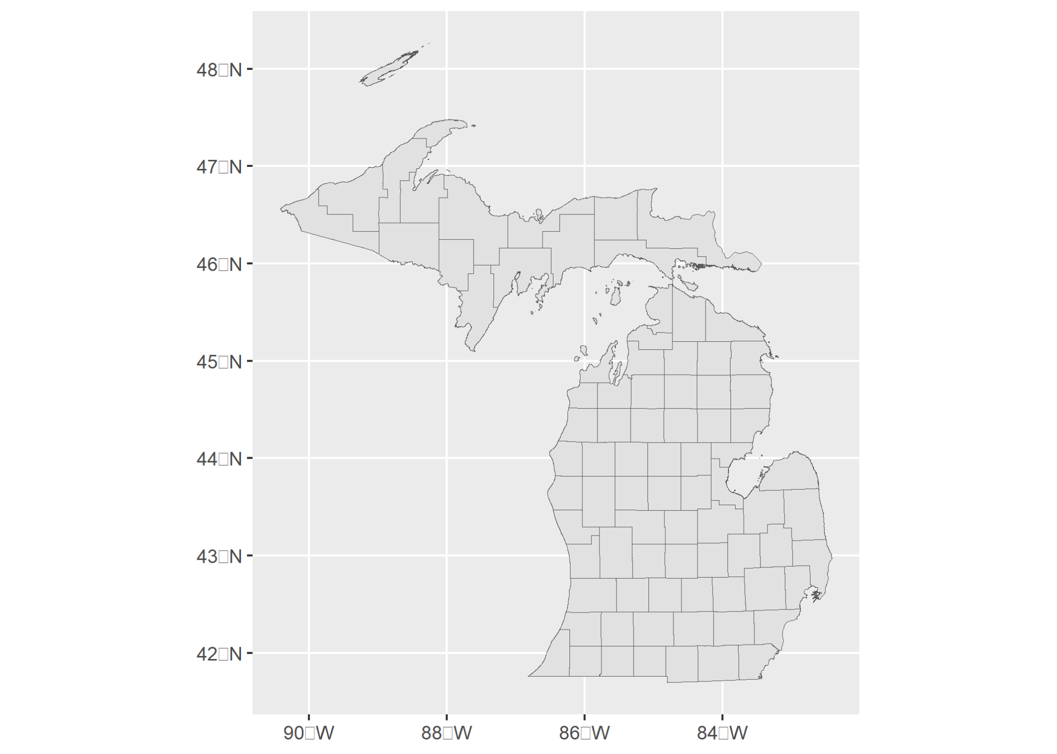
```r
counties <- st_make_valid(counties)
```

Data Wrangling Works Normally

```r
ingham <- filter(counties, NAME == "Ingham")
map_ingham <- tm_shape(counties) +
  tm_polygons(border.col = "white") +
  tm_shape(ingham) +
  tm_borders(col = "green", lwd = 3)
map_ingham
```

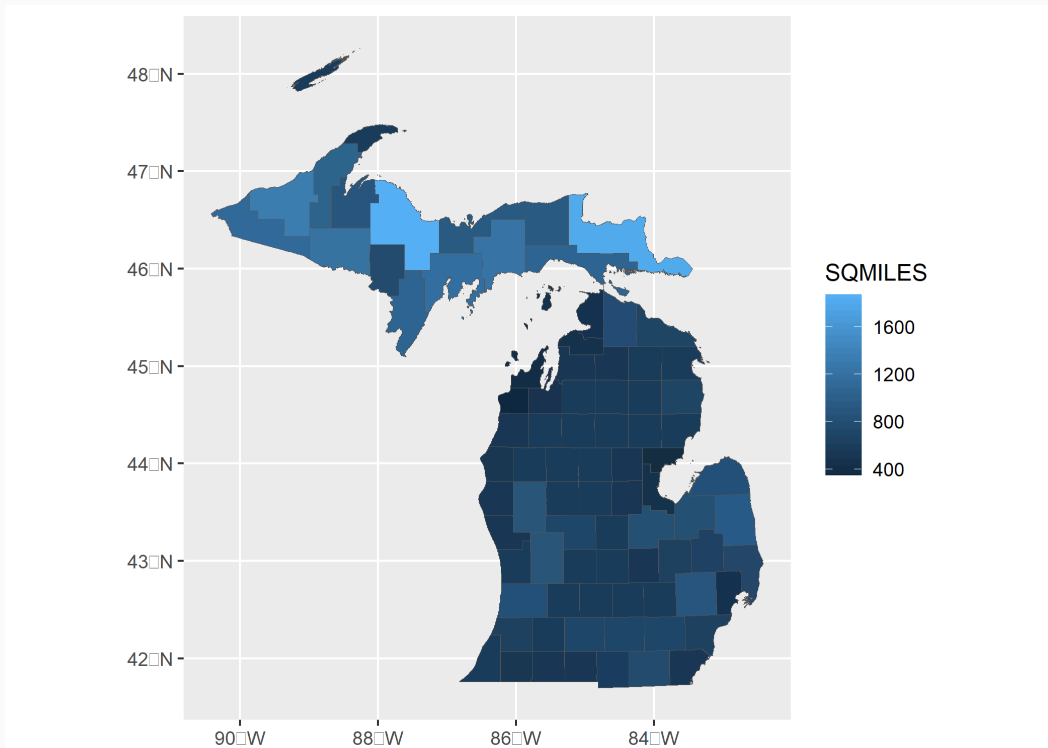
Quick Mapping: ggplot

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



Quick Mapping: ggplot

```
map_gg <- ggplot() +
  geom_sf(aes(fill = SQMILES), data = counties)
map_gg
```



Saving your Maps and Shapefiles

The save method will depend on *which* plotting approach you took.

```
# Static image, tmap
tmap_save(map_ingham, filename = "output/map_ingham.png")

# Static image, ggplot2
ggsave(map_gg, filename = "map_gg.png", path = "output/", device = "png")


# Interactive version
tmap_save(map_ingham, filename = "output/map_ingham.html")

# Shapefile
st_write(ingham, dsn = "output/ingham.shp", delete_dsn = TRUE)
```

Reference Systems and Projections

Reference Systems and Projections

As we discussed in the intro, we need

- A **coordinate reference system (CRS)** of our shapefiles to know where on earth a feature is pointing to
- A **projection** to go from our 3D datum to a 2D map

Let's work through the steps involved in setting/changing these in `sf`.

State Borders

To start, load in another shapefile of US state borders:

```
states <- st_read("data/us_states_contiguous/states_contiguous.shp")
```

```
## Reading layer `states_contiguous' from data source
##   `F:\OneDrive - Michigan State University\Teaching\MSU 2023-2024\AFRE 891 S
##   using driver `ESRI Shapefile'
## Simple feature collection with 49 features and 3 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -124.7318 ymin: 24.54547 xmax: -66.97626 ymax: 49.3843
## Geodetic CRS:  WGS 84
```

Getting the CRS

What is the CRS of `states`?

```
st_crs(states)
```

```
## Coordinate Reference System:
##   User input: WGS 84
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["latitude",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["longitude",east,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433]],
```

Getting the CRS

And the CRS of `counties`?

```
st_crs(counties)
```

```
## Coordinate Reference System:
##   User input: WGS 84
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["latitude",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433]],
##         AXIS["longitude",east,
##             ORDER[2],
```

Getting the CRS

Conveniently, these two use the same CRS: same datum (WGS 84)

```
identical(st_crs(counties), st_crs(states))
```

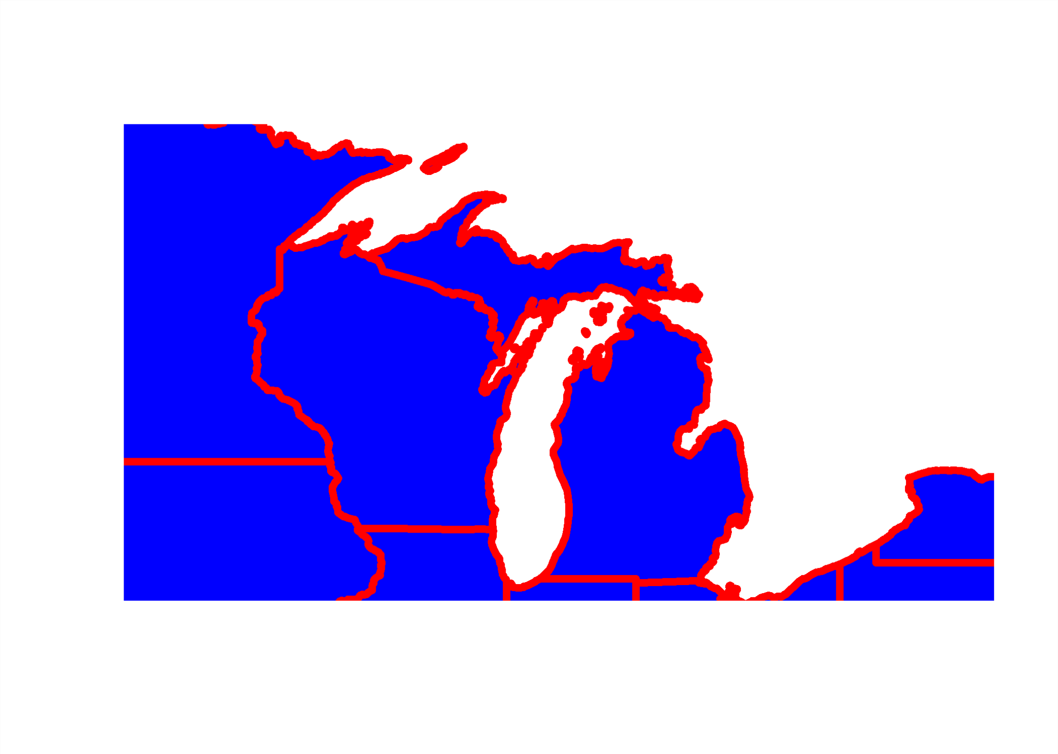
```
## [1] TRUE
```

We'll talk shortly about what to do if these *don't* match.

Plotting Two Shapefiles

And plot them both with the base `plot()` function:

```
plot(counties$geometry, col = 'lightgrey', border = 'white')
plot(states$geometry, col = 'blue', border = 'red', lwd = 5, add = T)
```



Plotting Two Shapefiles: base

What do you think happened?

`plot()` restricts to the **first object's extent**.

```
st_bbox(counties)
```

```
##      xmin      ymin      xmax      ymax 
## -90.41829  41.69613 -82.41348  48.26269 
```

```
st_bbox(states)
```

```
##       xmin       ymin       xmax       ymax 
## -124.73183   24.54547  -66.97626   49.38436 
```

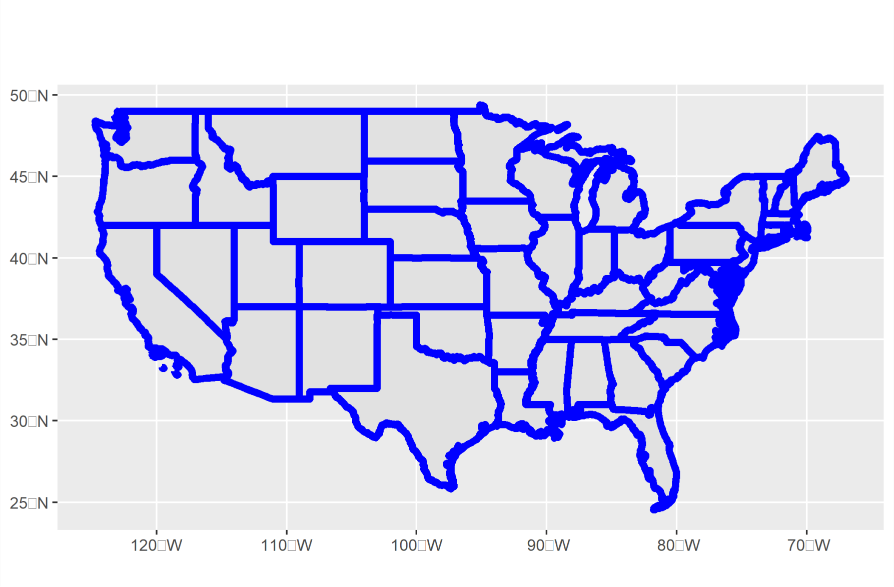
`st_bbox()` gets the bounding box dimensions as a `bbox` object

Plotting Two Shapefiles: ggplot

Note that if we do it with `ggplot()`, the default behavior is different:

- Plots the full extent of the **larger shapefile**
- Plots layers in the **opposite order** they're given in the code

```
ggplot() +
  geom_sf(data = counties, col = "forestgreen") +
  geom_sf(data = states, col = "blue", lwd = 2)
```



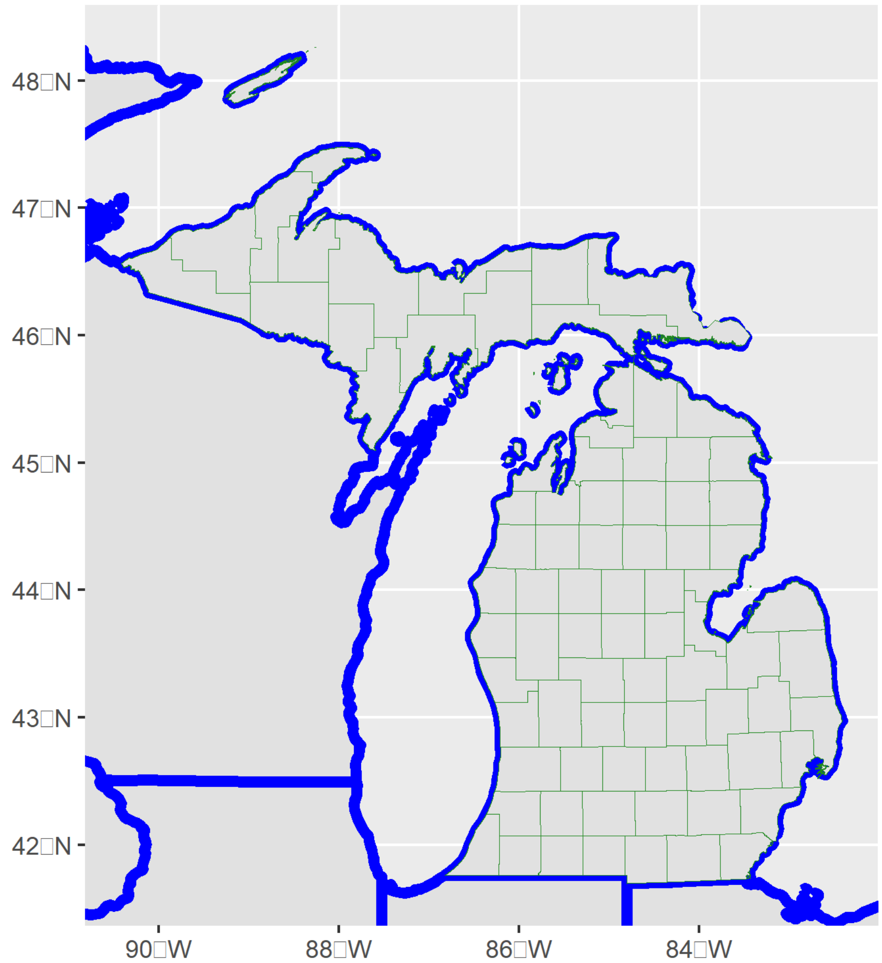
Plotting Two Shapefiles: ggplot

We can get around this with two adjustments:

- Specify extent to plot with `coord_sf()`
- Switch the layer order

```
ggplot() +
  geom_sf(data = states, col = "blue", lwd = 2) +
  geom_sf(data = counties, col = "forestgreen") +
  coord_sf(xlim = st_bbox(counties)[c("xmin", "xmax")],
           ylim = st_bbox(counties)[c("ymin", "ymax")])
```

Plotting Two Shapefiles: ggplot



EPSG codes and Projected CRS

CRSs are most commonly referenced by **EPSG codes**, which you can Google.

- The counties shapefile was in WGS 84, or EPSG:4326. (https://epsg.io/4326)

To get two shapefiles to work with each other, we need to give them the **same CRS**.

To change a CRS, we need to use a **Projected CRS**, which consists of

- A Geographic CRS
- A specific **map projection** and related parameters used to transform the geographic coordinates to a 2D plane.

Common Geographic CRSs

- **4326: WGS84** (units: decimal degrees) - most common
- **4269: NAD83** (units: decimal degrees) - best fit for USA, used by federal agencies
- NAD83 and WGS84 are *close* but can differ by up to 1 meter in the continental USA and elsewhere up to 3m.
- Context: census tract data are only accurate +/-7 meters.

Common Projected CRSs

- **5070: USA CONUS NAD83** (meters) projected CRS for mapping the entire contiguous USA (CONUS)
- **3857: Web Mercator** (meters) conformal (shape preserving) CRS used as the default in web mapping
- **26915: UTM Zone 15N, NAD83** (meters) projected CRS for MI (extreme western U.P.)
- **26916: UTM Zone 16N, NAD83** (meters) projected CRS for MI (Western/Central)
- **26917: UTM Zone 17N, NAD83** (meters) projected CRS for MI (Flint on East)

Full list/details: **https://www.spatialreference.org**

View UTM Zones on map: **mangomap.com**

Projected CRS

To align two shapefiles, we need to transform coordinates to the same **Projected CRS**

Do you *have* to use a projected CRS?

- No -- much of the time, it's fine to keep your spatial data in a geographic CRS.
- Most functions will project "on the fly" using a default.

When *should* you use a projected CRS?

- When doing calculations, like area or distance.
- When you want to control the appearance of your map output.

How should you choose a projected CRS?

- What you want to preserve (area, direction, or distance).
- Location on Earth of the area of focus.

Setting the CRS

Note that most (all?) the shapefiles we read in *should* have the right CRS already defined.

If you think the wrong CRS is defined or you manually created an sf object and need to add the CRS, use `st_crs()`

```
st_crs(counties_comb) ← 5070
```

This just **changed the metadata**, and didn't re-project the data! The warning tells us as much:

`st_crs ← : replacing crs does not reproject data; use st_transform for that`

Reprojecting the CRS

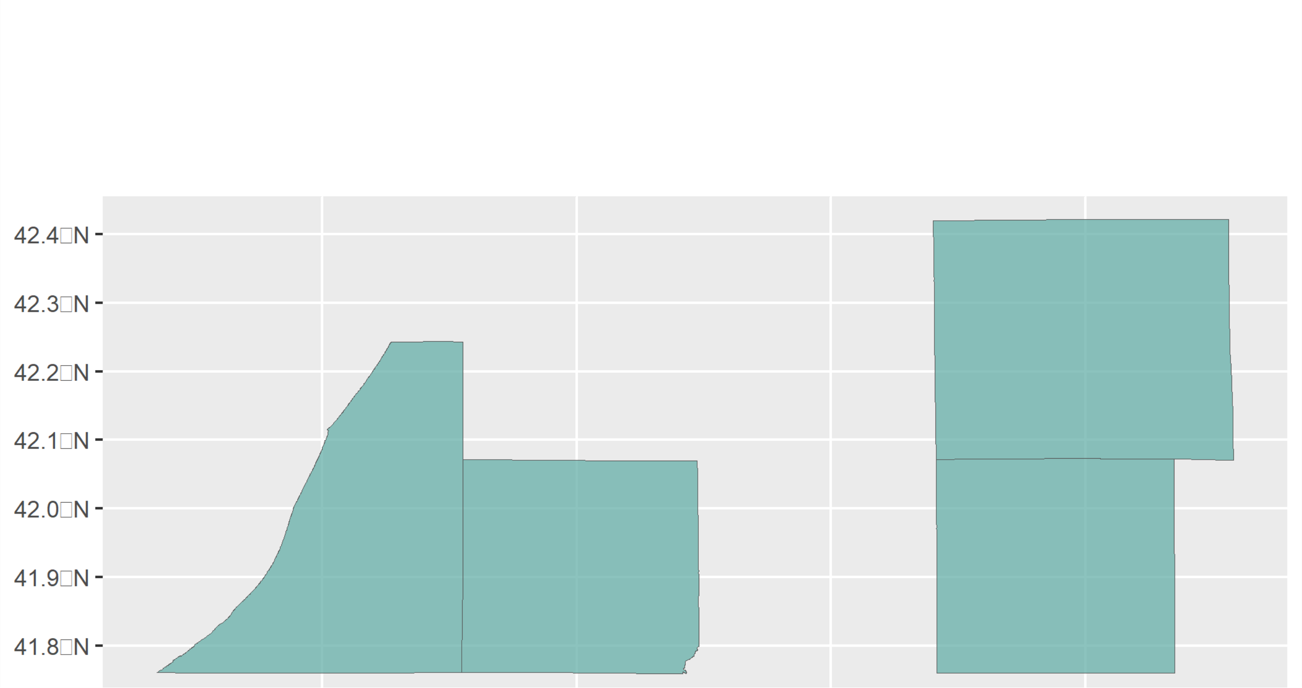
To *actually* transform (reproject) the CRS, use `st_transform()`

```
st_crs(counties_comb) <- 4326 # reset the CRS to what it was
counties_comb <- st_transform(counties_comb, crs = 3857) # project to web
```

Reprojecting the CRS

More commonly, we'll reproject to another sf object's CRS:

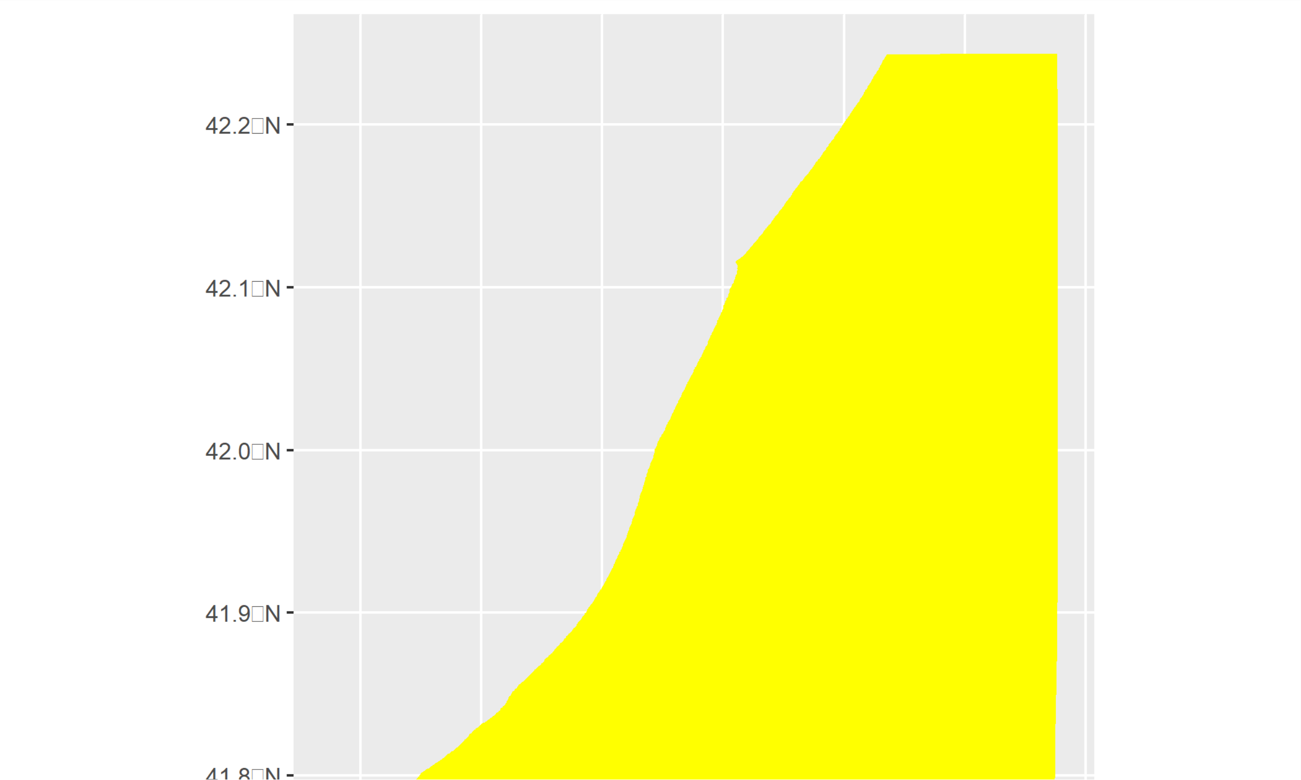
```
counties_wm <- st_transform(counties, crs = st_crs(counties_comb))

ggplot() +
  geom_sf(data = counties[11:14,], fill = "yellow", alpha = 0.5) +
  geom_sf(data = counties_wm[11:14,], fill = "dodgerblue", alpha = 0.5)
```



Reprojecting the CRS

We can see the difference when plotting the different object versions

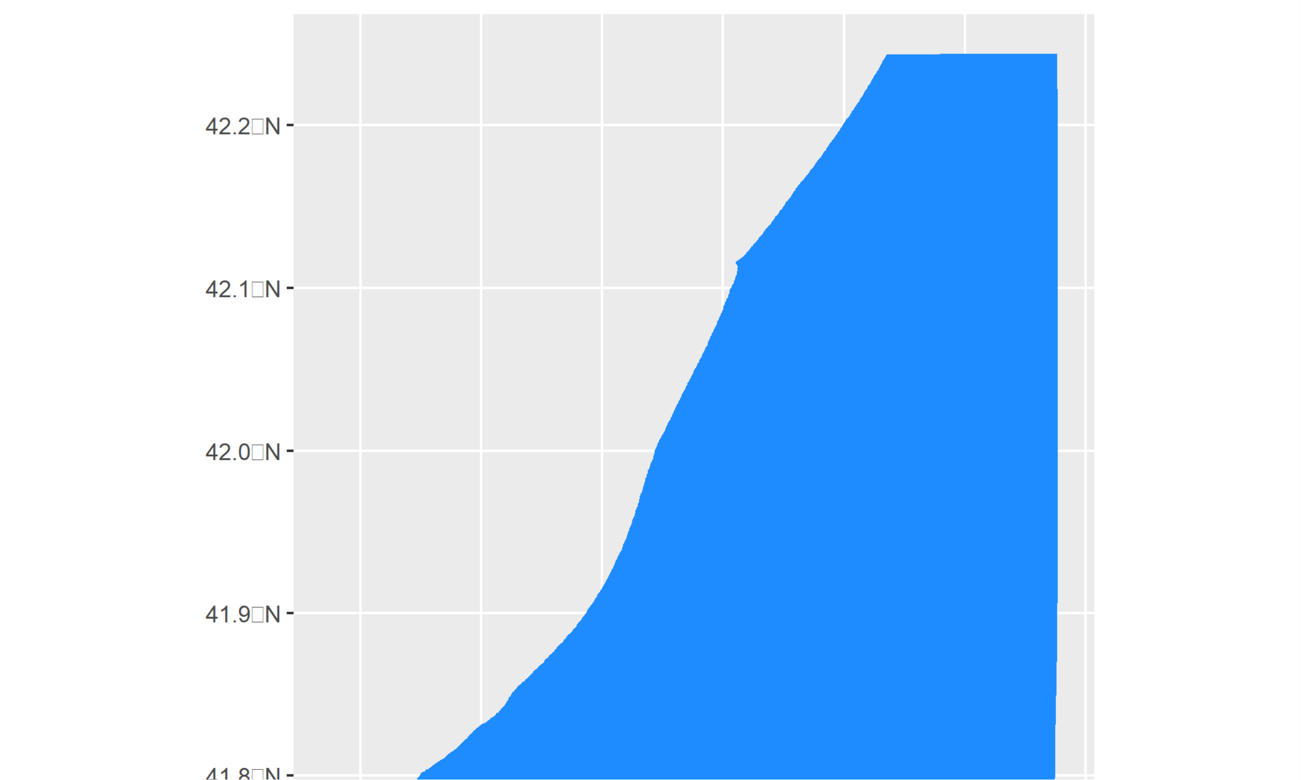
```
ggplot() +
  geom_sf(data = counties[11,],
          fill = "yellow", color = NA) #WGS 84 Geographic CRS
```



Reprojecting the CRS

We can see the difference when plotting the different object versions

```
ggplot() +
  geom_sf(data = counties_wm[11,],
          fill = "dodgerblue", color = NA) # Web Mercator Projected CRS
```



More Extreme Example

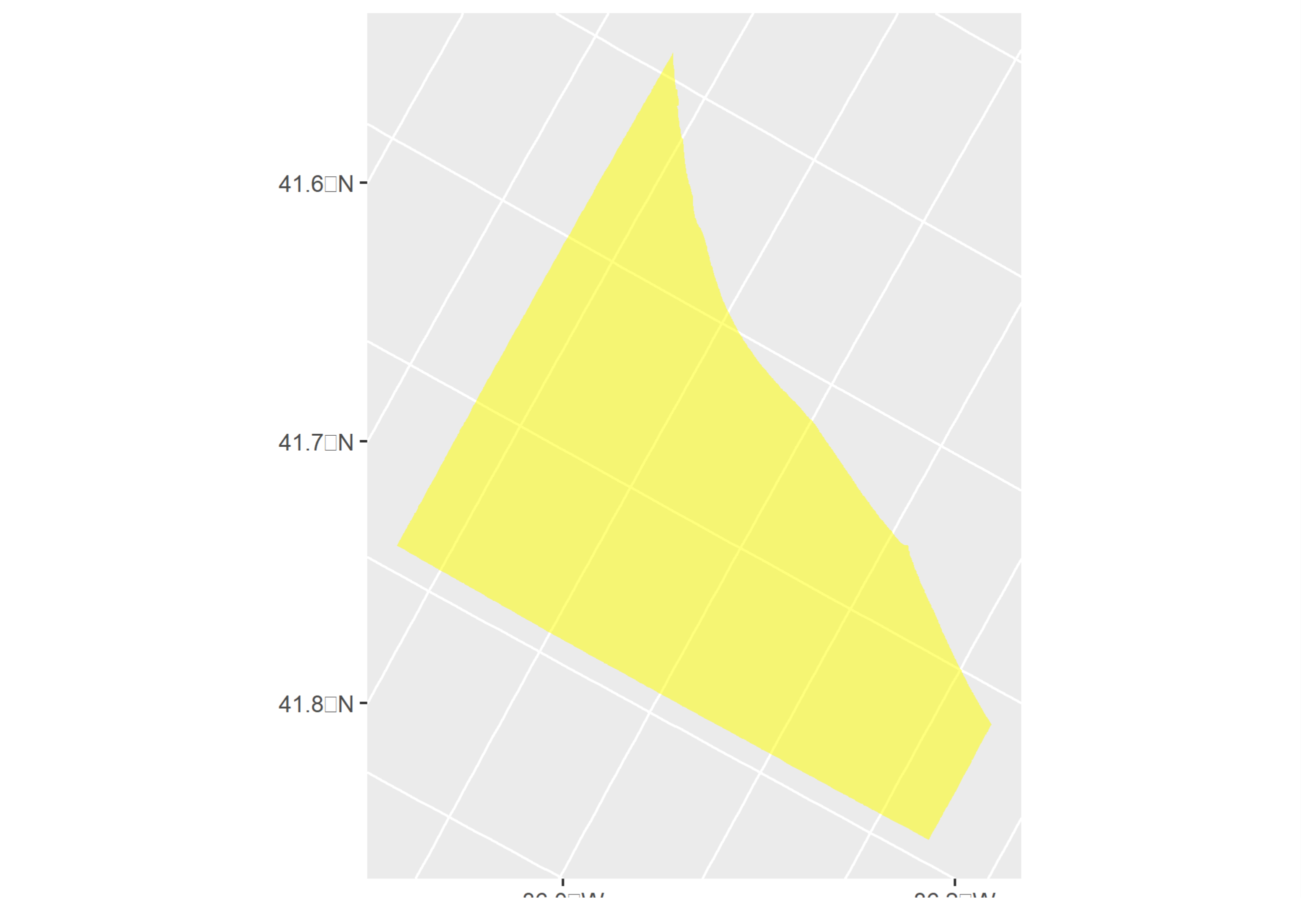
These don't actually differ, since Web Mercator is a projected version of WGS84, and matches the default projection used by ggplot for the WGS84 Object.

A more extreme example would be if we accidentally used a projection suitable for Greece (EPSG 2100):

```
counties_ex <- st_transform(counties, st_crs(2100))
```

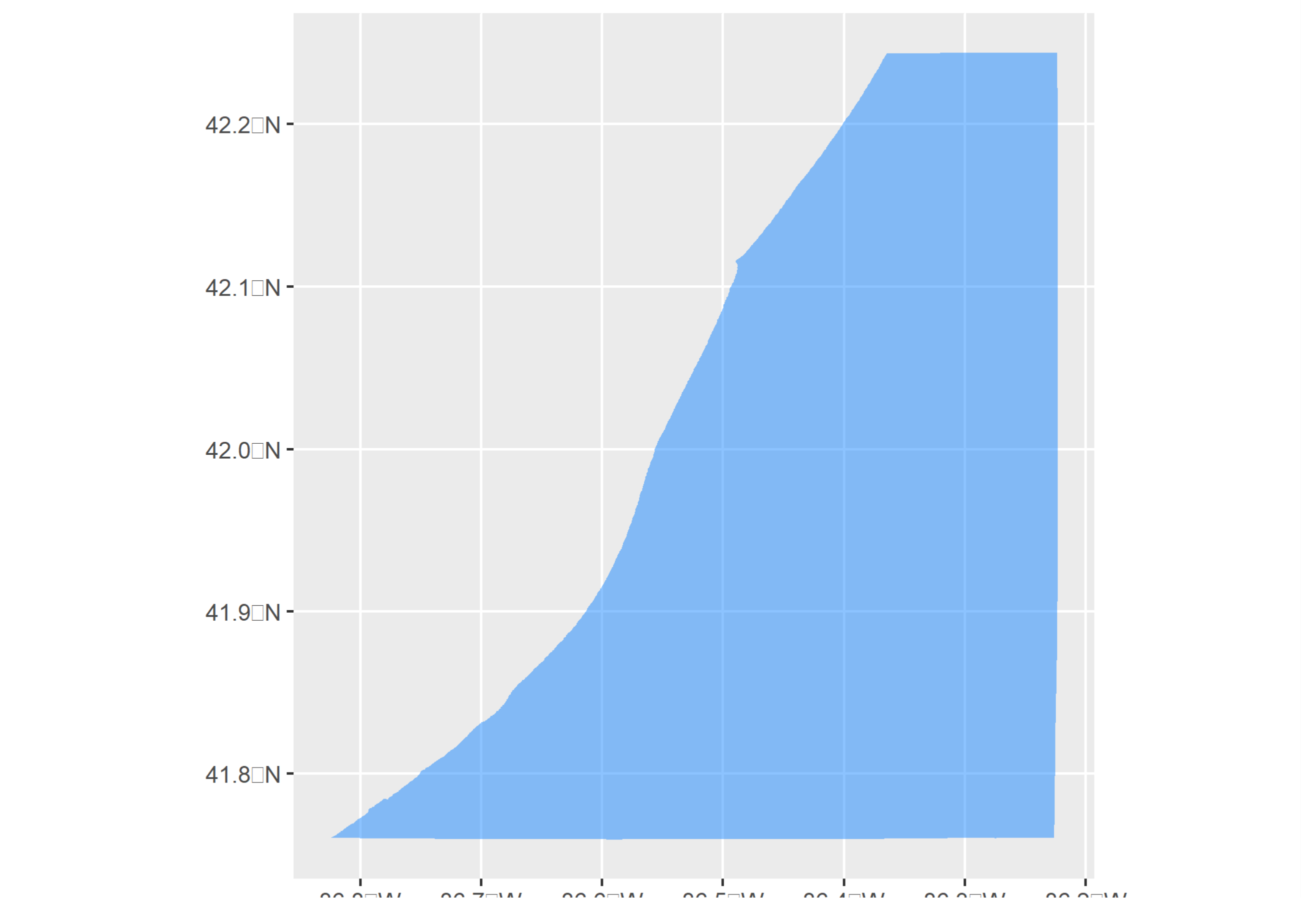
More Extreme Example

```
ggplot() +
  geom_sf(data = counties_ex[11,], fill = "yellow",
          color = NA, alpha = 0.5)  # Greek onshore projection
```



More Extreme Example

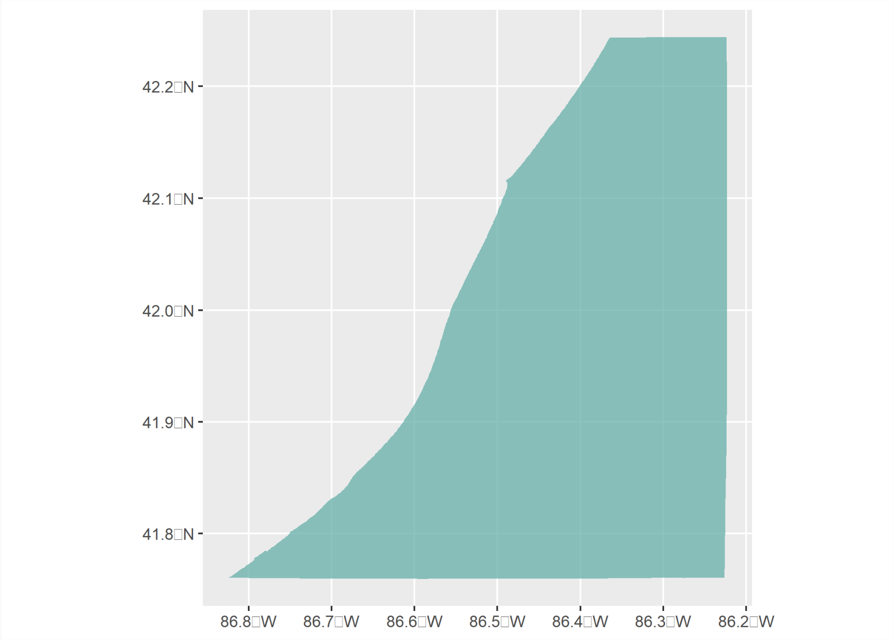
```r
ggplot() +
  geom_sf(data = counties_wm[11,], fill = "dodgerblue",
          color = NA, alpha = 0.5)  # Web Mercator Projected CRS
```



Reprojecting the CRS: in ggplot only

If we wanted to leave the sf objects in their own CRSs but map them together, we can use `coord_sf()` to reproject **within the plot**

```
ggplot() +
  geom_sf(data = counties[11,], fill = "yellow", color = NA, alpha = 0.5)
  geom_sf(data = counties_wm[11,], fill = "dodgerblue", color = NA, alpha
  coord_sf(crs = st_crs(3857)) # reproject both to web mercator
```



Common Projections

1. Web Mercator

- Preserves **direction/angle/shape** but distorts area and distance.
- Decent starting point for most places in the world.

```
states_mercator = st_transform(states, crs = 3857)
tmap_mode("plot")
qtm(states_mercator)
```

Common projections

2. U.S. National Atlas (Albers) Equal Area

- Preserves **area** but distorts direction/angle/shape and distance.
- Great for the continental U.S.

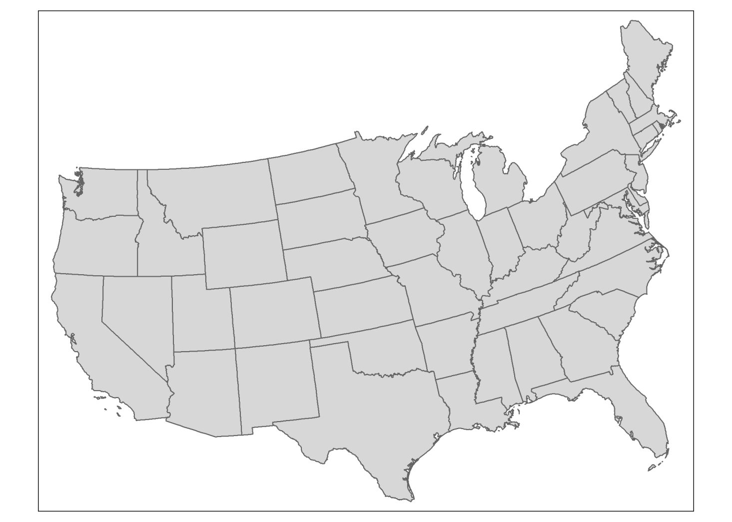
```
states_albers = st_transform(states, crs = 2163)
qtm(states_albers)
```

Common projections

3. UTM Zone 11N

- Preserves **direction/angle/shape** but distorts area and distance.
- Different UTM zones are centered at different locations.
- Good for maps of smaller areas.

```
states_utm11N = st_transform(states, crs = 2955)
qtm(states_utm11N)
```



Common projections

4. Pseudo Plate Carree

- Distorts **everything**! Simply a graph of latitude vs. longitude.
- Common default, but no excuse for using it these days.

```
plot(states$geometry, asp = 1)
```

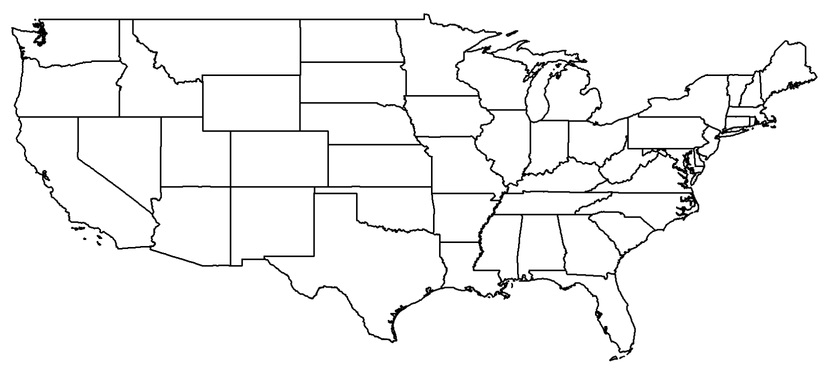


Table of Contents: Today

Part 1: Getting Started