Big Data and Economics

Spatial analysis in R

Kyle Coombs (adapted from Grant McDermott)

Bates College | ECON/DCS 368

Contents

*Note: This lecture will focus only on **vector-based** spatial analysis. We will not cover **raster-based** spatial analysis, although this is an equally important subject. Here is a link to it a lecture on it by Grant McDermott. There are further resources below. Rasters are used to make satellite images, relief maps, etc.*

Requirements

External libraries (requirements vary by OS)

We're going to be doing all our spatial analysis and plotting today in R. Behind the scenes, R provides bindings to powerful open-source GIS libraries. These include the Geospatial Data Abstraction Library (GDAL) and Interface to Geometry Engine Open Source (GEOS) API suite, as well as access to projection and transformation operations from the PROJ library. You needn't worry about all this, but for the fact that you *may* need to install some of these external libraries first. The requirements vary by OS:

- **Linux:** Requirements vary by distribution. See here.
- **Mac:** You should be fine to proceed directly to the R packages installation below. An unlikely exception is if you've configured R to install packages from source; in which case see here.
- **Windows:** Same as Mac, you should be good to go unless you're installing from source. In which case, see here.

R packages

- New: **sf**, **lwgeom**, **maps**, **mapdata**, **spData**, **tigris**, **tidycensus**, **leaflet**, **mapview**, **tmap**, **tmaptools**, **nngeo**
- Already used: **tidyverse**, **data.table**, **hrbrthemes**

Truth be told, you only need a handful of the above libraries to do 95% of the spatial work that you're likely to encounter. But R's spatial ecosystem and support is extremely rich, so I'll try to walk through a number of specific use-cases in this lecture. Run the following code chunk to install (if necessary) and load everything.

```
## Load and install the packages that we'll be using today
if (!require("pacman")) install.packages("pacman")
pacman::p_load(sf, tidyverse, data.table, hrbrthemes, lwgeom, rnaturalearth, maps, mapdata, spData, tig
## My preferred ggplot2 plotting theme (optional)
theme_set(theme_minimal())
```

Census API key

Finally, we'll be accessing some data from the US Census Bureau through the tidycensus package. This will require a Census API key, which you can request here. Once that's done, you can set it using the `tidycensus::census_api_key()` function. I recommend using the "install = TRUE" option to save your key for future usage. See the function's help file for more information.

```
tidycensus::census_api_key("PLACE_YOUR_API_KEY_HERE", install = TRUE)
```

Also tell the tigris package to automatically cache its results to save on repeated downloading. I recommend adding this line to your ~/.Rprofile file so that caching is automatically enabled for future sessions. A quick way to do that is with the `usethis::edit_r_profile()` function.

```
options(tigris_use_cache=TRUE)
```

Introduction: CRS and map projections

Student presentation time.

If you're reading this after the fact, I recommend these two helpful resources. The very short version is that spatial data, like all coordinate-based systems, only make sense relative to some fixed point. That fixed point is what the Coordinate Reference Systems, or **CRS**, is trying to set. In R, we can define the CRS in one of two ways:

1. EPSG code (e.g. 3857), or
2. PROJ string (e.g. `"+proj=merc"`).

We'll see examples of both implementations in in this lecture. For the moment, however, just know that they are equally valid ways of specifying CRS in R (albeit with with different strengths and weaknesses). You can search for many different CRS definitions here.

> **Aside:** There are some important updates happening in the world of CRS and geospatial software, which will percolate through to the R spatial ecosystem. Thanks to the hard work of various R package developers, these behind-the-scenes changes are unlikely to affect the way that you interact with spatial data in R. But they are worth understanding if you plan to make geospatial work a core component of your research. More here.

Similarly, whenever we try to plot (some part of) the earth on a map, we're effectively trying to **project** a 3-D object onto a 2-D surface. This will necessarily create some kind of distortion. Different types of map projections limit distortions for some parts of the world at the expense of others. For example, consider how badly the standard (but infamous) Mercator projection distorts the high latitudes in a global map (source):

```
## Sorry, this GIF is only available in the the HTML version of the notes.
```

Bottom line: You should always aim to choose a projection that best represents your specific area of study. I'll also show you how you can "re-orient" your projection to a specific latitude and longitude using the PROJ syntax. But first I'm obliged to share this XKCD summary. (Do yourself a favour and click on the link.)

Simple Features and the sf package

R has long provided excellent support for spatial analysis and plotting (primarily through the **sp**, **rgdal**, **rgeos**, and **raster** packages). However, until recently, the complex structure of spatial data necessitated a set of equally complex spatial objects in R. I won't go into details, but a spatial object (say, a SpatialPolygonsDataFrame) was typically comprised of several "layers" — much like a list — with each layer containing a variety of "slots". While this approach did (and still does) work perfectly well, the convoluted structure provided some barriers to entry for newcomers. It also made it very difficult to incorporate spatial data into the tidyverse ecosystem that we're familiar with. Luckily, all this has changed thanks to the advent of the **sf** package (link).

The "sf" stands for **s**imple **f**eatures, which is a simple (ahem) standard for representing the spatial geometries of real-world objects on a computer.[1] These objects — i.e. "features" — could include a tree, a building, a country's border, or the entire globe. The point is that they are characterised by a common set of rules, defining everything from how they are stored on our computer to which geometrical operations can be applied them. Of greater importance for our purposes, however, is the fact that **sf** represents these features in R as *data frames*. This means that all of our data wrangling skills from previous lectures can be applied to spatial data; say nothing of the specialized spatial functions that we'll cover next.

Reading in spatial data

Somewhat confusingly, most of the functions in the **sf** package start with the prefix `st_`. This stands for *spatial and temporal* and a basic command of this package is easy enough once you remember that you're probably looking for `st_SOMETHING()`.[2]

Let's demonstrate by reading in a shapefile for Maine.[3] As you might have guessed, we're going to use the `st_read()` command and **sf** package will handle all the heavy lifting behind the scenes. Shapefile are a "geospatial vector data format" that is widely used in GIS software that typically have the file ending `.shp`.[4] Basically, it contains the shape of each feature (e.g. county) represented as coordinates, along with a bunch of other information about the feature (e.g. county name, population, etc.).

How do we get the shapefile? Well the Census has loads that cover all kinds of different geographies at TIGER, which stands for Topologically Integrated Geographic Encoding and Referencing. You could go in and download any shapefile you want OR you could use a package like **tidycensus** or **tigris** to download it for you. You're already familiar with **tidycensus**, which you may have noticed you can get it to return `geometry` as a variable if you specify `geometry=TRUE`. This is a vector that **sf** can use.

```
me = tidycensus::get_acs(
  geography = "county",
  variables = "B01001_001",
  state = "ME",
  geometry = TRUE
  )
```

```
[**tigris**](https://github.com/walkerke/tigris) is explained in greater detail below, but it contains
```

```
```r
# library(tigris) ## Already loaded
me = tigris::counties(state='ME',
  year=2010)
```

Note: We are using the year 2010 to link it to the Opportunity Atlas.

[1] See the first of the excellent **sf** vignettes for more details.

[2] I rather wish they'd gone with a `sf_` prefix myself — or at least created aliases for it — but the package developers are apparently following standard naming conventions from PostGIS.

[3] **sf** includes the North Carolina county shapefile, but we're in Maine.

[4] See here for more details.
```

Simple Features as data frames

Let's print out the me object that we just created and take a look at its structure.

```
me
```

```
## Simple feature collection with 16 features and 19 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -71.08392 ymin: 42.91713 xmax: -66.88544 ymax: 47.45985
## Geodetic CRS:  NAD83
## First 10 features:
##      STATEFP10 COUNTYFP10 COUNTYNS10 GEOID10     NAME10        NAMELSAD10
## 45          23        019   00581295   23019  Penobscot  Penobscot County
## 231         23        029   00581300   23029 Washington Washington County
## 239         23        003   00581287   23003  Aroostook  Aroostook County
## 257         23        009   00581290   23009    Hancock    Hancock County
## 3009        23        007   00581289   23007   Franklin   Franklin County
## 3010        23        025   00581298   23025   Somerset   Somerset County
## 3011        23        017   00581294   23017     Oxford     Oxford County
## 3014        23        027   00581299   23027      Waldo      Waldo County
## 3015        23        015   00581293   23015    Lincoln    Lincoln County
## 3016        23        031   00581301   23031       York       York County
##      LSAD10 CLASSFP10 MTFCC10 CSAFP10 CBSAFP10 METDIVFP10 FUNCSTAT10
## 45       06        H1   G4020    <NA>    12620       <NA>          A
## 231      06        H1   G4020    <NA>     <NA>       <NA>          A
## 239      06        H1   G4020    <NA>     <NA>       <NA>          A
## 257      06        H1   G4020    <NA>     <NA>       <NA>          A
## 3009     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3010     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3011     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3014     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3015     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3016     06        H1   G4020     438    38860       <NA>          A
##          ALAND10   AWATER10  INTPTLAT10   INTPTLON10
## 45    8799125852  413670635 +45.3906022 -068.6574869
## 231   6637257545 1800019787 +44.9670088 -067.6093542
## 239  17278664655  404653951 +46.7270567 -068.6494098
## 257   4110034060 1963321064 +44.5649063 -068.3707034
## 3009  4394196449  121392907 +44.9730124 -070.4447268
## 3010 10164156961  438038365 +45.5074824 -069.9760395
## 3011  5378990983  256086721 +44.4945850 -070.7346875
## 3014  1890479704  318149622 +44.5053607 -069.1396775
## 3015  1180563700  631400289 +43.9942645 -069.5140292
## 3016  2565935077  722608929 +43.4272386 -070.6704023
##                           geometry COUNTYFP STATEFP
## 45   MULTIPOLYGON (((-69.28127 4...      019      23
## 231  MULTIPOLYGON (((-67.7543 45...      029      23
## 239  MULTIPOLYGON (((-68.43227 4...      003      23
## 257  MULTIPOLYGON (((-68.80096 4...      009      23
## 3009 MULTIPOLYGON (((-70.29383 4...      007      23
## 3010 MULTIPOLYGON (((-69.85327 4...      025      23
## 3011 MULTIPOLYGON (((-71.05825 4...      017      23
## 3014 MULTIPOLYGON (((-69.26888 4...      027      23
## 3015 MULTIPOLYGON (((-69.69056 4...      015      23
```

```
## 3016 MULTIPOLYGON (((-70.76779 4...      031      23
```

Now we can see the explicit data frame structure. The object has the familiar tibble-style output that we're used to (e.g. it only prints the first 10 rows of the data). However, it also has some additional information in the header, like a description of the geometry type ("MULTIPOLYGON") and CRS (e.g. EPSG ID 4267). One thing I want to note in particular is the `geometry` column right at the end of the data frame. This geometry column is how **sf** package achieves much of its magic: It stores the geometries of each row element in its own list column.[5] Since all we really care about are the key feature attributes — county name, FIPS code, population size, etc. — we can focus on those instead of getting bogged down by hundreds (or thousands or even millions) of coordinate points. In turn, this all means that our favourite **tidyverse** operations and syntax (including the pipe operator `%>%`) can be applied to spatial data. Let's review some examples, starting with plotting.

Plotting and projection with ggplot2

Plotting **sf** objects is incredibly easy thanks to the package's integration with both base R `plot()` and **ggplot2**. I'm going to focus on the latter here, but feel free to experiment.[6] The key geom to remember is `geom_sf()`. For example:

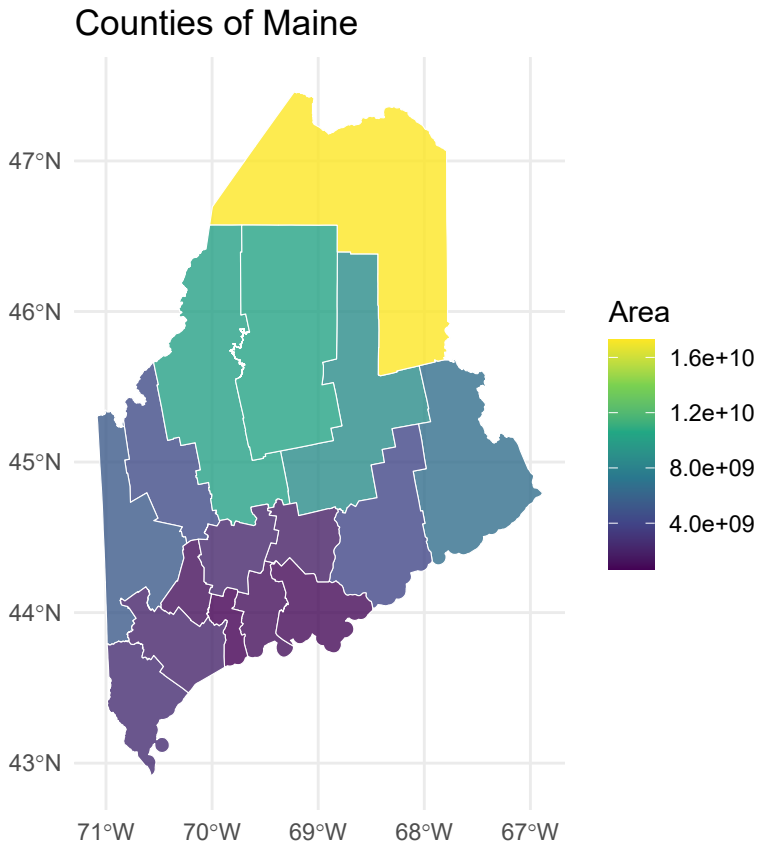
```
# library(tidyverse) ## Already loaded

me_plot =
  ggplot(me) +
  geom_sf(aes(fill = ALAND10), alpha=0.8, col="white") +
  scale_fill_viridis_c(name = "Area") +
  ggtitle("Counties of Maine")

me_plot
```

[5] For example, we could print out the coordinates needed to plot the first element in our data frame, Ashe county, by typing `me$geometry[[1]]`. In contrast, I invite you to see how complicated the structure of a traditional spatial object is by running, say, `str(as(me, "Spatial"))`.

[6] Plotting **sf** objects with the base `plot` function is generally faster. However, I feel that you give up a lot of control and intuition by moving away from the layered, "graphics of grammar" approach of **ggplot2**.



To reproject an **sf** object to a different CRS, we can use `sf::st_transform()`.

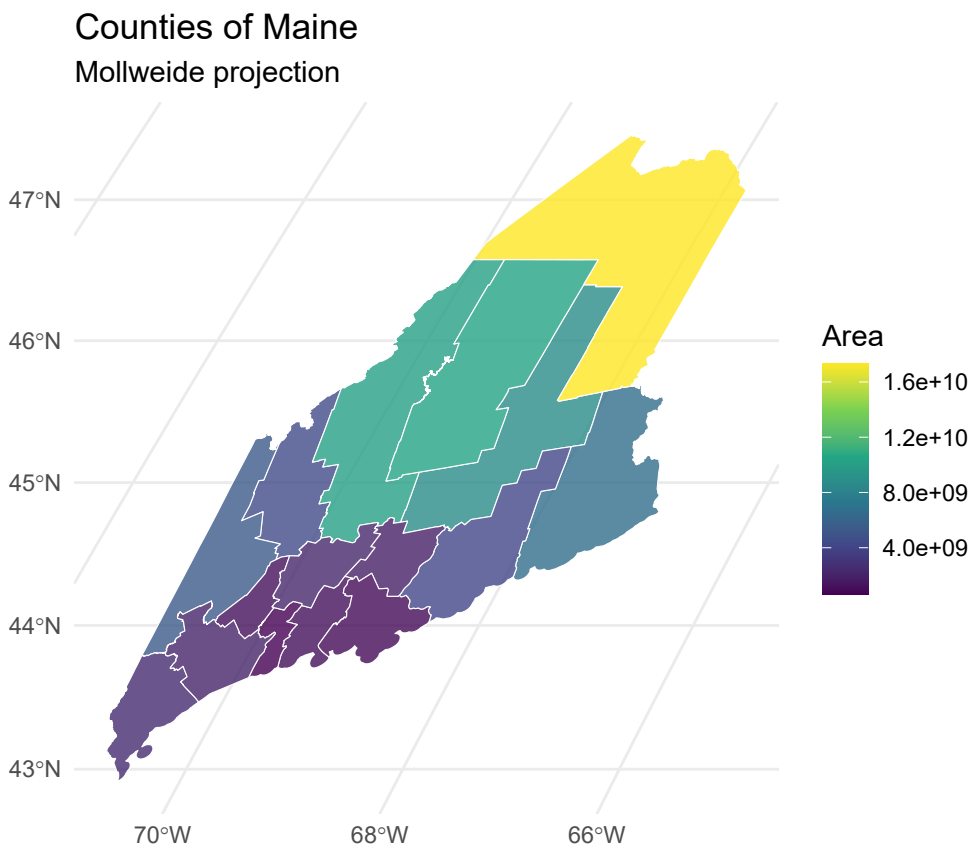
```
me %>%
  st_transform(crs = "+proj=moll") %>% ## Reprojecting to a Mollweide CRS
  head(2) ## Saving vertical space
```

```
## Simple feature collection with 2 features and 19 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -5611693 ymin: 5265977 xmax: -5411199 ymax: 5490974
## Projected CRS: +proj=moll
##     STATEFP10 COUNTYFP10 COUNTYNS10 GEOID10     NAME10        NAMELSAD10 LSAD10
## 45         23         019   00581295   23019  Penobscot  Penobscot County     06
## 231        23         029   00581300   23029 Washington Washington County     06
##     CLASSFP10 MTFCC10 CSAFP10 CBSAFP10 METDIVFP10 FUNCSTAT10    ALAND10
## 45         H1   G4020    <NA>    12620       <NA>          A 8799125852
## 231        H1   G4020    <NA>     <NA>       <NA>          A 6637257545
##       AWATER10  INTPTLAT10   INTPTLON10                       geometry COUNTYFP
## 45   413670635 +45.3906022 -068.6574869 MULTIPOLYGON (((-5607577 53...      019
## 231 1800019787 +44.9670088 -067.6093542 MULTIPOLYGON (((-5432146 54...      029
##     STATEFP
## 45       23
## 231      23
```

Or, we can specify a common projection directly in the ggplot call using `coord_sf()`. This is often the most convenient approach when you are combining multiple **sf** data frames in the same plot.

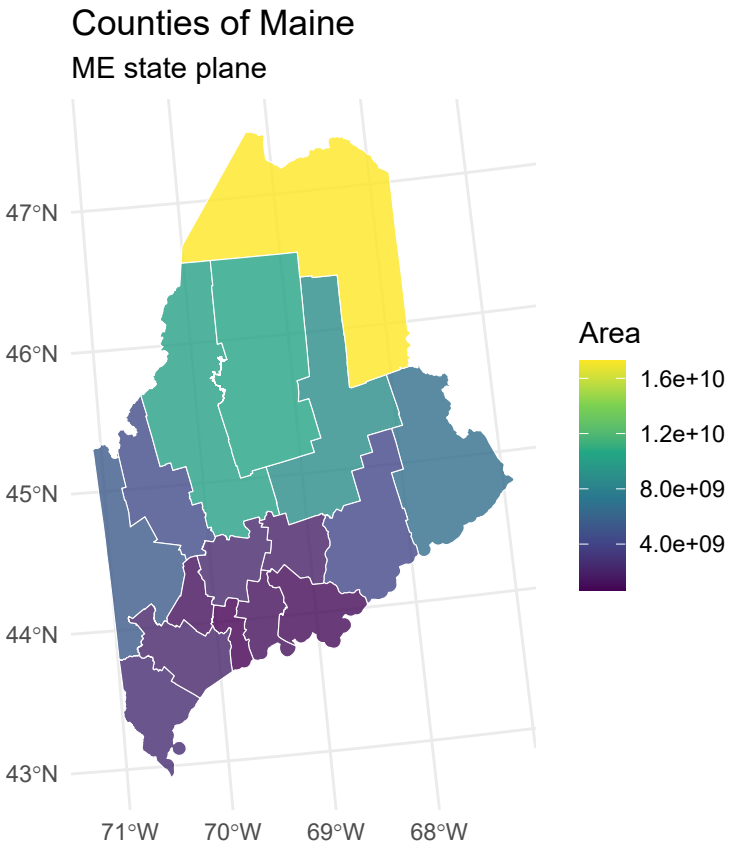
```
me_plot +
  coord_sf(crs = "+proj=moll") +
```

```
  labs(subtitle = "Mollweide projection")
```



Note that we used a PROJ string to define the CRS reprojection above. But we could easily use an EPSG code instead. For example, here's the me state plane projection.

```
me_plot +
  coord_sf(crs = 32119) +
  labs(subtitle = "ME state plane")
```



Data wrangling with dplyr and tidyr

As I keep saying, the tidyverse approach to data wrangling carries over very smoothly to **sf** objects. For example, the standard **dplyr** verbs like `filter()`, `mutate()` and `select()` all work:

```
me %>%
  filter(NAME10 %in% c("Sagadahoc", "Androscoggin", "Cumberland")) %>%
  mutate(ALAND_div_1000 = ALAND10/1000) %>%
  select(NAME10, contains("ALAND"), everything())
```

```
## Simple feature collection with 3 features and 20 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -70.86662 ymin: 43.46688 xmax: -69.66474 ymax: 44.48722
## Geodetic CRS:  NAD83
##         NAME10    ALAND10 ALAND_div_1000 STATEFP10 COUNTYFP10 COUNTYNS10
## 1    Sagadahoc  657066836       657066.8        23        023   00581297
## 2 Androscoggin 1211926439      1211926.4        23        001   00581286
## 3   Cumberland 2163263369      2163263.4        23        005   00581288
##   GEOID10          NAMELSAD10 LSAD10 CLASSFP10 MTFCC10 CSAFP10 CBSAFP10
## 1   23023    Sagadahoc County     06        H1   G4020     438    38860
## 2   23001 Androscoggin County     06        H1   G4020     438    30340
## 3   23005   Cumberland County     06        H1   G4020     438    38860
##   METDIVFP10 FUNCSTAT10  AWATER10  INTPTLAT10   INTPTLON10 COUNTYFP STATEFP
## 1       <NA>          A 301332007 +43.9166939 -069.8439936      023      23
## 2       <NA>          A  75610678 +44.1676811 -070.2074347      001      23
## 3       <NA>          A 989949911 +43.8083479 -070.3303753      005      23
##                         geometry
```

```
## 1 MULTIPOLYGON (((-69.78507 4...
## 2 MULTIPOLYGON (((-70.07575 4...
## 3 MULTIPOLYGON (((-69.89595 4...
```

You can also perform `group_by()` and `summarise()` operations as per normal (see here for a nice example). Furthermore, the **dplyr** family of join functions also work, which can be especially handy when combining different datasets by (say) FIPS code or some other *attribute*. However, this presumes that only one of the objects has a specialized geometry column. In other words, it works when you are joining an **sf** object with a normal data frame. In cases where you want to join two **sf** objects based on their *geometries*, there's a specialized `st_join()` function. I provide an example of this latter operation in the section on geometric operations below.

Let's try this by joining on the Opportunity Atlas county level data, which we can get from Opportunity Insights or the Census.

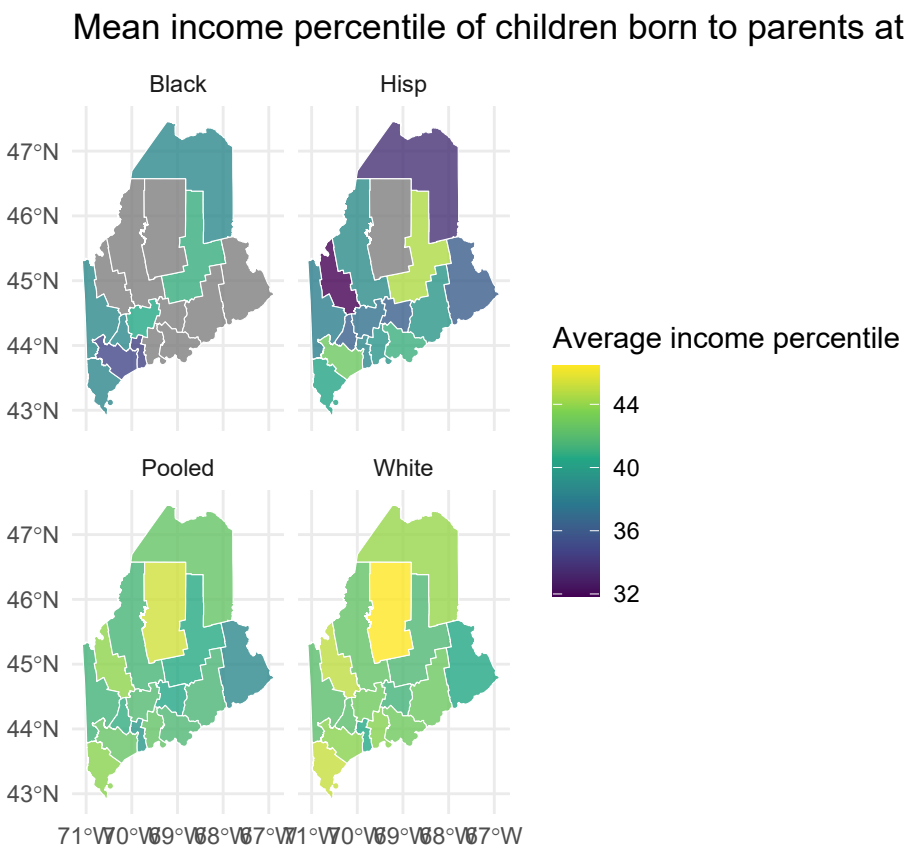
```
op_atlas <- read_csv(
    file='https://www2.census.gov/ces/opportunity/county_outcomes_simple.csv',
    show_col_types = FALSE
) %>%
  mutate(county_fips=state*1000+county,
  #Make character and add leading zero to county_fips
        GEOID10=ifelse(county_fips<10000,paste0("0",county_fips),as.character(county_fips))) %>%
  select(GEOID10,matches('kfr_.*_pooled_p25$')) %>%
  #Rename columns to remove _pooled_p25
  rename_with(~str_remove(.,'_pooled_p25'),matches('_pooled_p25$'))

me_join <- me %>% inner_join(op_atlas, by="GEOID10")
```

And, just to show that we've got the bases covered, you can also implement your favourite **tidyr** verbs. For example, we can `tidyr::gather()` the data to long format, which is useful for facetted plotting.[7] Here I demonstrate using the "BIR74" and "BIR79" columns (i.e. the number of births in each county in 1974 and 1979, respectively).

```
me_join %>%
  select(county = NAME10, matches('kfr'), geometry) %>%
  # Gather reshapes the data.
  # The key (where the variable names go) is "race." The values of each variable go into a variable nam
  gather(key=race, value=kfr, kfr_pooled, kfr_black, kfr_hisp, kfr_white) %>%
  mutate(race = str_to_title(gsub("kfr_", "", race)),
    kfr=kfr*100) %>% # put in percent terms
  ggplot() +
  geom_sf(aes(fill = kfr), alpha=0.8, col="white") +
  scale_fill_viridis_c(name = "Average income percentile") +
  facet_wrap(~race, ncol = 2) +
  labs(title = "Mean income percentile of children born to parents at the 25th percentile")
```

[7] In case you're wondering: the newer `tidyr::pivot_*` functions do not yet work with **sf** objects.



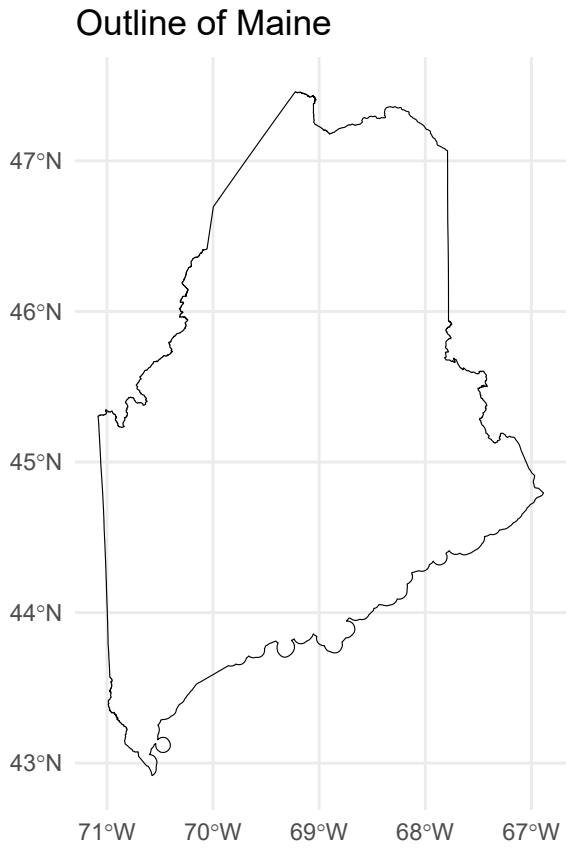
On your problem set, you'll have to do something similar at the Census tract level, so take note.

Specialized geometric operations

Alongside all the tidyverse functionality, the **sf** package comes with a full suite of geometrical operations. You should take a look at at the third **sf** vignette or the *Geocomputation with R* book to get a complete overview. However, here are a few examples to get you started:

Unary operations So-called *unary* operations are applied to a single object. For instance, you can "melt" sub-elements of an **sf** object (e.g. counties) into larger elements (e.g. states) using `sf::st_union()`:

```
me %>%
  st_union() %>%
  ggplot() +
  geom_sf(fill=NA, col="black") +
  labs(title = "Outline of Maine")
```



Or, you can get the `st_area()`, `st_centroid()`, `st_boundary()`, `st_buffer()`, etc. of an object using the appropriate command. For example:

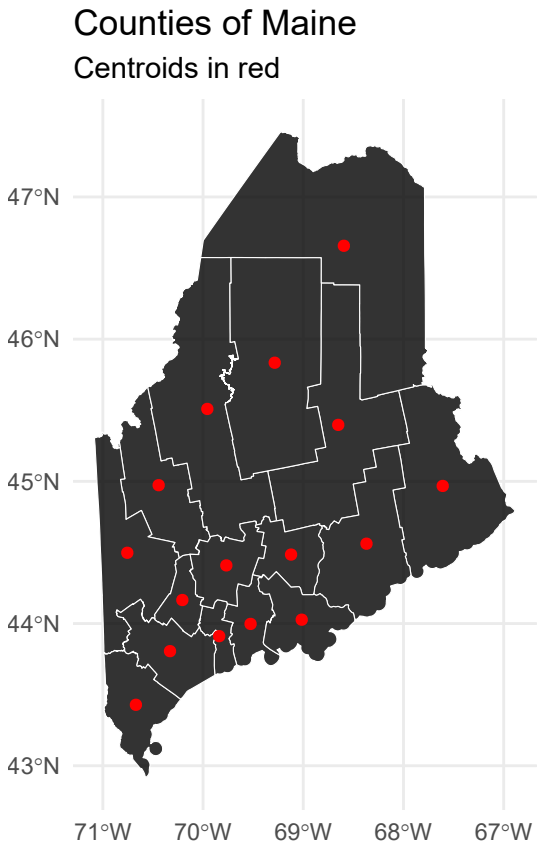
```
me %>% st_area() %>% head(5) ## Only show the area of the first five counties to save space.
```

```
## Units: [m^2]
## [1]  9191355157  8418644946 17637152538  6060468737  4506281023
```

And:

```
me_centroid = st_centroid(me)

ggplot(me) +
  geom_sf(fill = "black", alpha = 0.8, col = "white") +
  geom_sf(data = me_centroid, col = "red") + ## Notice how easy it is to combine different sf objects
  labs(
    title = "Counties of Maine",
    subtitle = "Centroids in red"
    )
```



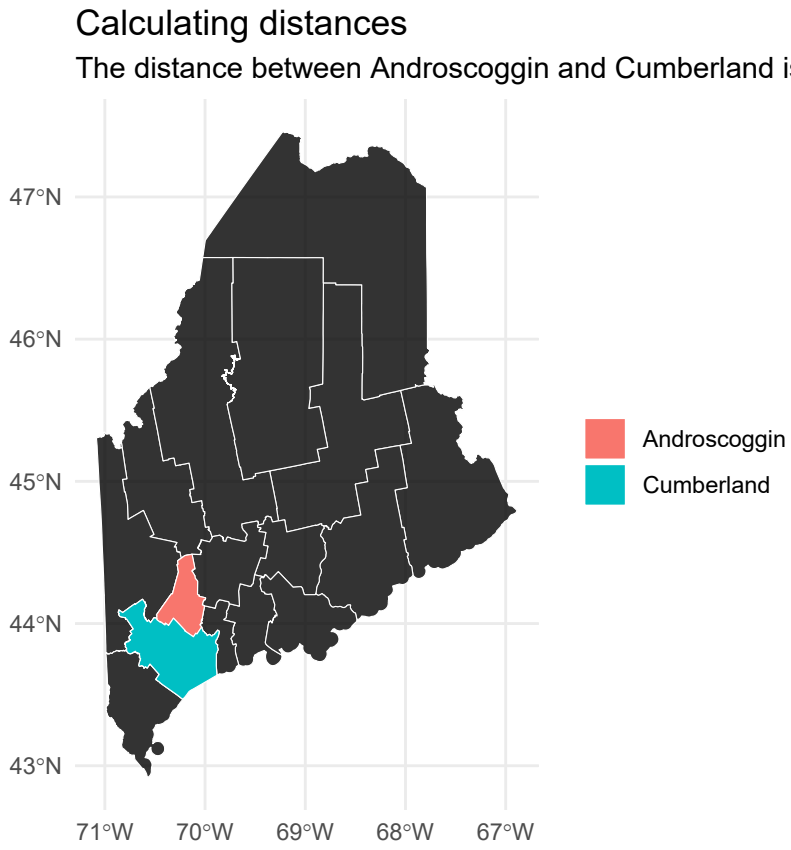
Binary operations Another set of so-called *binary* operations can be applied to multiple objects. So, we can get things like the distance between two spatial objects using `sf::st_distance()`. In the below example, I'm going to get the distance from Ashe county to Brunswick county, as well as itself. The latter is just a silly addition to show that we can easily make multiple pairwise comparisons, even when the distance from one element to another is zero.

```
andro_cumby = me %>% filter(NAME10 %in% c("Androscoggin", "Cumberland"))
cumby = me %>% filter(NAME10 %in% c("Cumberland"))

ac_dist = st_distance(andro_cumby, cumby)

## We can use the `units` package (already installed as sf dependency) to convert to kilometres
ac_dist = ac_dist %>% units::set_units(km) %>% round()

ggplot(me) +
  geom_sf(fill = "black", alpha = 0.8, col = "white") +
  geom_sf(data = me %>% filter(NAME10 %in% c("Androscoggin", "Cumberland")), aes(fill = NAME10), col =
  labs(
    title = "Calculating distances",
    subtitle = paste0("The distance between Androscoggin and Cumberland is ", ac_dist[1], " km")
    ) +
  theme(legend.title = element_blank())
```



Binary logical operations A sub-genre of binary geometric operations falls into the category of logic rules — typically characterising the way that geometries relate in space. (Do they overlap, are they nearby, etc.)

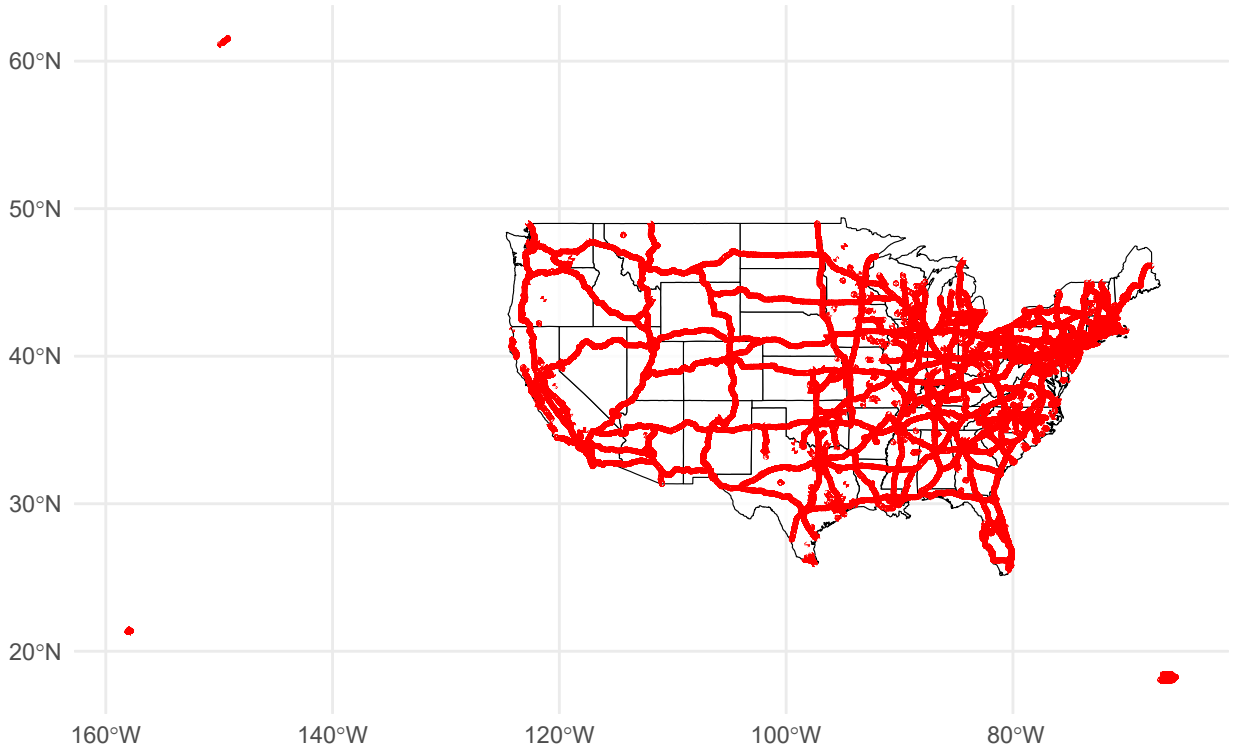
For example, we can calculate the intersection of different spatial objects using `sf::st_intersection()`. For this next example, I'm going to use two new spatial objects: 1) A state map of the USA from the **maps** package and 2) the primary roads from the **tigris** package. Don't worry too much about the process used for loading these datasets; I'll cover that in more depth shortly. For the moment, just focus on the idea that we want to see which adminstrative regions are intersected by the river network. Start by plotting all of the data to get a visual sense of the overlap:

```
## Get the data
usa = st_as_sf(map('state', plot = FALSE, fill = TRUE))
road = tigris::primary_roads()
```

Make sure they have the same projection

road = st_transform(road, crs = st_crs(usa))

```
ggplot() +
  geom_sf(data = usa, alpha = 0.8, fill = "white", col = "black") +
  geom_sf(data = road, col = "red", lwd = 1) +
  labs(
    title = "States of the US",
    subtitle = "Also showing the US road network"
    )
```



Uh oh, it looks like we don't have Alaska in our USA map and yet the roads went up there anyway. Also, why don't we just look at Maine?

```
# Turn off spherical geometry see: https://r-spatial.org/r/2020/06/17/s2.html
sf_use_s2(FALSE)
road = st_transform(road, crs = st_crs(me))
me_intersected = st_intersection(road, me)
me_intersected
```

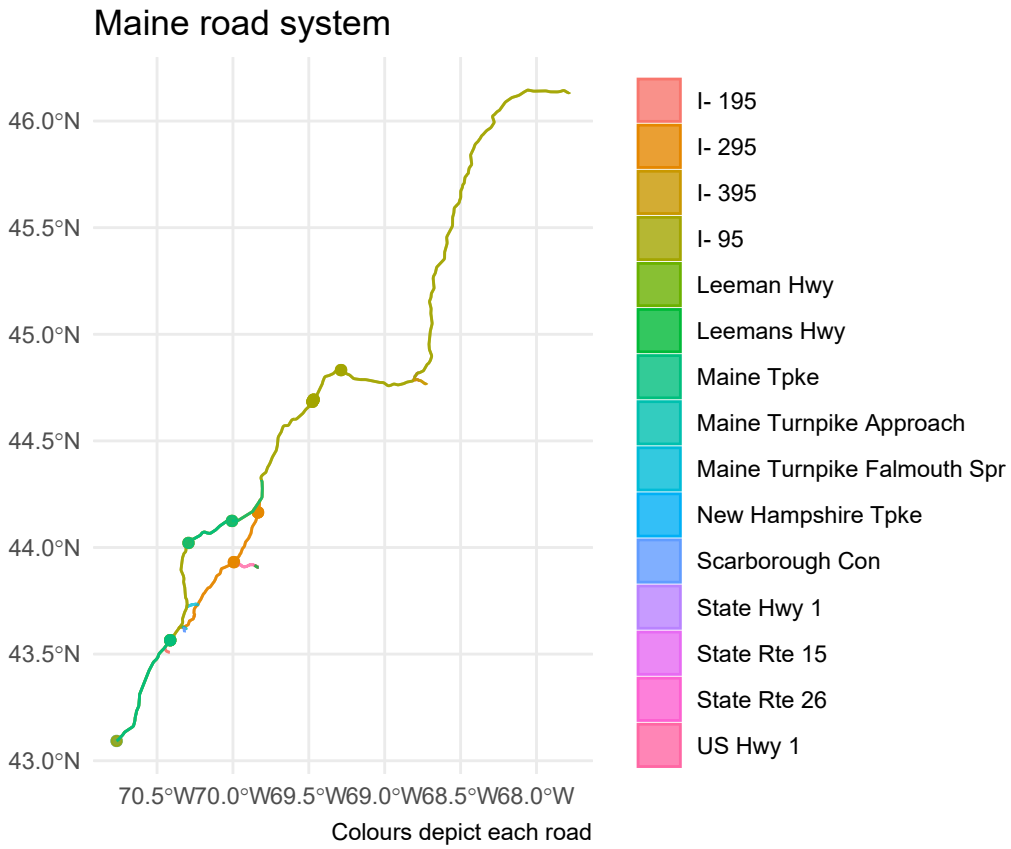
```
## Simple feature collection with 103 features and 23 fields
## Geometry type: GEOMETRY
## Dimension:     XY
## Bounding box:  xmin: -70.76643 ymin: 43.09272 xmax: -67.78126 ymax: 46.14542
## Geodetic CRS:  NAD83
## First 10 features:
##            LINEARID     FULLNAME RTTYP MTFCC STATEFP10 COUNTYFP10 COUNTYNS10
## 131   1105598252807 State Rte 15     S S1100        23        019   00581295
## 136   1104470316493 State Rte 15     S S1100        23        019   00581295
## 11246  110170156233        I- 95     I S1100        23        019   00581295
## 11247  110170156234        I- 95     I S1100        23        019   00581295
## 11298  110184905347        I- 95     I S1100        23        019   00581295
## 11421 1104470496492        I- 95     I S1100        23        019   00581295
## 11425 1104470495048        I- 95     I S1100        23        019   00581295
## 11457  110469245634        I- 95     I S1100        23        019   00581295
## 12595  110184905876       I- 395     I S1100        23        019   00581295
## 12614 1104469410451       I- 395     I S1100        23        019   00581295
##       GEOID10    NAME10       NAMELSAD10 LSAD10 CLASSFP10 MTFCC10 CSAFP10
## 131     23019 Penobscot Penobscot County     06        H1   G4020    <NA>
```

```
## 136     23019 Penobscot Penobscot County     06        H1   G4020    <NA>
## 11246   23019 Penobscot Penobscot County     06        H1   G4020    <NA>
## 11247   23019 Penobscot Penobscot County     06        H1   G4020    <NA>
## 11298   23019 Penobscot Penobscot County     06        H1   G4020    <NA>
## 11421   23019 Penobscot Penobscot County     06        H1   G4020    <NA>
## 11425   23019 Penobscot Penobscot County     06        H1   G4020    <NA>
## 11457   23019 Penobscot Penobscot County     06        H1   G4020    <NA>
## 12595   23019 Penobscot Penobscot County     06        H1   G4020    <NA>
## 12614   23019 Penobscot Penobscot County     06        H1   G4020    <NA>
##       CBSAFP10 METDIVFP10 FUNCSTAT10    ALAND10  AWATER10  INTPTLAT10
## 131      12620       <NA>          A 8799125852 413670635 +45.3906022
## 136      12620       <NA>          A 8799125852 413670635 +45.3906022
## 11246    12620       <NA>          A 8799125852 413670635 +45.3906022
## 11247    12620       <NA>          A 8799125852 413670635 +45.3906022
## 11298    12620       <NA>          A 8799125852 413670635 +45.3906022
## 11421    12620       <NA>          A 8799125852 413670635 +45.3906022
## 11425    12620       <NA>          A 8799125852 413670635 +45.3906022
## 11457    12620       <NA>          A 8799125852 413670635 +45.3906022
## 12595    12620       <NA>          A 8799125852 413670635 +45.3906022
## 12614    12620       <NA>          A 8799125852 413670635 +45.3906022
##         INTPTLON10 COUNTYFP STATEFP                       geometry
## 131   -068.6574869      019      23 LINESTRING (-68.7755 44.819...
## 136   -068.6574869      019      23 LINESTRING (-68.77229 44.78...
## 11246 -068.6574869      019      23     POINT (-69.28763 44.83182)
## 11247 -068.6574869      019      23      POINT (-69.28772 44.8321)
## 11298 -068.6574869      019      23 LINESTRING (-68.42664 45.85...
## 11421 -068.6574869      019      23 LINESTRING (-69.28763 44.83...
## 11425 -068.6574869      019      23 LINESTRING (-68.42663 45.85...
## 11457 -068.6574869      019      23 LINESTRING (-68.42664 45.85...
## 12595 -068.6574869      019      23 LINESTRING (-68.81279 44.78...
## 12614 -068.6574869      019      23 LINESTRING (-68.7216 44.771...
```

I had to set `sf_use_s2()` to false because the `st_intersection()` function is not yet compatible with the new s2, which uses spherical geometry. (The curse of open source is things update.)

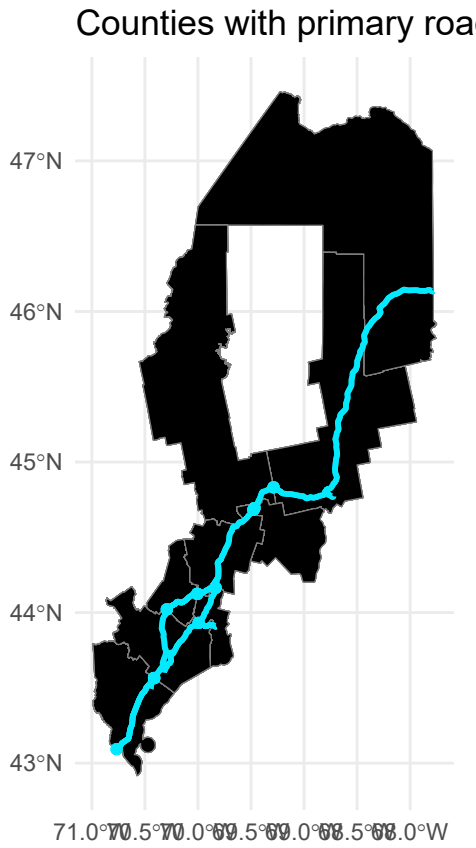
Note that `st_intersection()` only preserves *exact* points of overlap. As in, this is the exact path that the road follow within these regions. We can see this more explicitly in map form:

```
me_intersected %>%
  ggplot() +
  geom_sf(alpha = 0.8, aes(fill = FULLNAME, col = FULLNAME)) +
  labs(
    title = "Maine road system",
    caption = "Colours depict each road"
    ) +
  theme(legend.title = element_blank())
```



If we instead wanted to plot the intersected counties (i.e. keeping their full geometries), we have a couple options. We could filter the `me` object by matching its region IDs with the `me_intersected` object. However, a more direct option is to use the `sf::st_join()` function which matches objects based on overlapping (i.e. intersecting) geometries:

```
# Select only the road variables
st_join(me, road) %>%
  filter(!is.na(LINEARID)) %>% ## Get rid of regions with no overlap
  #distinct(GEOID10, .keep_all = T) %>% ## Some regions are duplicated b/c two branches of the road net
  ggplot() +
  geom_sf(alpha = 0.8, fill = "black", col = "gray50") +
  geom_sf(data = me_intersected, col = "#05E9FF", lwd = 1) +
  labs(title = "Counties with primary roads only")
```



One thing you'll often see in spatial analysis is the calculation of nearest neighbors. For example, we might want to know which county is closest to Cumberland. We can do this using the `sf::st_nearest_feature()` function. Note that this function returns the *index* of the nearest feature, not the feature itself. So we'll have to use that index to extract the relevant row from our data frame.

```
nearest_row <- st_nearest_feature(cumby, me %>% filter(NAME10!='Cumberland')) ## Returns the index of t
me %>% slice(nearest_row) # Get the nearest row
```

```
## Simple feature collection with 1 feature and 19 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -70.05182 ymin: 43.63647 xmax: -69.66474 ymax: 44.16802
## Geodetic CRS:  NAD83
##   STATEFP10 COUNTYFP10 COUNTYNS10 GEOID10    NAME10       NAMELSAD10 LSAD10
## 1        23        023   00581297   23023 Sagadahoc Sagadahoc County     06
##   CLASSFP10 MTFCC10 CSAFP10 CBSAFP10 METDIVFP10 FUNCSTAT10   ALAND10  AWATER10
## 1        H1   G4020     438    38860       <NA>          A 657066836 301332007
##    INTPTLAT10   INTPTLON10 COUNTYFP STATEFP                       geometry
## 1 +43.9166939 -069.8439936      023      23 MULTIPOLYGON (((-69.78507 4...
```

We can even do this for all of the rows, but it can get a little tedious with `sf::st_nearest_feature()`.

```
me %>%
  mutate(nearest_row = st_nearest_feature(.),
         nearest_county = NAME10[nearest_row])
```

```
## Simple feature collection with 16 features and 21 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
```

```
## Bounding box:  xmin: -71.08392 ymin: 42.91713 xmax: -66.88544 ymax: 47.45985
## Geodetic CRS:  NAD83
## First 10 features:
##      STATEFP10 COUNTYFP10 COUNTYNS10 GEOID10     NAME10        NAMELSAD10
## 45          23        019   00581295   23019  Penobscot  Penobscot County
## 231         23        029   00581300   23029 Washington Washington County
## 239         23        003   00581287   23003  Aroostook  Aroostook County
## 257         23        009   00581290   23009    Hancock    Hancock County
## 3009        23        007   00581289   23007   Franklin   Franklin County
## 3010        23        025   00581298   23025   Somerset   Somerset County
## 3011        23        017   00581294   23017     Oxford     Oxford County
## 3014        23        027   00581299   23027      Waldo      Waldo County
## 3015        23        015   00581293   23015    Lincoln    Lincoln County
## 3016        23        031   00581301   23031       York       York County
##      LSAD10 CLASSFP10 MTFCC10 CSAFP10 CBSAFP10 METDIVFP10 FUNCSTAT10
## 45       06        H1   G4020    <NA>    12620       <NA>          A
## 231      06        H1   G4020    <NA>     <NA>       <NA>          A
## 239      06        H1   G4020    <NA>     <NA>       <NA>          A
## 257      06        H1   G4020    <NA>     <NA>       <NA>          A
## 3009     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3010     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3011     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3014     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3015     06        H1   G4020    <NA>     <NA>       <NA>          A
## 3016     06        H1   G4020     438    38860       <NA>          A
##          ALAND10   AWATER10  INTPTLAT10   INTPTLON10
## 45    8799125852  413670635 +45.3906022 -068.6574869
## 231   6637257545 1800019787 +44.9670088 -067.6093542
## 239  17278664655  404653951 +46.7270567 -068.6494098
## 257   4110034060 1963321064 +44.5649063 -068.3707034
## 3009  4394196449  121392907 +44.9730124 -070.4447268
## 3010 10164156961  438038365 +45.5074824 -069.9760395
## 3011  5378990983  256086721 +44.4945850 -070.7346875
## 3014  1890479704  318149622 +44.5053607 -069.1396775
## 3015  1180563700  631400289 +43.9942645 -069.5140292
## 3016  2565935077  722608929 +43.4272386 -070.6704023
##                              geometry COUNTYFP STATEFP nearest_row nearest_county
## 45   MULTIPOLYGON (((-69.28127 4...      019      23           6       Somerset
## 231  MULTIPOLYGON (((-67.7543 45...      029      23           4        Hancock
## 239  MULTIPOLYGON (((-68.43227 4...      003      23           6       Somerset
## 257  MULTIPOLYGON (((-68.80096 4...      009      23          11           Knox
## 3009 MULTIPOLYGON (((-70.29383 4...      007      23          13   Androscoggin
## 3010 MULTIPOLYGON (((-69.85327 4...      025      23          14       Kennebec
## 3011 MULTIPOLYGON (((-71.05825 4...      017      23          10           York
## 3014 MULTIPOLYGON (((-69.26888 4...      027      23          14       Kennebec
## 3015 MULTIPOLYGON (((-69.69056 4...      015      23          14       Kennebec
## 3016 MULTIPOLYGON (((-70.76779 4...      031      23          15     Cumberland
```

Instead, I recommend using the `nngeo` package, which provides a more intuitive interface for nearest neighbor calculations. This makes it easy to find the k-nearest neighbors. `st_nn` takes an argument `k`, which is the number of nearest neighbors. `sf::st_join` allows you to specify that you want to join by `st_nn`, which means it will join by the nearest neighbor. You can even specify a max distance to look for neighbors with `maxdist`.

```
#library(nngeo) ## Already loaded
```

```
nngeo::st_nn(cumby, filter(me,NAME10!='Cumberland'), k = 2)
```

```
##   |                                                                      |
```

```
## [[1]]
## [1]  7 10
```

```
sf::st_join(cumby, filter(me,NAME10!='Cumberland'), join = st_nn, k = 5,maxdist=1)
```

```
##   |                                                                      |
```

```
## Simple feature collection with 4 features and 38 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -70.86662 ymin: 43.46688 xmax: -69.85703 ymax: 44.17104
## Geodetic CRS:  NAD83
##     STATEFP10.x COUNTYFP10.x COUNTYNS10.x GEOID10.x   NAME10.x
## 1            23          005     00581288     23005 Cumberland
## 1.1          23          005     00581288     23005 Cumberland
## 1.2          23          005     00581288     23005 Cumberland
## 1.3          23          005     00581288     23005 Cumberland
##          NAMELSAD10.x LSAD10.x CLASSFP10.x MTFCC10.x CSAFP10.x CBSAFP10.x
## 1   Cumberland County       06          H1     G4020       438      38860
## 1.1 Cumberland County       06          H1     G4020       438      38860
## 1.2 Cumberland County       06          H1     G4020       438      38860
## 1.3 Cumberland County       06          H1     G4020       438      38860
##     METDIVFP10.x FUNCSTAT10.x  ALAND10.x AWATER10.x INTPTLAT10.x INTPTLON10.x
## 1           <NA>            A 2163263369  989949911  +43.8083479 -070.3303753
## 1.1         <NA>            A 2163263369  989949911  +43.8083479 -070.3303753
## 1.2         <NA>            A 2163263369  989949911  +43.8083479 -070.3303753
## 1.3         <NA>            A 2163263369  989949911  +43.8083479 -070.3303753
##     COUNTYFP.x STATEFP.x STATEFP10.y COUNTYFP10.y COUNTYNS10.y GEOID10.y
## 1          005        23          23          017     00581294     23017
## 1.1        005        23          23          031     00581301     23031
## 1.2        005        23          23          023     00581297     23023
## 1.3        005        23          23          001     00581286     23001
##         NAME10.y        NAMELSAD10.y LSAD10.y CLASSFP10.y MTFCC10.y CSAFP10.y
## 1         Oxford       Oxford County       06          H1     G4020      <NA>
## 1.1         York         York County       06          H1     G4020       438
## 1.2    Sagadahoc    Sagadahoc County       06          H1     G4020       438
## 1.3 Androscoggin Androscoggin County       06          H1     G4020       438
##     CBSAFP10.y METDIVFP10.y FUNCSTAT10.y  ALAND10.y AWATER10.y INTPTLAT10.y
## 1         <NA>         <NA>            A 5378990983  256086721  +44.4945850
## 1.1      38860         <NA>            A 2565935077  722608929  +43.4272386
## 1.2      38860         <NA>            A  657066836  301332007  +43.9166939
## 1.3      30340         <NA>            A 1211926439   75610678  +44.1676811
##     INTPTLON10.y COUNTYFP.y STATEFP.y                       geometry
## 1   -070.7346875        017        23 MULTIPOLYGON (((-69.89595 4...
## 1.1 -070.6704023        031        23 MULTIPOLYGON (((-69.89595 4...
## 1.2 -069.8439936        023        23 MULTIPOLYGON (((-69.89595 4...
## 1.3 -070.2074347        001        23 MULTIPOLYGON (((-69.89595 4...
```

That's about as much **sf** functionality as I can show you for today. The remaining part of this lecture will cover some additional mapping considerations and some bonus spatial R "swag". However, I'll try to slip in a few more **sf**-specific operations along the way.

Aside: sf and data.table

sf objects are designed to integrate with a **tidyverse** workflow. They can also be made to work a **data.table** workflow too, but the integration is not as slick. This is a known issue and I'll only just highlight a few very brief considerations.

You can convert an **sf** object into a data.table. But note that the key geometry column appears to lose its attributes.

```
# library(data.table) ## Already loaded

me_dt = as.data.table(me)
head(me_dt)
```

```
##    STATEFP10 COUNTYFP10 COUNTYNS10 GEOID10     NAME10        NAMELSAD10 LSAD10
## 1:        23        019   00581295   23019  Penobscot  Penobscot County     06
## 2:        23        029   00581300   23029 Washington Washington County     06
## 3:        23        003   00581287   23003  Aroostook  Aroostook County     06
## 4:        23        009   00581290   23009    Hancock    Hancock County     06
## 5:        23        007   00581289   23007   Franklin   Franklin County     06
## 6:        23        025   00581298   23025   Somerset   Somerset County     06
##    CLASSFP10 MTFCC10 CSAFP10 CBSAFP10 METDIVFP10 FUNCSTAT10     ALAND10
## 1:        H1   G4020    <NA>    12620       <NA>          A  8799125852
## 2:        H1   G4020    <NA>     <NA>       <NA>          A  6637257545
## 3:        H1   G4020    <NA>     <NA>       <NA>          A 17278664655
## 4:        H1   G4020    <NA>     <NA>       <NA>          A  4110034060
## 5:        H1   G4020    <NA>     <NA>       <NA>          A  4394196449
## 6:        H1   G4020    <NA>     <NA>       <NA>          A 10164156961
##      AWATER10  INTPTLAT10   INTPTLON10 geometry COUNTYFP STATEFP
## 1:  413670635 +45.3906022 -068.6574869  <XY[1]>      019      23
## 2: 1800019787 +44.9670088 -067.6093542  <XY[1]>      029      23
## 3:  404653951 +46.7270567 -068.6494098  <XY[1]>      003      23
## 4: 1963321064 +44.5649063 -068.3707034  <XY[1]>      009      23
## 5:  121392907 +44.9730124 -070.4447268  <XY[1]>      007      23
## 6:  438038365 +45.5074824 -069.9760395  <XY[1]>      025      23
```

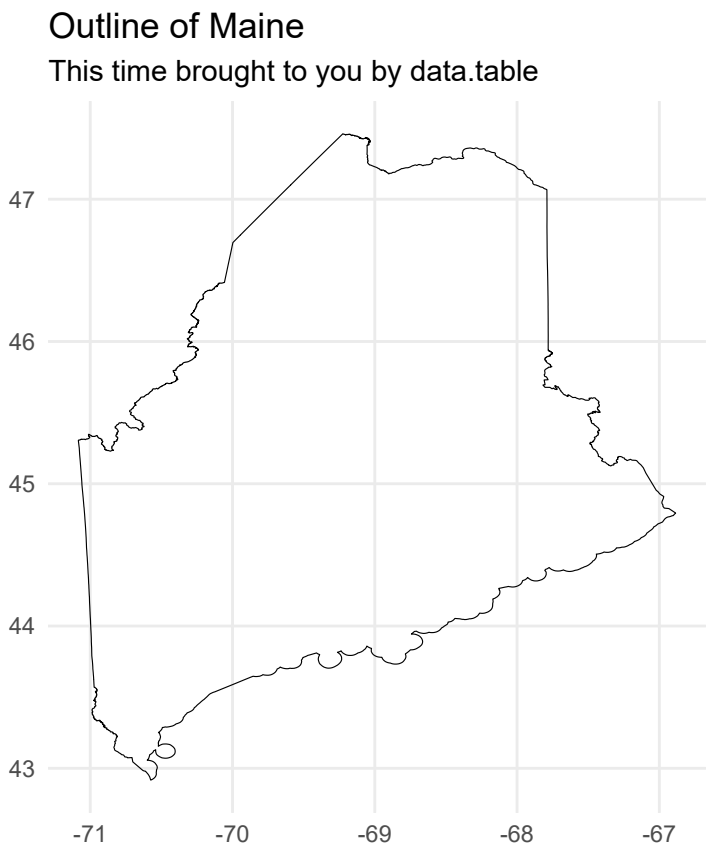
The good news is that all of this information is still there. It's just hidden from display.

```
me_dt$geometry
```

```
## Geometry set for 16 features
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -71.08392 ymin: 42.91713 xmax: -66.88544 ymax: 47.45985
## Geodetic CRS:  NAD83
## First 5 geometries:
## MULTIPOLYGON (((-69.28127 44.80866, -69.28143 4...
## MULTIPOLYGON (((-67.7543 45.66757, -67.75186 45...
## MULTIPOLYGON (((-68.43227 46.03557, -68.43285 4...
## MULTIPOLYGON (((-68.80096 44.46203, -68.80007 4...
## MULTIPOLYGON (((-70.29383 45.1099, -70.29348 45...
```

What's the upshot? Well, basically it means that you have to refer to this "geometry" column explicitly whenever you implement a spatial operation. For example, here's a repeat of the `st_union()` operation that we saw earlier. Note that I explicitly refer to the "geometry" column both for the `st_union()` operation (which, moreover, takes place in the "j" data.table slot) and when assigning the aesthetics for the `ggplot()` call.

```
me_dt[, .(geometry = st_union(geometry))] %>% ## Explicitly refer to 'geometry' col
    ggplot(aes(geometry = geometry)) +        ## And here again for the aes()
    geom_sf(fill=NA, col="black") +
    labs(title = "Outline of Maine",
         subtitle = "This time brought to you by data.table")
```



Of course, it's also possible to efficiently convert between the two classes — e.g. with `as.data.table()` and `st_as_sf()` — depending on what a particular section of code does (data wrangling or spatial operation). I find that often use this approach in my own work.

Bonus 1: Where to get map data

As our first Maine examples demonstrate, you can easily import external shapefiles, KML files, etc., into R. Just use the generic `sf::st_read()` function on any of these formats and the **sf** package will take care of the rest. However, we've also seen with the France example that you might not even need an external shapefile. Indeed, R provides access to a large number of base maps — e.g. countries of the world, US states and counties, etc. — through the **maps**, (higher resolution) **mapdata** and **spData** packages, as well as a whole ecosystem of more specialized GIS libraries.[8] To convert these maps into "**sf**-friendly" data frame format, we can use the `sf::st_as_sf()` function as per the below examples.

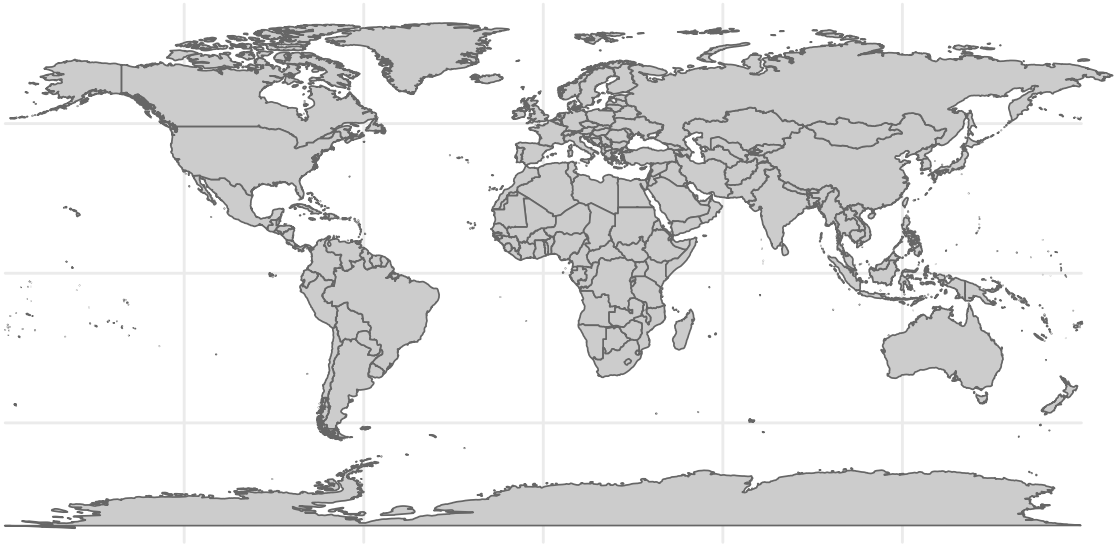
Example 1: The World

```
# library(maps) ## Already loaded

world = st_as_sf(map("world", plot = FALSE, fill = TRUE))
```

[8] The list of specialised maps packages is far too long for me to cover here. You can get marine regions, protected areas, nightlights, …, etc., etc.

```
world_map =
  ggplot(world) +
  geom_sf(fill = "grey80", col = "grey40", lwd = 0.3) +
  labs(
    title = "The world",
    subtitle = paste("EPSG:", st_crs(world)$epsg)
    )
world_map
```

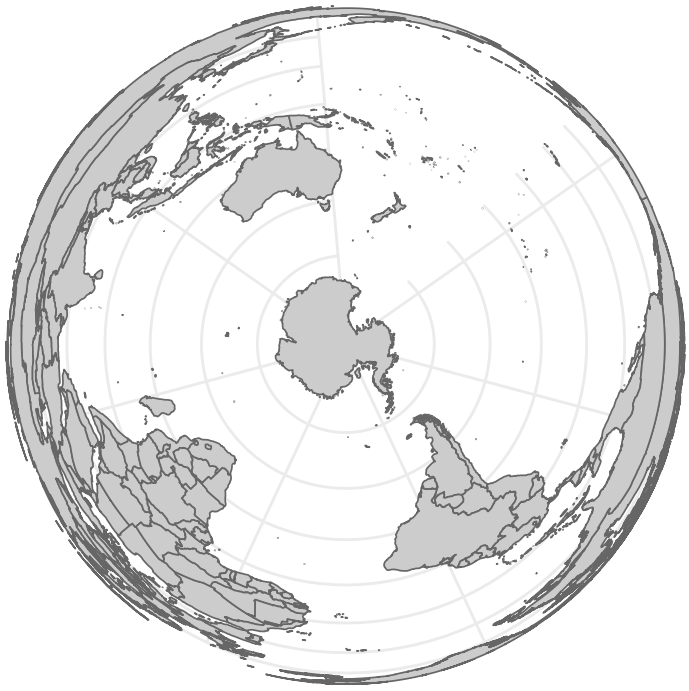


All of the usual **sf** functions and transformations can then be applied. For example, we can reproject the above world map onto the Lambert Azimuthal Equal Area projection (and further orientate it at the South Pole) as follows.

```
world_map +
  coord_sf(crs = "+proj=laea +y_0=0 +lon_0=155 +lat_0=-90") +
  labs(subtitle = "Lambert Azimuthal Equal Area projection")
```

The world

Lambert Azimuthal Equal Area projection



Several digressions on projection considerations

Winkel tripel projection As we've already seen, most map projections work great "out of the box" with **sf**. One niggling and notable exception is the Winkel tripel projection. This is the preferred global map projection of *National Geographic* and requires a bit more work to get it to play nicely with **sf** and **ggplot2** (as detailed in this thread). Here's a quick example of how to do it:

```
# library(lwgeom) ## Already loaded

wintr_proj = "+proj=wintri +datum=WGS84 +no_defs +over"

world_wintri = lwgeom::st_transform_proj(world, crs = wintr_proj)

## Don't necessarily need a graticule, but if you do then define it manually:
gr =
  st_graticule(lat = c(-89.9,seq(-80,80,20),89.9)) %>%
  lwgeom::st_transform_proj(crs = wintr_proj)

ggplot(world_wintri) +
  geom_sf(data = gr, color = "#cccccc", size = 0.15) + ## Manual graticule
  geom_sf(fill = "grey80", col = "grey40", lwd = 0.3) +
  coord_sf(datum = NA) +
  theme_ipsum(grid = F) +
  labs(title = "The world", subtitle = "Winkel tripel projection")
```

The world

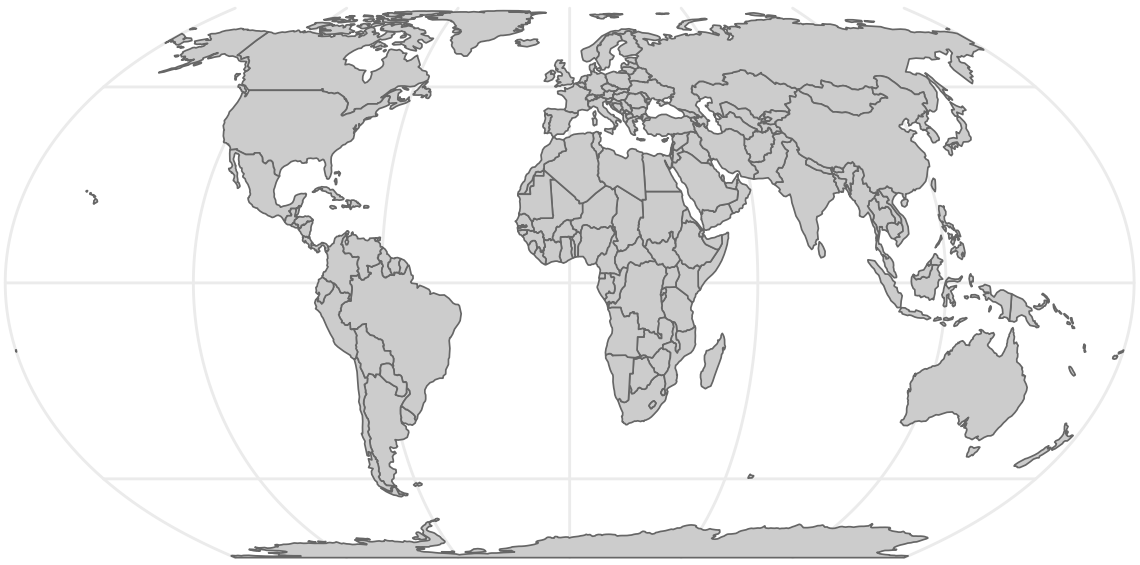
Winkel tripel projection

Equal Earth projection The latest and greatest projection, however, is the "Equal Earth" projection. This *does* work well out of the box, in part due to the `ne_countries` dataset that comes bundled with the **rnaturalearth** package (link). I'll explain that second part of the previous sentence in moment. But first let's see the Equal Earth projection in action.

```
# library(rnaturalearth) ## Already loaded

countries =
  ne_countries(returnclass = "sf") %>%
  st_transform(8857) ## Transform to equal earth projection
  # st_transform("+proj=eqearth +wktext") ## PROJ string alternative

ggplot(countries) +
  geom_sf(fill = "grey80", col = "grey40", lwd = 0.3) +
  labs(title = "The world", subtitle = "Equal Earth projection")
```

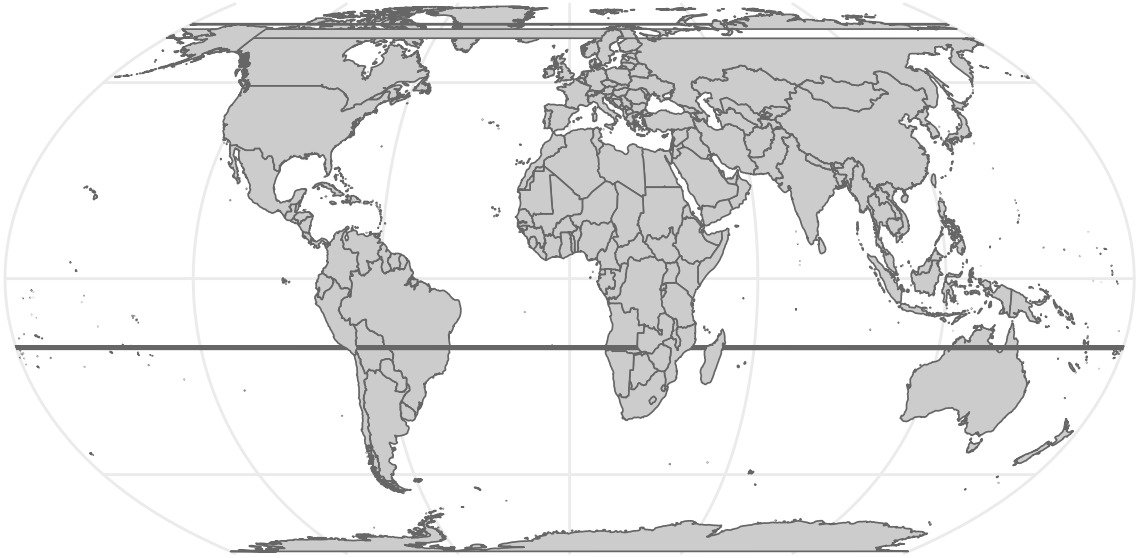


Pacfic-centered maps and other polygon mishaps As noted, the `rnaturalearth::ne_countries` spatial data frame is important for correctly displaying the Equal Earth projection. On the face of it, this looks pretty similar to our `maps::world` spatial data frame from earlier. They both contain polygons of all the countries in the world and appear to have similar default projections. However, some underlying nuances in how those polygons are constructed allows us avoid some undesirable visual artefacts that arise when reprojecting to the Equal Earth projection. Consider:

```
world %>%
  st_transform(8857) %>% ## Transform to equal earth projection
  ggplot() +
  geom_sf(fill = "grey80", col = "grey40", lwd = 0.3) +
  labs(title = "The... uh, world", subtitle = "Projection fail")
```

The... uh, world

Projection fail



These types of visual artefacts are particularly common for Pacific-centered maps and, in that case, arise from polygons extending over the Greenwich prime meridian. It's a suprisingly finicky problem to solve. Even the **rnaturalearth** doesn't do a good job. Luckily, Nate Miller has you covered with an excellent guide to set you on the right track.

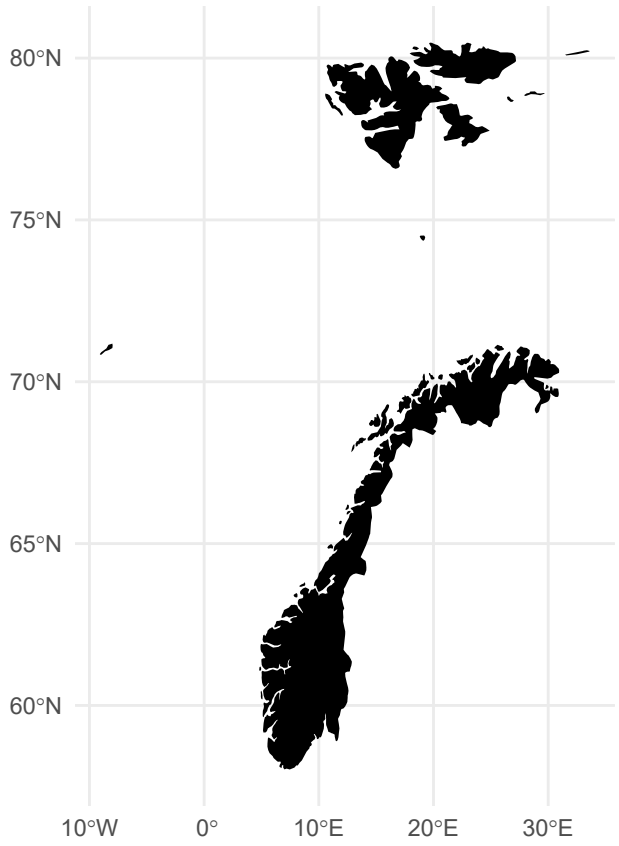
Example 2: A single country (i.e. Norway)

The **maps** and **mapdata** packages have detailed county- and province-level data for several individual nations. We've already seen this with France, but it includes the USA, New Zealand and several other nations. However, we can still use it to extract a specific country's border using some intuitive syntax. For example, we could plot a base map of Norway as follows.

```
norway = st_as_sf(map("world", "norway", plot = FALSE, fill = TRUE))

## For a hi-resolution map (if you *really* want to see all the fjords):
# norway = st_as_sf(map("worldHires", "norway", plot = FALSE, fill = TRUE))

norway %>%
  ggplot() +
  geom_sf(fill="black", col=NA)
```



Hmmm. Looks okay, but I don't really want to include non-mainland territories like Svalbaard (to the north) and the Faroe Islands (to the east). This gives me the chance to show off another handy function, `sf::st_crop()`, which I'll use to crop our **sf** object to a specific extent (i.e. rectangle). While I am at, we could also improve the projection. The Norwegian Mapping Authority recommends the ETRS89 / UTM projection, for which we can easily obtain the equivalent EPSG code (i.e. 25832) from this website.

```
norway %>%
  st_crop(c(xmin=0, xmax=35, ymin=0, ymax=72)) %>%
  st_transform(crs = 25832) %>%
  ggplot() +
  geom_sf(fill="black", col=NA)
```

There you go. A nice-looking map of Norway. Fairly appropriate that it resembles a gnarly black metal guitar.

> **Aside:** I recommend detaching the **maps** package once you're finished using it, since it avoids potential namespace conflicts with `purrr::map`.

```
detach(package:maps) ## To avoid potential purrr::map() conflicts
```

BONUS 2: More on US Census data with tidycensus and tigris

> **Note:** Before continuing with this section, you will first need to request an API key from the Census.

Working with Census data has traditionally quite a pain. You need to register on the website, then download data from various years or geographies separately, merge these individual files, etc. Thankfully, this too has recently become much easier thanks to the Census API and — for R at least — the **tidycensus** (link) and **tigris** (link) packages from Kyle Walker (a UO alum). This next section will closely follow a tutorial on his website.

We start by loading the packages and setting our Census API key. Note that I'm not actually running the below chunk, since I expect you to fill in your own Census key. You only have to run this function once.

```
# library(tidycensus) ## Already loaded
# library(tigris) ## Already loaded

## Replace the below with your own census API key. We'll use the "install = TRUE"
## option to save the key for future use, so we only ever have to run this once.
census_api_key("YOUR_CENSUS_API_KEY_HERE", install = TRUE)

## Also tell the tigris package to automatically cache its results to save on
## repeated downloading. I recommend adding this line to your ~/.Rprofile file
## so that caching is automatically enabled for future sessions. A quick way to
```

```
## do that is with the `usethis::edit_r_profile()` function.
options(tigris_use_cache=TRUE)
```

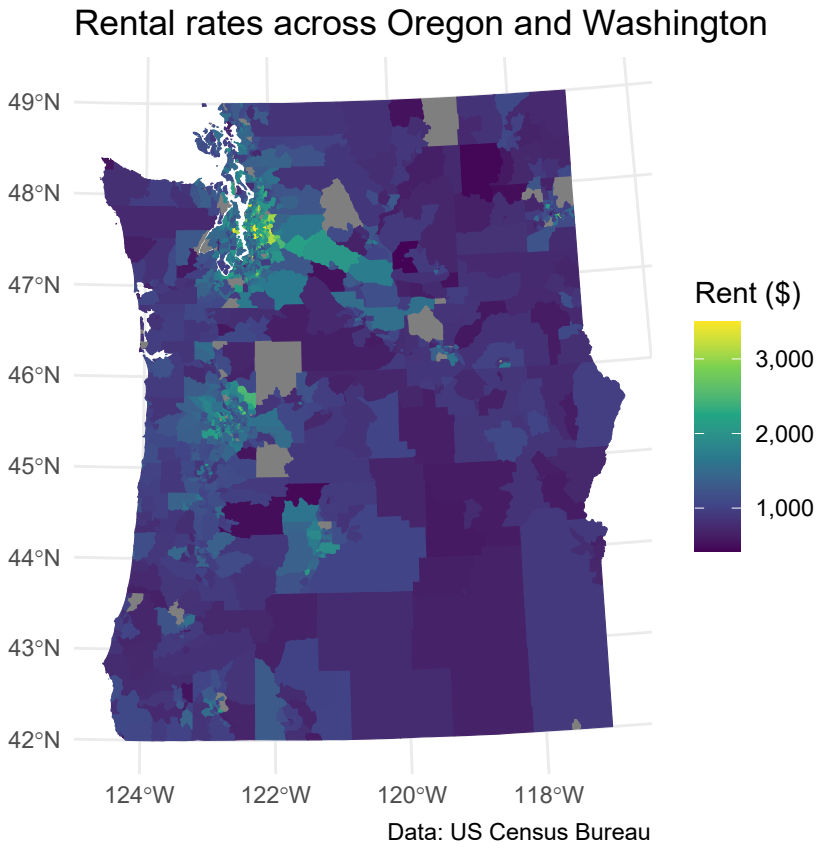
Let's say that our goal is to provide a snapshot of Census rental estimates across different cities in the Pacific Northwest. We start by downloading tract-level rental data for Oregon and Washington using the `tidycensus::get_acs()` function. Note that you'll need to look up the correct ID variable (in this case: "DP04_0134").

```
rent =
  tidycensus::get_acs(
    geography = "tract", variables = "DP04_0134",
    state = c("WA", "OR"), geometry = TRUE
    )
rent
```

```
## Simple feature collection with 2785 features and 5 fields (with 11 geometries empty)
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -124.7631 ymin: 41.99179 xmax: -116.4635 ymax: 49.00249
## Geodetic CRS:  NAD83
## # A tibble: 2,785 x 6
##    GEOID       NAME            variable estimate   moe                  geometry
##    <chr>       <chr>           <chr>       <dbl> <dbl>        <MULTIPOLYGON [°]>
##  1 53033010102 Census Tract 1~ DP04_01~     2182   676 (((-122.291 47.57247, -1~
##  2 53053062901 Census Tract 6~ DP04_01~     1296    72 (((-122.4834 47.21488, -~
##  3 53053072305 Census Tract 7~ DP04_01~     1215    61 (((-122.5264 47.22459, -~
##  4 53067990100 Census Tract 9~ DP04_01~       NA    NA (((-122.6984 47.10377, -~
##  5 53077001504 Census Tract 1~ DP04_01~      766   264 (((-120.501 46.60201, -1~
##  6 53053062400 Census Tract 6~ DP04_01~     1421   434 (((-122.442 47.22307, -1~
##  7 53063000700 Census Tract 7~ DP04_01~     1168    87 (((-117.454 47.71541, -1~
##  8 53033005803 Census Tract 5~ DP04_01~     1850   260 (((-122.393 47.64116, -1~
##  9 53053062802 Census Tract 6~ DP04_01~     1189   229 (((-122.5088 47.19201, -~
## 10 53063012701 Census Tract 1~ DP04_01~     1082    95 (((-117.2399 47.64983, -~
## # i 2,775 more rows
```

This returns an **sf** object, which we can plot directly.

```
rent %>%
  ggplot() +
  geom_sf(aes(fill = estimate, color = estimate)) +
  coord_sf(crs = 26910) +
  scale_fill_viridis_c(name = "Rent ($)", labels = scales::comma) +
  scale_color_viridis_c(name = "Rent ($)", labels = scales::comma) +
  labs(
    title = "Rental rates across Oregon and Washington",
    caption = "Data: US Census Bureau"
    )
```



Hmmm, looks like you want to avoid renting in Seattle if possible…

The above map provides rental information for pretty much all of the Pacific Northwest. Perhaps we're not interested in such a broad swatch of geography. What if we'd rather get a sense of rents within some smaller and well-defined metropolitan areas? Well, we'd need some detailed geographic data for starters, say from the TIGER/Line shapefiles collection. The good news is that the **tigris** package has you covered here. For example, let's say we want to narrow down our focus and compare rents across three Oregon metros: Portland (and surrounds), Corvallis, and Eugene.

```
or_metros =
  tigris::core_based_statistical_areas(cb = TRUE) %>%
  # filter(GEOID %in% c("21660", "18700", "38900")) %>% ## Could use GEOIDs directly if you know them
  filter(grepl("Portland|Corvallis|Eugene", NAME)) %>%
  filter(grepl("OR", NAME)) %>% ## Filter out Portland, ME
  select(metro_name = NAME)
```

Now we do a spatial join on our two data sets using the `sf::st_join()` function.

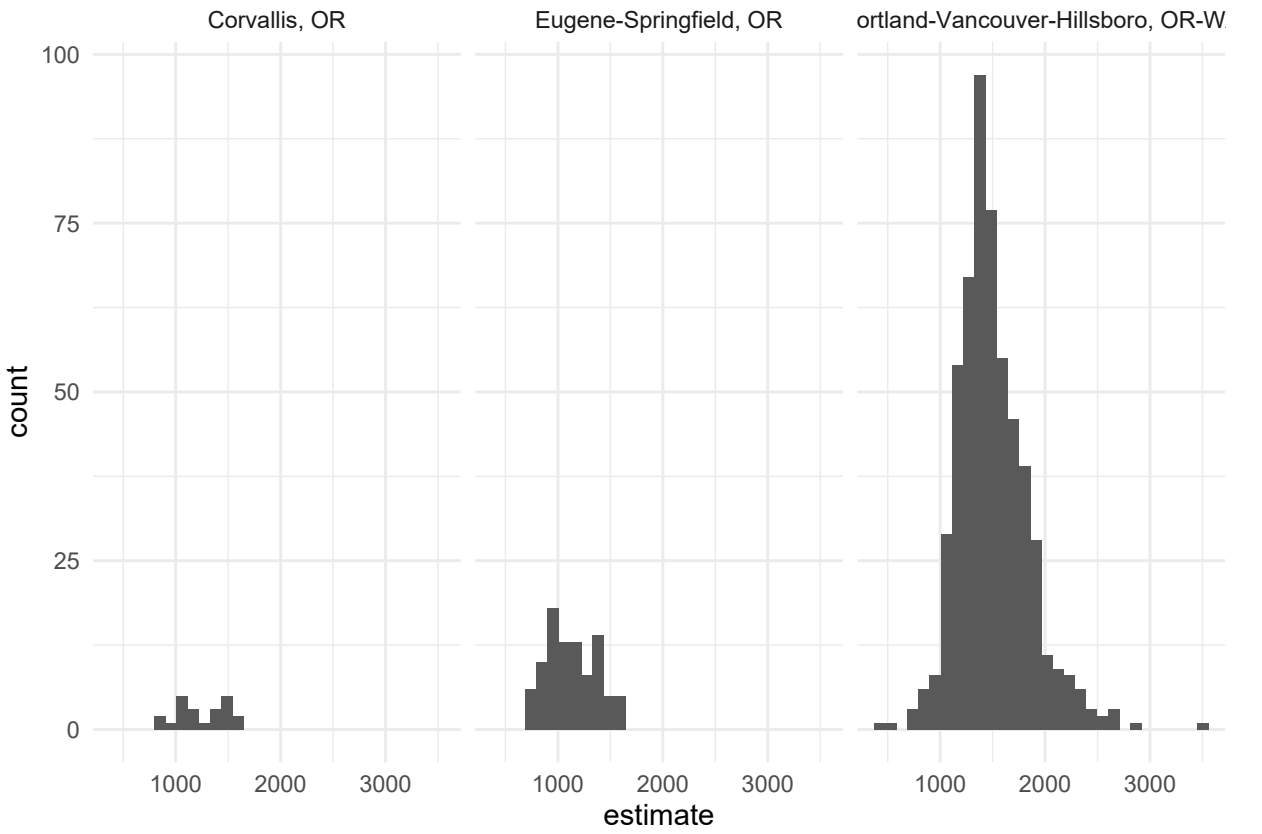
```
or_rent =
  st_join(
    rent,
    or_metros,
    join = st_within, left = FALSE
    )
or_rent
```

```
## Simple feature collection with 682 features and 6 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -124.1587 ymin: 43.4374 xmax: -121.5144 ymax: 46.38863
```

```
## Geodetic CRS:  NAD83
## # A tibble: 682 x 7
##    GEOID      NAME  variable estimate   moe                 geometry metro_name
##  * <chr>      <chr> <chr>       <dbl> <dbl>       <MULTIPOLYGON [°]> <chr>
##  1 530110407~ Cens~ DP04_01~     1370    45 (((-122.5528 45.7037, -1~ Portland-~
##  2 530110404~ Cens~ DP04_01~       NA    NA (((-122.641 45.72918, -1~ Portland-~
##  3 530110423~ Cens~ DP04_01~     1016   112 (((-122.6871 45.64037, -~ Portland-~
##  4 530110411~ Cens~ DP04_01~     1302   210 (((-122.5861 45.6788, -1~ Portland-~
##  5 530110413~ Cens~ DP04_01~     1564   189 (((-122.5277 45.6284, -1~ Portland-~
##  6 530110407~ Cens~ DP04_01~     1688    85 (((-122.5759 45.68595, -~ Portland-~
##  7 530110404~ Cens~ DP04_01~     1764    54 (((-122.6614 45.75089, -~ Portland-~
##  8 530110412~ Cens~ DP04_01~     1445   104 (((-122.5822 45.60845, -~ Portland-~
##  9 530110407~ Cens~ DP04_01~     1346   114 (((-122.5525 45.67782, -~ Portland-~
## 10 530110413~ Cens~ DP04_01~     1404    45 (((-122.5589 45.62102, -~ Portland-~
## # i 672 more rows
```

One useful way to summarize this data and compare across metros is with a histogram. Note that "regular" **ggplot2** geoms and functions play perfectly nicely with **sf** objects (i.e. we aren't limited to `geom_sf()`).

```
or_rent %>%
  ggplot(aes(x = estimate)) +
  geom_histogram() +
  facet_wrap(~metro_name)
```



That's a quick taste of working with **tidycensus** (and **tigris**). In truth, the package can do a lot more than I've shown you here. For example, you can also use it to download a variety of other Census microdata such as PUMS, which is much more detailed. See the **tidycensus** website for more information.

BONUS 3: Interactive maps

Now that you've grasped the basic properties of **sf** objects and how to plot them using **ggplot2**, its time to scale up with interactive maps.[9] You have several package options here. But I think the best are **leaflet** (link) and friends, or **plotly** (link). We've already covered the latter in a previous lecture, so I'll simply redirect interested parties to this link for some map-related examples. To expand on the former in more depth, leaflet.js is a lightweight JavaScript library for interactive mapping that has become extremely popular in recent years. You would have seen it being used across all the major news media publications (*New York Times*, *Washington Post*, *The Economist*, etc.). The good people at RStudio have kindly packaged a version of **leaflet** for R, which basically acts as a wrapper to the underlying JavaScript library.

The **leaflet** syntax is a little different to what you've seen thus far and I strongly encourage you to visit the package's excellent website for the full set of options. However, a key basic principle that it shares with **ggplot2** is that you *build your plot in layers*. Here's an example adapted from Julia Silge, which builds on the **tidycensus** package that we saw above. This time, our goal is to plot county-level population densities for Oregon as a whole and produce some helpful popup text if a user clicks on a particular county. First, we download the data using **tidycensus** and inspect the resulting data frame.

```
# library(tidycensus) ## Already loaded

oregon =
  get_acs(
    geography = "county", variables = "B01003_001",
    state = "OR", geometry = TRUE
    )
oregon
```

```
## Simple feature collection with 36 features and 5 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -124.5662 ymin: 41.99179 xmax: -116.4635 ymax: 46.29083
## Geodetic CRS:  NAD83
## First 10 features:
##    GEOID                     NAME   variable estimate moe
## 1  41035   Klamath County, Oregon B01003_001    68899  NA
## 2  41051 Multnomah County, Oregon B01003_001   810011  NA
## 3  41043      Linn County, Oregon B01003_001   127200  NA
## 4  41021   Gilliam County, Oregon B01003_001     1954  NA
## 5  41007   Clatsop County, Oregon B01003_001    40720  NA
## 6  41005 Clackamas County, Oregon B01003_001   418577  NA
## 7  41061     Union County, Oregon B01003_001    26255  NA
## 8  41003    Benton County, Oregon B01003_001    94667  NA
## 9  41033 Josephine County, Oregon B01003_001    87686  NA
## 10 41023     Grant County, Oregon B01003_001     7225  NA
##                          geometry
## 1  MULTIPOLYGON (((-122.29 42....
## 2  MULTIPOLYGON (((-122.9292 4...
## 3  MULTIPOLYGON (((-123.2608 4...
## 4  MULTIPOLYGON (((-120.6535 4...
## 5  MULTIPOLYGON (((-123.6647 4...
## 6  MULTIPOLYGON (((-122.8679 4...
## 7  MULTIPOLYGON (((-118.6978 4...
## 8  MULTIPOLYGON (((-123.8167 4...
## 9  MULTIPOLYGON (((-124.042 42...
```

[9] The ability to easily plot interactive maps from R is one of the main reasons that I switched all of my public presentations from PDFs to R Markdown-driven HMTL.

```
## 10 MULTIPOLYGON (((-119.6722 4...
```

So, the popup text of interest is held within the "NAME" and "estimate" columns. I'll use a bit of regular expression work to extract the county name from the "NAME" column (i.e. without the state) and then build up the map layer by layer. Note that the **leaflet** syntax requires that I prepend variables names with a tilde (~) when I refer to them in the plot building process. This tilde operates in much the same way as the asthetics (`aes()`) function does in **ggplot2**. One other thing to note is that I need to define a colour palette — which I'll call `col_pal` here — separately from the main plot. This is a bit of an inconvenience if you're used to the fully-integrated **ggplot2** API, but only a small one.

```
# library(leaflet) ## Already loaded

col_pal = colorQuantile(palette = "viridis", domain = oregon$estimate, n = 10)

oregon %>%
  mutate(county = gsub(",.*", "", NAME)) %>% ## Get rid of everything after the first comma
  st_transform(crs = 4326) %>%
  leaflet(width = "100%") %>%
  addProviderTiles(provider = "CartoDB.Positron") %>%
  addPolygons(
    popup = ~paste0(county, "<br>", "Population: ", prettyNum(estimate, big.mark=",")),
    stroke = FALSE,
    smoothFactor = 0,
    fillOpacity = 0.7,
    color = ~col_pal(estimate)
    ) %>%
  addLegend(
    "bottomright",
    pal = col_pal,
    values = ~estimate,
    title = "Population percentiles",
    opacity = 1
    )
```

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, pleas
```

No suprises here. The bulk of Oregon's population is situated just west of the Cascades, in cities connected along the I-5.

A particularly useful feature of interactive maps is not being limited by scale or granularity, so you can really dig in to specific areas and neighbourhoods. Here's an example using home values from Lane County.

```
lane =
  get_acs(
    geography = "tract", variables = "B25077_001",
    state = "OR", county = "Lane County", geometry = TRUE
    )
```

```
## Getting data from the 2017-2021 5-year ACS
```

```
## Downloading feature geometry from the Census website.  To cache shapefiles for use in future sessions
```

```
lane_pal = colorNumeric(palette = "plasma", domain = lane$estimate)

lane %>%
  mutate(tract = gsub(",.*", "", NAME)) %>% ## Get rid of everything after the first comma
  st_transform(crs = 4326) %>%
  leaflet(width = "100%") %>%
  addProviderTiles(provider = "CartoDB.Positron") %>%
  addPolygons(
```

```
    # popup = ~tract,
    popup = ~paste0(tract, "<br>", "Median value: $", prettyNum(estimate, big.mark=",")),
    stroke = FALSE,
    smoothFactor = 0,
    fillOpacity = 0.5,
    color = ~lane_pal(estimate)
    ) %>%
  addLegend(
    "bottomright",
    pal = lane_pal,
    values = ~estimate,
    title = "Median home values<br>Lane County, OR",
    labFormat = labelFormat(prefix = "$"),
    opacity = 1
    )
```

Having tried to convince you of the conceptual similarities between building maps with either **leaflet** or **ggplot2** — layering etc. — there's no denying that the syntax does take some getting used to. If you only plan to spin up the occasional interactive map, that cognitive overhead might be more effort than it's worth. The good news is that R spatial community has created the **mapview** package (link) for very quickly generating interactive maps. Behind the scenes, it uses **leaflet** to power everything, alongside some sensible defaults. Here's a quick example that recreates our Lane county home value map from above.

```
# library(mapview) ## Already loaded

mapview::mapview(lane, zcol = "estimate",
                 layer.name = 'Median home values<br>Lane County, OR')
```

Super easy, no? While it doesn't offer quite the same flexibility as the native **leaflet** syntax, **mapview** is a great way to get decent interactive maps up and running with minimal effort.[10]

BONUS 4: Other map plotting options (plot, tmap, etc.)

While I think that **ggplot2** (together with **sf**) and **leaflet** are the best value bet for map plotting in R — especially for relative newcomers that have been inculcated into the tidyverse ecosystem — it should be said that there are many other options available to you. The base R `plot()` function is *very* powerful and handles all manner of spatial objects. (It is also very fast.) The package that I'll highlight briefly in closing, however, is **tmap** (link). The focus of **tmap** is *thematic maps* that are "production ready". The package is extremely flexible and can accept various spatial objects and output various map types (including interactive). Moreover, the syntax should look very familar to us, since it is inspired by `ggplot2`'s layered graphics of grammar approach. Here's an example of a great looking map taken from the **tmap** homepage.

```
# library(tmap) ## Already loaded
# library(tmaptools) ## Already loaded

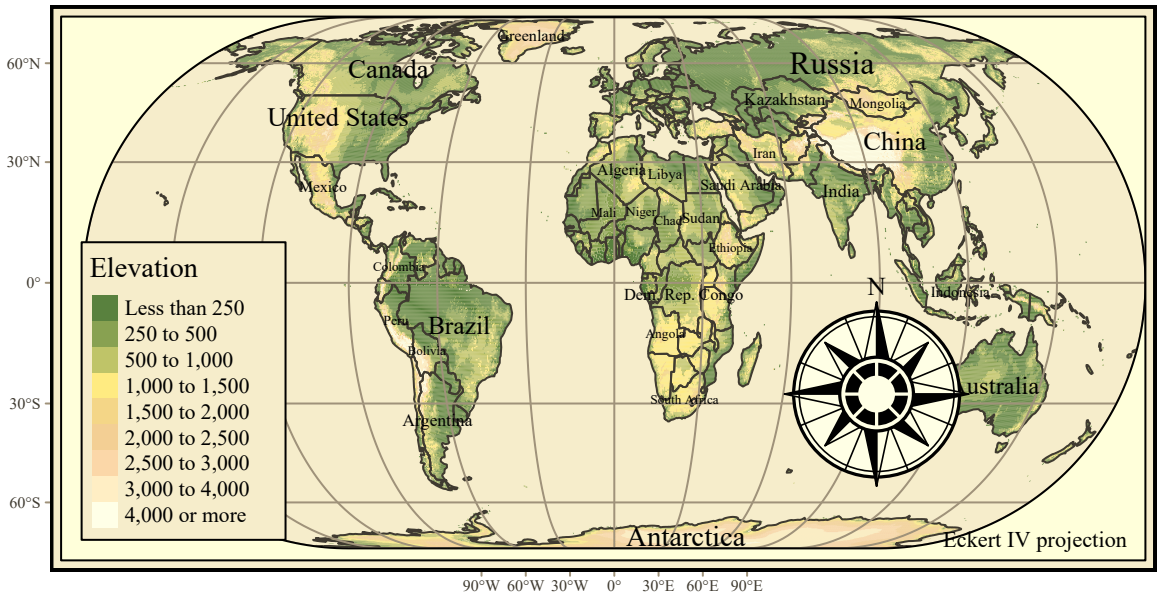
## Load elevation raster data, and country polygons
data(land, World)

## Convert to Eckert IV projection
land_eck4 = st_transform(land, "+proj=eck4")

## Plot
tm_shape(land_eck4) +
  tm_raster(
```

[10] I also want to flag the companion **mapedit** package (link), which lets you *edit* maps interactively by hand (e.g. redrawing polygons). I've used it for some of my projects and it works very well.

```
    "elevation",
    breaks=c(-Inf, 250, 500, 1000, 1500, 2000, 2500, 3000, 4000, Inf),
    palette = terrain.colors(9),
    title="Elevation"
    ) +
  tm_shape(World) +
  tm_borders("grey20") +
  tm_graticules(labels.size = .5) +
  tm_text("name", size="AREA") +
  tm_compass(position = c(.65, .15), color.light = "grey90") +
  tm_credits("Eckert IV projection", position = c("RIGHT", "BOTTOM")) +
  tm_style("classic") +
  tm_layout(
    inner.margins=c(.04,.03, .02, .01),
    legend.position = c("left", "bottom"),
    legend.frame = TRUE,
    bg.color="lightblue",
    legend.bg.color="lightblue",
    earth.boundary = TRUE,
    space.color="grey90"
    )
```



Further reading

You could easily spend a whole semester (or degree!) on spatial analysis and, more broadly, geocomputation. I've simply tried to give you as much useful information as can reasonably be contained in one lecture. Here are some resources for further reading and study:

- The package websites that I've linked to throughout this tutorial are an obvious next port of call for delving deeper into their functionality: **sf**, **leaflet**, etc.
- The best overall resource right now may be *Geocomputation with R*, a superb new text by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow. This is a "living", open-source document, which is constantly updated by its authors and features a very modern approach to working with geographic data. Highly recommended.
- Similarly, the rockstar team behind **sf**, Edzer Pebesma and Roger Bivand, are busy writing their own book, *Spatial Data Science*. This project is currently less developed, but I expect it to become the key reference point in years to

come. Imporantly, both of the above books cover **raster-based** spatial data.

- On the subject of raster data… If you're in the market for shorter guides, Jamie Afflerbach has a great introduction to rasters here. At a slightly more advanced level, UO's very own Ed Rubin has typically excellent tutorial here. Finally, the **sf** team is busy developing a new package called **stars**, which will provide equivalent functionality (among other things) for raster data. **UPDATE:** I ended up caving and wrote up a short set of bonus notes on rasters here.
- If you want more advice on drawing maps, including a bunch that we didn't cover today (choropleths, state-bins, etc.), Kieran Healy's *Data Vizualisation* book has you covered.
- Something else we didn't really cover at all today was **spatial statistics**. This too could be subject to a degree-length treatment. However, for now I'll simply point you to *Spatio-Temporal Statistics with R*, by Christopher Wikle and coauthors. (Another free book!) Finally, since it is likely the most interesting thing for economists working with spatial data, I'll also add that Darin Christensen and Thiemo Fetzer have written a very fast R-implementation (via C++) of Conley standard errors. The GitHub repo is here. See their original blog post (and update) for more details.