

Data Science for Economists

Lecture 9: Spatial analysis in R

Grant R. McDermott

University of Oregon | [EC 607](#)

Contents

Requirements	1
Introduction: CRS and map projections	2
Simple Features and the <code>sf</code> package	2
Where to get map data	13
BONUS 1: US Census data with <code>tidycensus</code> and <code>tigris</code>	19
BONUS 2: Interactive maps with <code>leaflet</code>	22
BONUS 3: Other map plotting options (<code>plot</code> , <code>tmap</code> , etc.)	25
Further reading	26

*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. I'll provide some links to further resources at the bottom of this document for those of you who want to explore further on your own.*

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`, `tmap`, `tmaptools`
- 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, tigris, tidy
```

```
## My preferred ggplot2 plotting theme (optional)
theme_set(hrbrthemes::theme_ipsum())
```

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)
```

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 this lecture. For the moment, however, just know that they are equally valid ways of specifying CRS in R (albeit 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 [simple features](#), 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

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

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 to 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 the North Carolina counties shapefile that comes bundled with **sf**. 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.

```
# library(sf) ## Already loaded

## Location of our shapefile (here: bundled together with the sf package)
file_loc = system.file("shape/nc.shp", package="sf")

## Read the shapefile into R
nc = st_read(file_loc, quiet = TRUE)
```

Simple Features as data frames

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

```
nc
```

```
## Simple feature collection with 100 features and 14 fields
## geometry type: MULTIPOLYGON
## dimension: XY
## bbox: xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
## geographic CRS: NAD27
## First 10 features:
##   AREA PERIMETER CNTY_ CNTY_ID      NAME FIPS FIPSNO CRESS_ID BIR74 SID74
## 1 0.114    1.442 1825    1825    Ashe 37009 37009      5 1091    1
## 2 0.061    1.231 1827    1827  Alleghany 37005 37005      3 487    0
## 3 0.143    1.630 1828    1828    Surry 37171 37171     86 3188    5
## 4 0.070    2.968 1831    1831 Currituck 37053 37053     27 508    1
## 5 0.153    2.206 1832    1832 Northampton 37131 37131     66 1421    9
## 6 0.097    1.670 1833    1833   Hertford 37091 37091     46 1452    7
## 7 0.062    1.547 1834    1834    Camden 37029 37029     15 286    0
## 8 0.091    1.284 1835    1835    Gates 37073 37073     37 420    0
## 9 0.118    1.421 1836    1836   Warren 37185 37185     93 968    4
## 10 0.124   1.428 1837    1837   Stokes 37169 37169     85 1612    1
##   NWBIR74 BIR79 SID79 NWBIR79          geometry
## 1       10 1364    0    19 MULTIPOLYGON (((-81.47276 3 ...
## 2       10  542    3    12 MULTIPOLYGON (((-81.23989 3 ...
## 3      208 3616    6    260 MULTIPOLYGON (((-80.45634 3 ...
## 4      123  830    2    145 MULTIPOLYGON (((-76.00897 3 ...
## 5     1066 1606    3   1197 MULTIPOLYGON (((-77.21767 3 ...
## 6      954 1838    5   1237 MULTIPOLYGON (((-76.74506 3 ...
## 7      115  350    2   139 MULTIPOLYGON (((-76.00897 3 ...
## 8      254  594    2   371 MULTIPOLYGON (((-76.56251 3 ...
```

²I 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](#).

```
## 9      748 1190      2      844 MULTIPOLYGON (((-78.30876 3 ...
## 10     160 2038      5      176 MULTIPOLYGON (((-80.02567 3 ...
```

Now we can see the explicit data frame structure that was I talking about earlier. 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.³ 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**

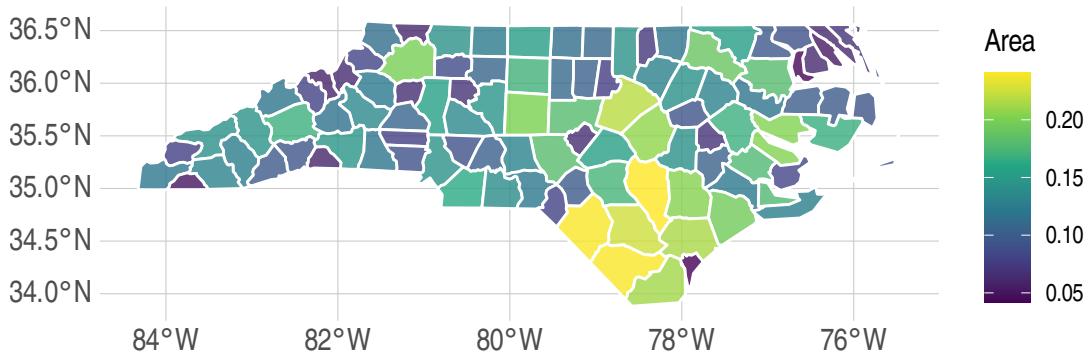
Plotting **sf** objects is incredibly easy thanks to the package's integration with both base R `plot()` and **ggplot2**. I'm going to focus on the latter here, but feel free to experiment.⁴ The key geom to remember is `geom_sf()`. For example:

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

nc_plot =
  ggplot(nc) +
  geom_sf(aes(fill = AREA), alpha=0.8, col="white") +
  scale_fill_viridis_c(name = "Area") +
  ggtitle("Counties of North Carolina")

nc_plot
```

Counties of North Carolina



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

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

```
## Simple feature collection with 2 features and 14 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:            xmin: -7160486 ymin: 4364317 xmax: -7077213 ymax: 4404770
## CRS:            +proj=moll
```

³For example, we could print out the coordinates needed to plot the first element in our data frame, Ashe county, by typing `nc$geometry[[1]]`. In contrast, I invite you to see how complicated the structure of a traditional spatial object is by running, say, `str(as(nc, "Spatial"))`.

⁴Plotting **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**.

```

##      AREA PERIMETER CNTY_ CNTY_ID      NAME  FIPS FIPSNO CRESS_ID BIR74 SID74
## 1 0.114     1.442  1825    1825      Ashe 37009 37009      5 1091     1
## 2 0.061     1.231  1827    1827  Alleghany 37005 37005      3 487     0
##   NWBIR74 BIR79 SID79 NWBIR79          geometry
## 1      10 1364     0      19 MULTIPOLYGON (((-7145980 43 ...
## 2      10  542     3      12 MULTIPOLYGON (((-7118089 43 ...

```

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.

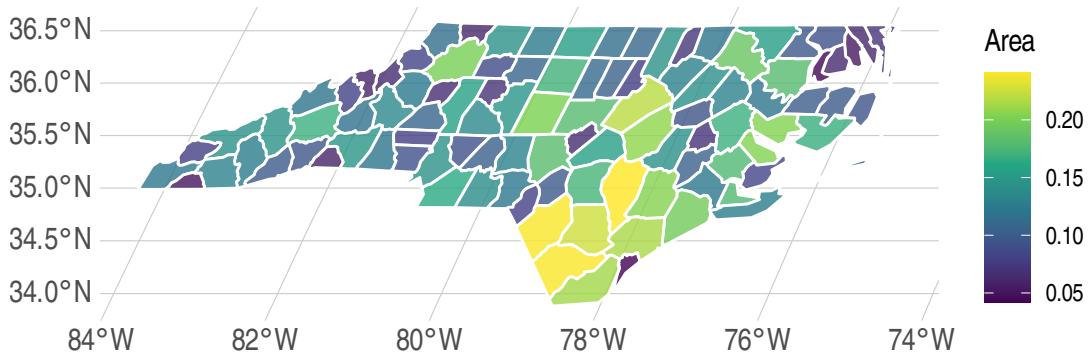
```

nc_plot +
  coord_sf(crs = "+proj=moll") +
  labs(subtitle = "Mollweide projection")

```

Counties of North Carolina

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 [NC state plane](#) projection.

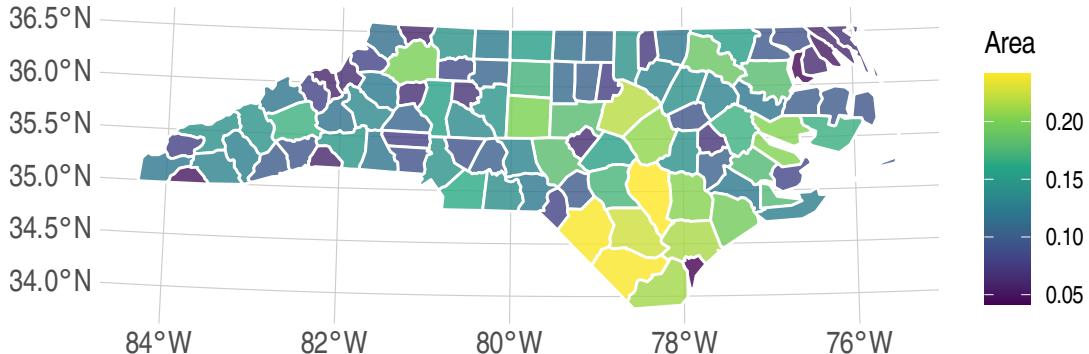
```

nc_plot +
  coord_sf(crs = 32119) +
  labs(subtitle = "NC state plane")

```

Counties of North Carolina

NC state plane



Data wrangling with dplyr and tidyverse

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:

```
nc %>%
  filter(NAME %in% c("Camden", "Durham", "Northampton")) %>%
  mutate(AREA_1000 = AREA*1000) %>%
  select(NAME, contains("AREA"), everything())

## Simple feature collection with 3 features and 15 fields
## geometry type: MULTIPOLYGON
## dimension: XY
## bbox: xmin: -79.01814 ymin: 35.85786 xmax: -75.95718 ymax: 36.55629
## geographic CRS: NAD27
##           NAME AREA AREA_1000 PERIMETER CNTY_ CNTY_ID FIPS FIPSNO CRESS_ID
## 1 Northampton 0.153      153    2.206  1832    1832 37131 37131       66
## 2 Camden 0.062        62    1.547  1834    1834 37029 37029       15
## 3 Durham 0.077       77    1.271  1908    1908 37063 37063       32
##   BIR74 SID74 NWBIR74 BIR79 SID79 NWBIR79                         geometry
## 1 1421     9    1066  1606     3    1197 MULTIPOLYGON (((-77.21767 3 ...
## 2 286      0    115   350     2    139 MULTIPOLYGON ((((-76.00897 3 ...
## 3 7970    16    3732 10432    22    4948 MULTIPOLYGON ((((-79.01814 3 ...
```

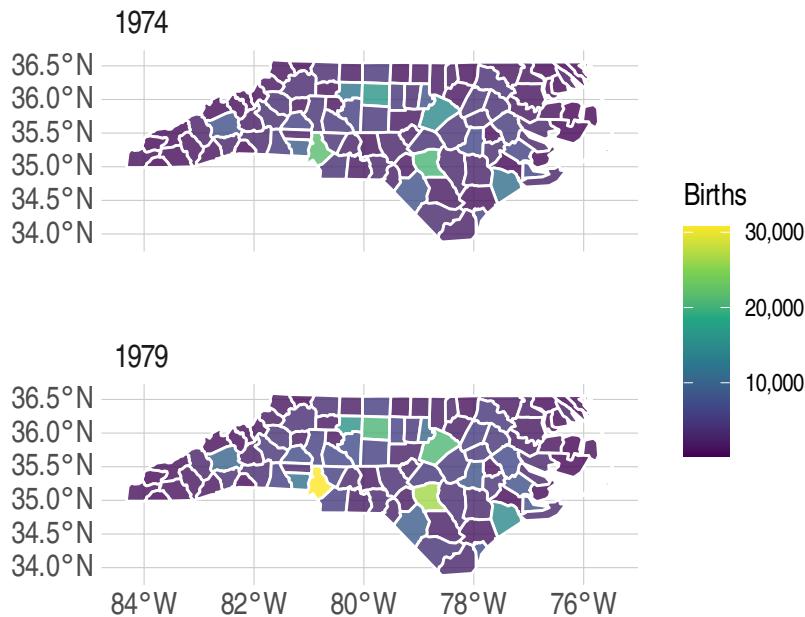
You can also perform `group_by()` and `summarise()` operations as per normal (see [here](#) for a nice example). Furthermore, the **dplyr** family of [join functions](#) also work, which can be especially handy when combining different datasets by (say) FIPS code or some other *attribute*. However, this 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.

And, just to show that we've got the bases covered, you can also implement your favourite **tidyverse** verbs. For example, we can `tidyverse::gather()` the data to long format, which is useful for faceted plotting.⁵ Here I demonstrate using the “BIR74” and “BIR79” columns (i.e. the number of births in each county in 1974 and 1979, respectively).

```
nc %>%
  select(county = NAME, BIR74, BIR79, -geometry) %>%
  gather(year, births, BIR74, BIR79) %>%
  mutate(year = gsub("BIR", "19", year)) %>%
  ggplot() +
  geom_sf(aes(fill = births), alpha=0.8, col="white") +
  scale_fill_viridis_c(name = "Births", labels = scales::comma) +
  facet_wrap(~year, ncol = 1) +
  labs(title = "Births by North Carolina county")
```

⁵In case you're wondering: the newer `tidyverse::pivot_*` functions [do not](#) yet work with **sf** objects.

Births by North Carolina county



Specialized geometric operations

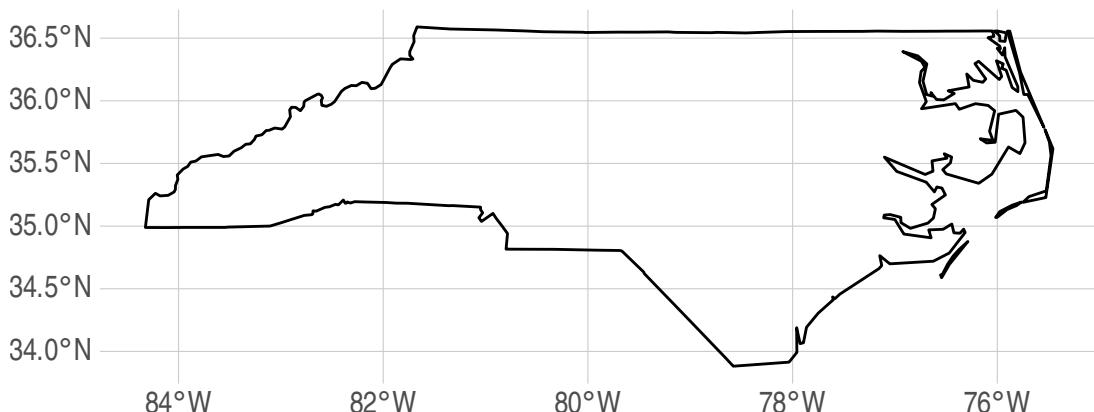
Alongside all the tidyverse functionality, the `sf` package comes with a full suite of geometrical operations. You should take a look at the [third sf vignette](#) or the [Geocomputation with R](#) book to get a complete overview. However, here are a few examples to get you started:

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()`:

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

## although coordinates are longitude/latitude, st_union assumes that they are planar
```

Outline of North Carolina



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

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

```
## Units: [m^2]
## [1] 1137388604 611077263 1423489919 694546292 1520740530
```

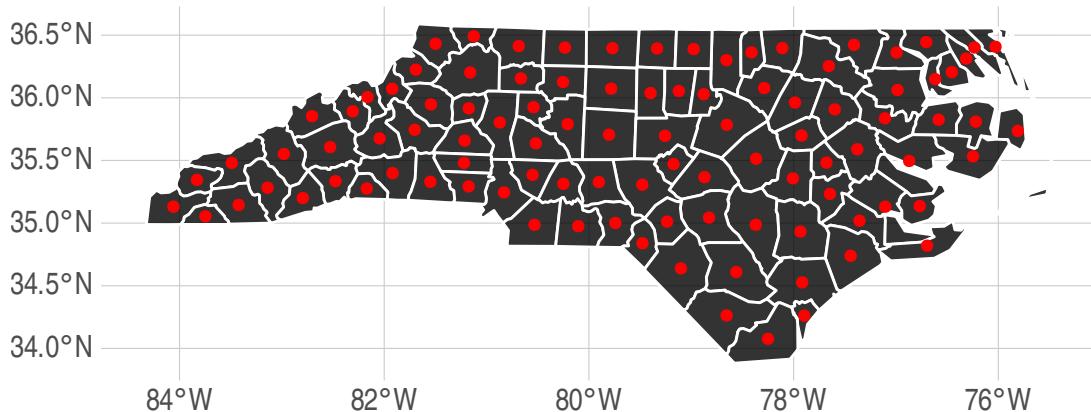
And:

```
nc_centroid = st_centroid(nc)

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

Counties of North Carolina

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.

```
ashe_brunswick = nc %>% filter(NAME %in% c("Ashe", "Brunswick"))
brunswick = nc %>% filter(NAME %in% c("Brunswick"))

## Use "by_element = TRUE" to give a vector instead of the default pairwise matrix
ab_dist = st_distance(ashe_brunswick, brunswick, by_element = TRUE)
# Units: [m]
# [1] 347930.7      0.0

## We can use the `units` package (already installed as sf dependency) to convert to kilometres
ab_dist = ab_dist %>% units::set_units(km) %>% round()
# Units: [km]
# [1] 348     0
```

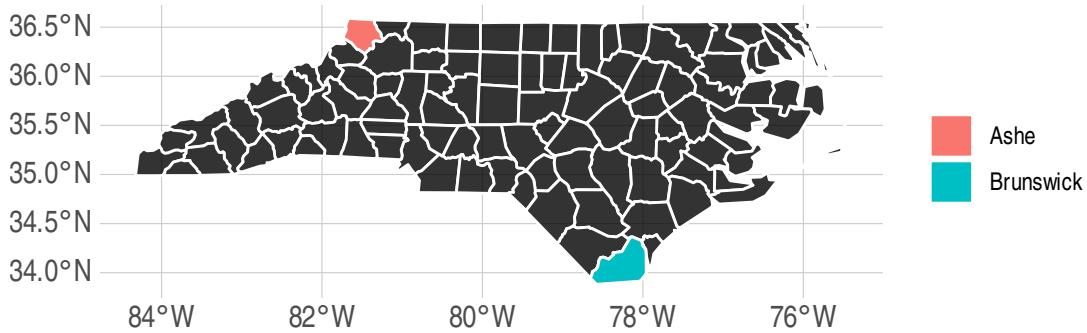
```

ggplot(nc) +
  geom_sf(fill = "black", alpha = 0.8, col = "white") +
  geom_sf(data = nc %>% filter(NAME %in% c("Ashe", "Brunswick")), aes(fill = NAME), col = "white") +
  labs(
    title = "Calculating distances",
    subtitle = paste0("The distance between Ashe and Brunswick is ", ab_dist[1], " km")
  ) +
  theme(legend.title = element_blank())

```

Calculating distances

The distance between Ashe and Brunswick is 348 km



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, 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 regional map of France from the `maps` package and 2) part of the Seine river network (including its Marne and Yonne tributaries) from the `spData` 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 administrative 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
france = st_as_sf(map('france', plot = FALSE, fill = TRUE))
data("seine", package = "spData")

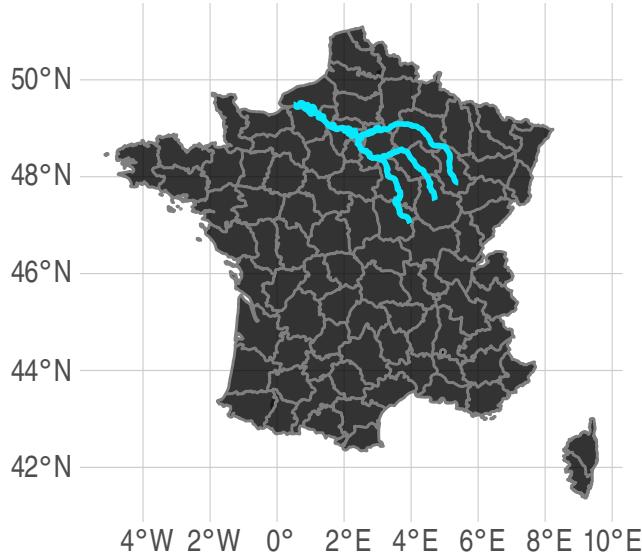
## Make sure they have the same projection
seine = st_transform(seine, crs = st_crs(france))

ggplot() +
  geom_sf(data = france, alpha = 0.8, fill = "black", col = "gray50") +
  geom_sf(data = seine, col = "#05E9FF", lwd = 1) +
  labs(
    title = "Administrative regions of France",
    subtitle = "Also showing the Seine, Marne and Yonne rivers"
  )

```

Administrative regions of France

Also showing the Seine, Marne and Yonne rivers



Now let's limit it to the intersected regions:

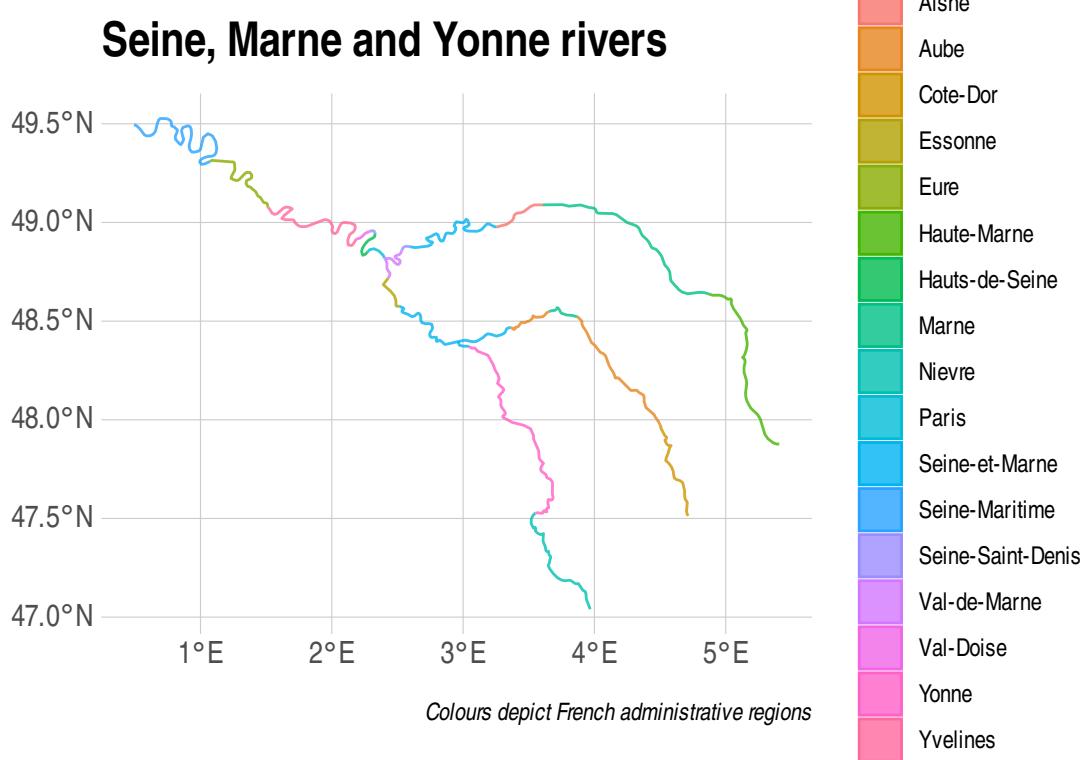
```
seine = st_transform(seine, crs = st_crs(france))
france_intersected = st_intersection(france, seine)
france_intersected

## Simple feature collection with 22 features and 2 fields
## geometry type:  GEOMETRY
## dimension:      XY
## bbox:            xmin: 0.4931747 ymin: 47.04007 xmax: 5.407725 ymax: 49.52717
## geographic CRS: WGS 84
## First 10 features:
##           ID   name          geom
## 6       Aisne Marne LINESTRING (3.608053 49.089 ...
## 14      Marne Marne LINESTRING (4.872966 48.637 ...
## 17  Seine-et-Marne Marne LINESTRING (3.254238 48.977 ...
## 19 Seine-Saint-Denis Marne LINESTRING (2.595133 48.876 ...
## 25    Val-de-Marne Marne LINESTRING (2.53346 48.8581 ...
## 30    Haute-Marne Marne LINESTRING (5.407725 47.877 ...
## 5     Seine-Maritime Seine LINESTRING (1.071621 49.309 ...
## 12      Eure Seine LINESTRING (1.514229 49.077 ...
## 14.1     Marne Seine LINESTRING (3.868337 48.522 ...
## 15    Val-Doise Seine LINESTRING (2.286567 48.954 ...
```

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

```
france_intersected %>%
  ggplot() +
  geom_sf(alpha = 0.8, aes(fill = ID, col = ID)) +
  labs(
    title = "Seine, Marne and Yonne rivers",
    caption = "Colours depict French administrative regions"
```

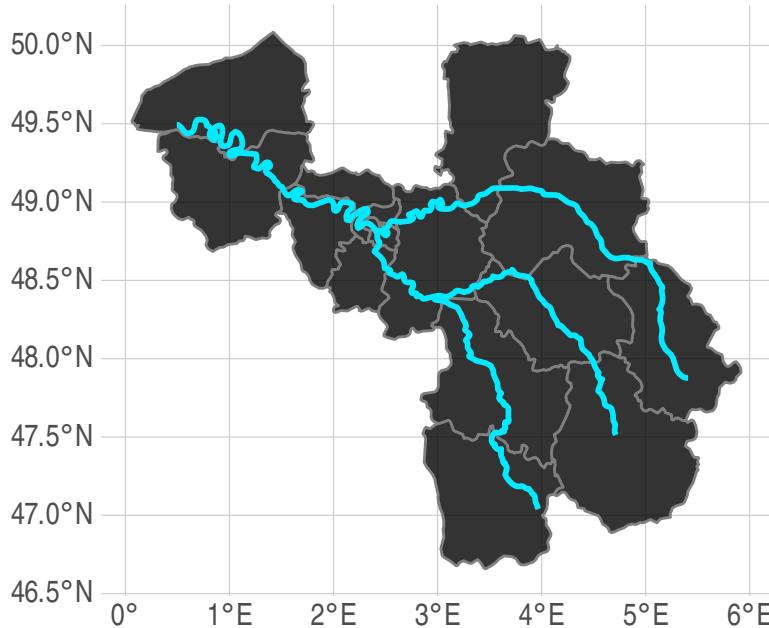
```
) +
theme(legend.title = element_blank())
```



If we instead wanted to plot the subsample of intersected provinces (i.e. keeping their full geometries), we have a couple options. We could filter the `france` object by matching its region IDs with the `france_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:

```
st_join(france, seine) %>%
  filter(!is.na(name)) %>% ## Get rid of regions with no overlap
  distinct(ID, .keep_all = T) %>% ## Some regions are duplicated b/c two branches of the river network flow thro
  ggplot() +
  geom_sf(alpha = 0.8, fill = "black", col = "gray50") +
  geom_sf(data = seine, col = "#05E9FF", lwd = 1) +
  labs(title = "Intersected regions only")
```

Intersected regions only



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

nc_dt = as.data.table(nc)
head(nc_dt)

##      AREA PERIMETER CNTY_ CNTY_ID      NAME FIPS FIPSNO CRESS_ID BIR74 SID74
## 1: 0.114     1.442 1825    1825    Ashe 37009 37009      5 1091     1
## 2: 0.061     1.231 1827    1827  Alleghany 37005 37005      3 487     0
## 3: 0.143     1.630 1828    1828   Surry 37171 37171     86 3188     5
## 4: 0.070     2.968 1831    1831 Currituck 37053 37053     27 508     1
## 5: 0.153     2.206 1832    1832 Northampton 37131 37131     66 1421     9
## 6: 0.097     1.670 1833    1833 Hertford 37091 37091     46 1452     7
##      NWBIR74 BIR79 SID79 NWBIR79 geometry
## 1:      10 1364     0     19 <XY[1]>
## 2:      10  542     3     12 <XY[1]>
## 3:     208 3616     6    260 <XY[1]>
## 4:     123  830     2    145 <XY[3]>
## 5:    1066 1606     3   1197 <XY[1]>
## 6:     954 1838     5   1237 <XY[1]>
```

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

```

nc_dt$geometry

## Geometry set for 100 features
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:            xmin: -84.32385 ymin: 33.88199 xmax: -75.45698 ymax: 36.58965
## geographic CRS: NAD27
## First 5 geometries:

## MULTIPOLYGON (((-81.47276 36.23436, -81.54084 3 ...
## MULTIPOLYGON (((-81.23989 36.36536, -81.24069 3 ...
## MULTIPOLYGON (((-80.45634 36.24256, -80.47639 3 ...
## MULTIPOLYGON (((-76.00897 36.3196, -76.01735 36 ...
## MULTIPOLYGON (((-77.21767 36.24098, -77.23461 3 ...

```

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.

```

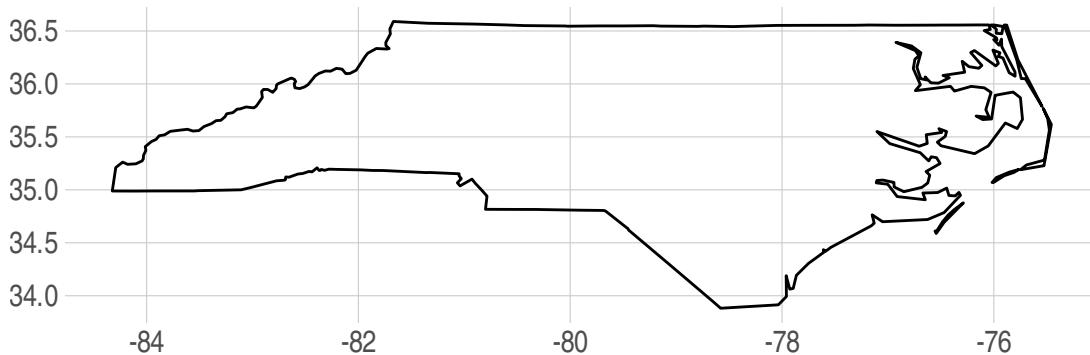
nc_dt[, .(geometry = st_union(geometry))] %>% ## Explicitly refer to 'geometry' col
  ggplot(aes(geometry = geometry)) +           ## And here again for the aes()
  geom_sf(fill=NA, col="black") +
  labs(title = "Outline of North Carolina",
       subtitle = "This time brought to by data.table")

## although coordinates are longitude/latitude, st_union assumes that they are planar

```

Outline of North Carolina

This time brought to 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.

Where to get map data

As our first North Carolina 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.⁶ 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

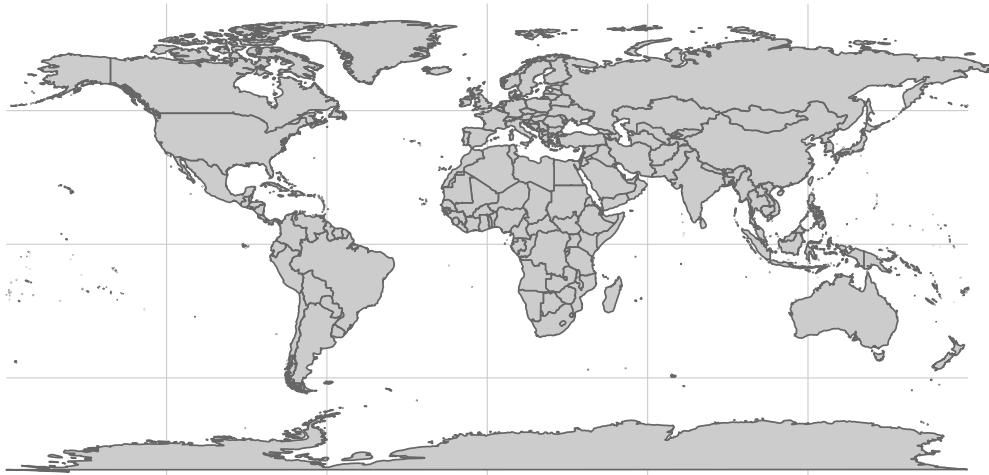
```
# library(maps) ## Already loaded

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

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
```

The world

EPSG: 4326



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")
```

⁶The list of specialised maps packages is far too long for me to cover here. You can get [marine regions](#), [protected areas](#), [nightlights](#), ..., etc., etc.

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"

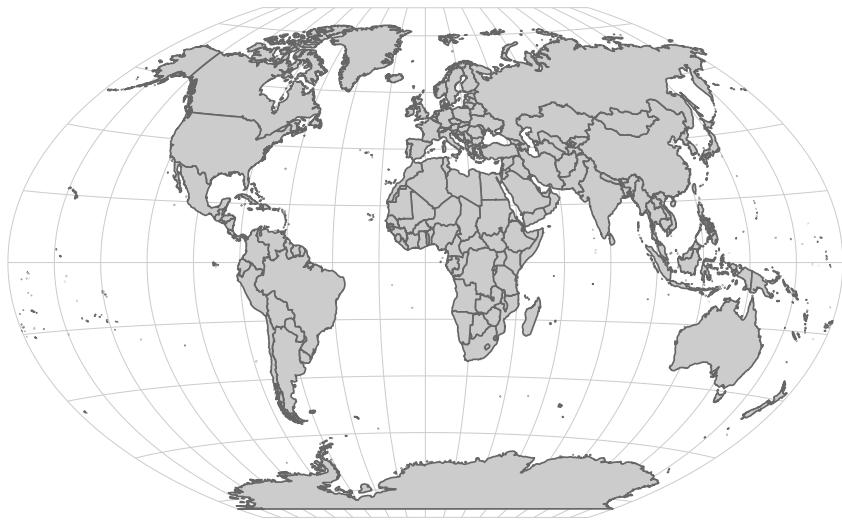
world_wintri = lwgeom::st_transform_proj(world, crs = wintr_proj)

## 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 **rnatu**re**earth** 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(rnatu
```

re

```
) ## Already loaded
```



```
countries =
```

```
  ne_countries(returnclass = "sf") %>%
```

```
  st_transform(8857) ## Transform to equal earth projection
```

```
  # st_transform("+proj=eqearth +wktext") ## PROJ string alternative
```



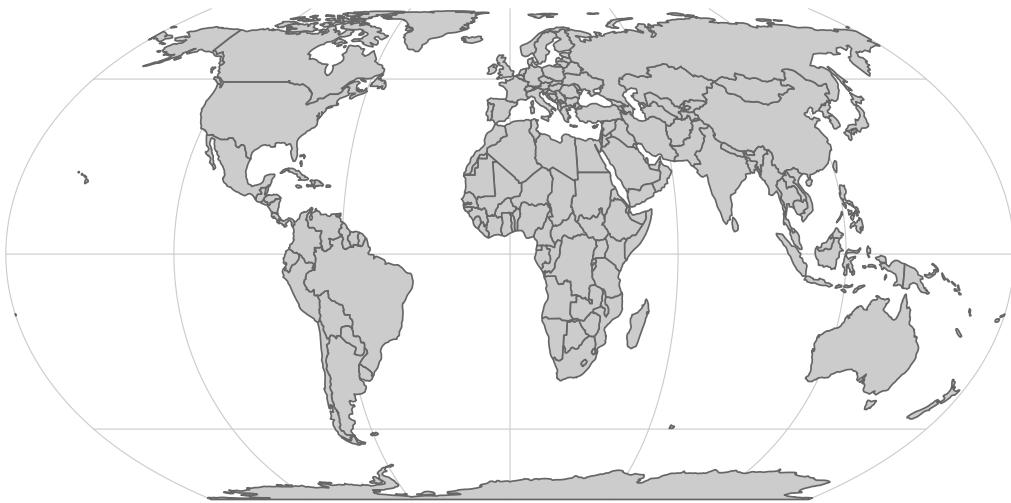
```
ggplot(countries) +
```

```
  geom_sf(fill = "grey80", col = "grey40", lwd = 0.3) +
```

```
  labs(title = "The world", subtitle = "Equal Earth projection")
```

The world

Equal Earth projection

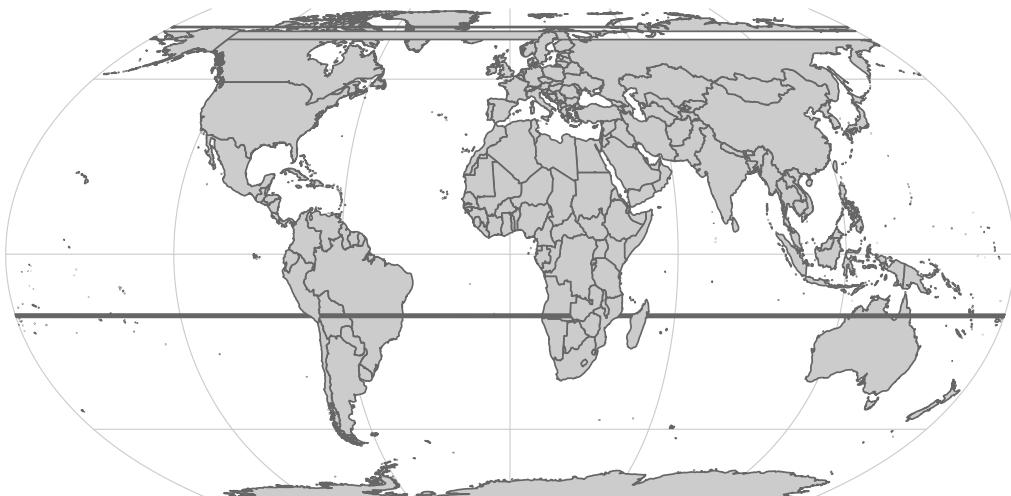


Pacific-centered maps and other polygon mishaps As noted, the `rnatrualearth::ne_countries` spatial data frame is important for correctly displaying the Equal Earth projection. On the face of it, this looks pretty similar to our `maps :: world` spatial data frame from earlier. They both contain polygons 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 surprisingly finicky problem to solve. Even the `rnatu`re`earth` doesn't do a good job. Luckily, Nate Miller has you covered with an [excellent guide](#) to set you on the right track.

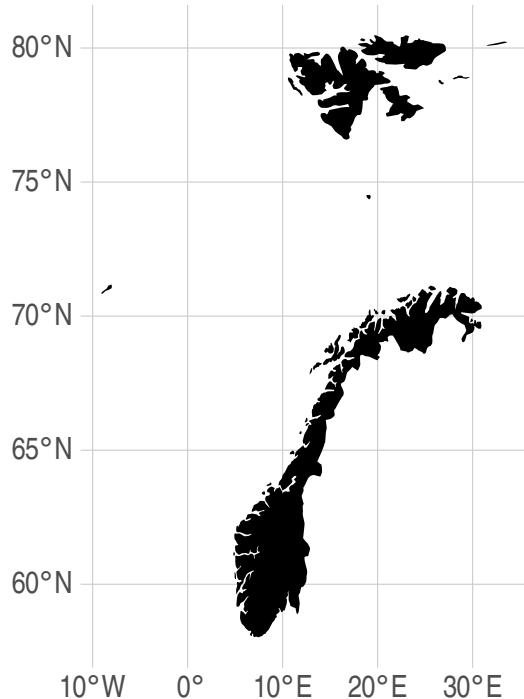
Example 2: A single country (i.e. Norway)

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 Svalbard (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 1: 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 grad). 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)
```

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 2292 features and 5 fields (with 10 geometries empty)
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -124.7631 ymin: 41.99179 xmax: -116.4635 ymax: 49.00249
## geographic CRS: NAD83
## # A tibble: 2,292 x 6
##   GEOID NAME     variable estimate    moe          geometry
##   <chr> <chr>     <chr>     <dbl> <dbl>  <MULTIPOLYGON [°]>
## 1 53061~ Census Trac~ DP04_01~    1181    168 (((-122.2311 48.00875, -122.2288~
## 2 53033~ Census Trac~ DP04_01~    1208     97 (((-122.3551 47.52103, -122.3551~
## 3 53077~ Census Trac~ DP04_01~    845    208 (((-120.9789 46.66908, -120.9792~
## 4 53033~ Census Trac~ DP04_01~   1727    112 (((-122.3555 47.65542, -122.3555~
## 5 53027~ Census Trac~ DP04_01~    727    147 (((-123.7275 47.07135, -123.7264~
## 6 53061~ Census Trac~ DP04_01~   2065    331 (((-122.1409 48.06446, -122.1402~
## 7 53033~ Census Trac~ DP04_01~   1333    142 (((-122.3551 47.50711, -122.3551~
## 8 53077~ Census Trac~ DP04_01~   1290    155 (((-120.5724 46.59989, -120.5619~
## 9 53011~ Census Trac~ DP04_01~   1272     62 (((-122.5589 45.62102, -122.5548~
## 10 53021~ Census Trac~ DP04_01~   550    189 (((-119.0936 46.28581, -119.0934~
## # ... with 2,282 more rows

```

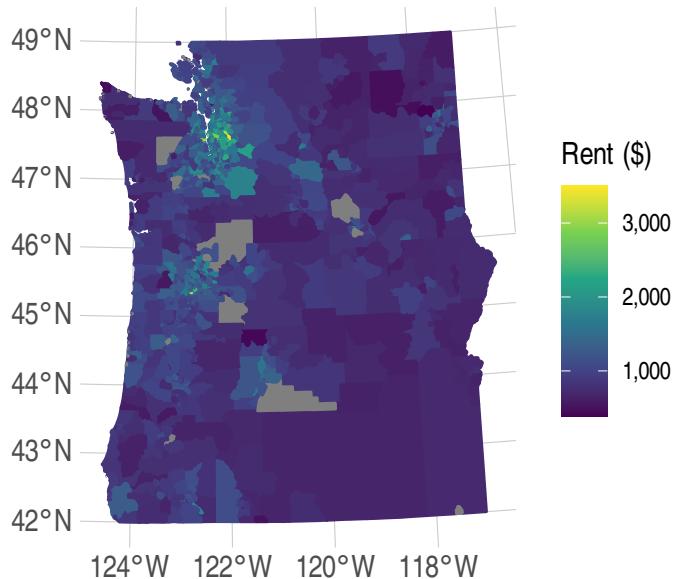
This returns an `sf` object, which we can plot directly.

```

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"
  )

```

Rental rates across Oregon and Washington



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 595 features and 6 fields  
## geometry type: MULTIPOLYGON  
## dimension: XY  
## bbox: xmin: -124.1587 ymin: 43.43739 xmax: -121.5144 ymax: 46.38863  
## geographic CRS: NAD83  
## # A tibble: 595 x 7  
##   GEOID NAME variable estimate    moe      geometry metro_name  
##   <chr>  <chr>  <chr>     <dbl>  <dbl>  <MULTIPOLYGON [°]> <chr>
```

```

## 1 53011~ Censu~ DP04_01~      1272    62 (((-122.5589 45.62102, -12~ Portland-V~
## 2 53011~ Censu~ DP04_01~      1185    112 (((-122.5528 45.69343, -12~ Portland-V~
## 3 53011~ Censu~ DP04_01~       866     64 (((-122.6143 45.63259, -12~ Portland-V~
## 4 53011~ Censu~ DP04_01~      1268     62 (((-122.5282 45.60814, -12~ Portland-V~
## 5 53011~ Censu~ DP04_01~      1268    267 (((-122.6002 45.78026, -12~ Portland-V~
## 6 53011~ Censu~ DP04_01~      1274    107 (((-122.5603 45.66535, -12~ Portland-V~
## 7 53011~ Censu~ DP04_01~      1092     77 (((-122.3562 45.57666, -12~ Portland-V~
## 8 53011~ Censu~ DP04_01~      1427    106 (((-122.6576 45.69122, -12~ Portland-V~
## 9 53011~ Censu~ DP04_01~      1148    132 (((-122.667 45.6664, -122.~ Portland-V~
## 10 53011~ Censu~ DP04_01~     1074     76 (((-122.6716 45.62722, -12~ Portland-V~
## # ... with 585 more rows

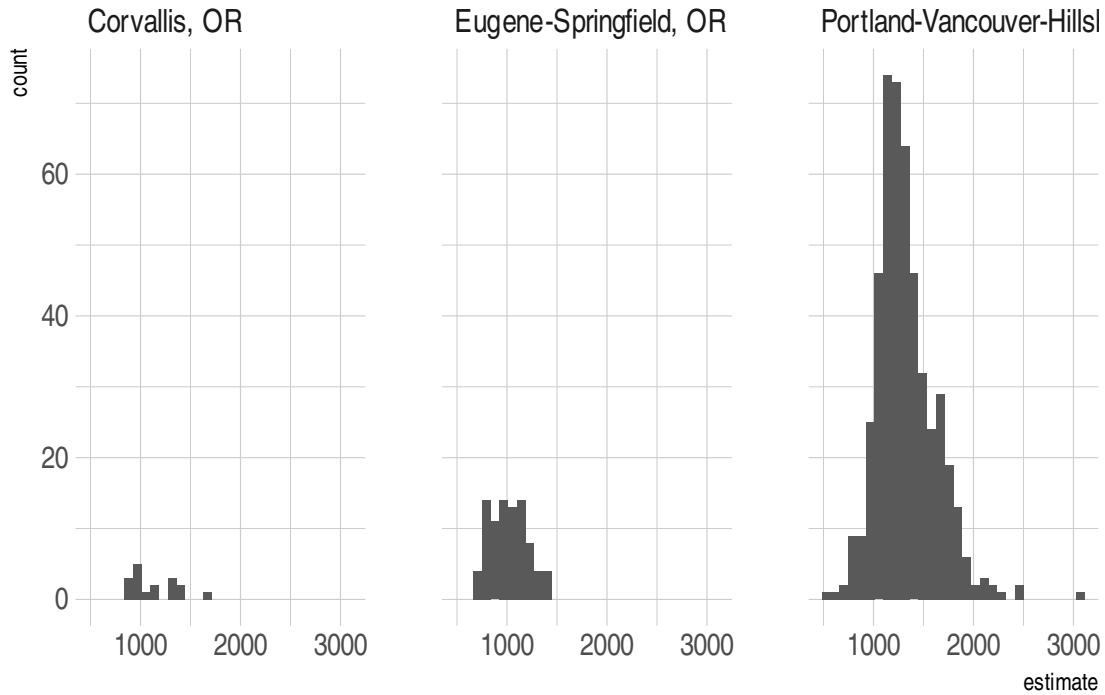
```

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 2: Interactive maps with leaflet

Now that you’ve grasped the basic properties of **sf** objects and how to plot them using **ggplot2**, it’s time to scale up with interactive maps.⁷ You have several package options here, but I think the best are **leaflet** (and derivatives) and **plotly**. We’ve already covered the latter in a previous lecture, so I’ll simply redirect interested parties to [this link](#) for some **sf**-specific examples. To expand on the former in more depth, **leaflet** is a lightweight JavaScript library for interactive mapping that has become extremely popular in recent years... and with good reason. The good people at RStudio have kindly packaged

⁷The 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 RMarkdown-driven HMTL.

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
## bbox: xmin: -124.5662 ymin: 41.99179 xmax: -116.4635 ymax: 46.29204
## geographic CRS: NAD83
## First 10 features:
##   GEOID             NAME variable estimate moe
## 1 41017 Deschutes County, Oregon B01003_001 186251 NA
## 2 41003 Benton County, Oregon B01003_001 91107 NA
## 3 41015 Curry County, Oregon B01003_001 22650 NA
## 4 41061 Union County, Oregon B01003_001 26337 NA
## 5 41055 Sherman County, Oregon B01003_001 1642 114
## 6 41051 Multnomah County, Oregon B01003_001 804606 NA
## 7 41007 Clatsop County, Oregon B01003_001 39102 NA
## 8 41033 Josephine County, Oregon B01003_001 86251 NA
## 9 41031 Jefferson County, Oregon B01003_001 23607 NA
## 10 41039 Lane County, Oregon B01003_001 373340 NA
##
##   geometry
## 1 MULTIPOLYGON (((-122.0019 4 ...
## 2 MULTIPOLYGON (((-123.8167 4 ...
## 3 MULTIPOLYGON (((-124.3239 4 ...
## 4 MULTIPOLYGON (((-118.6978 4 ...
## 5 MULTIPOLYGON (((-121.0312 4 ...
## 6 MULTIPOLYGON (((-122.9292 4 ...
## 7 MULTIPOLYGON (((-123.5989 4 ...
## 8 MULTIPOLYGON (((-124.042 42 ...
## 9 MULTIPOLYGON (((-121.8495 4 ...
## 10 MULTIPOLYGON (((-124.1503 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 aesthetics (`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.

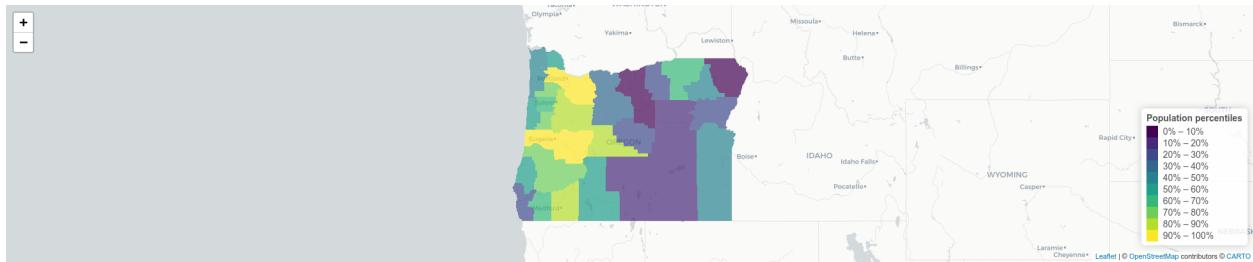
```
# library(leaflet) ## Already loaded
```

```

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
  )

```



No surprises 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 2015-2019 5-year ACS
lane_pal = colorNumeric(palette = "plasma", domain = lane$estimate)

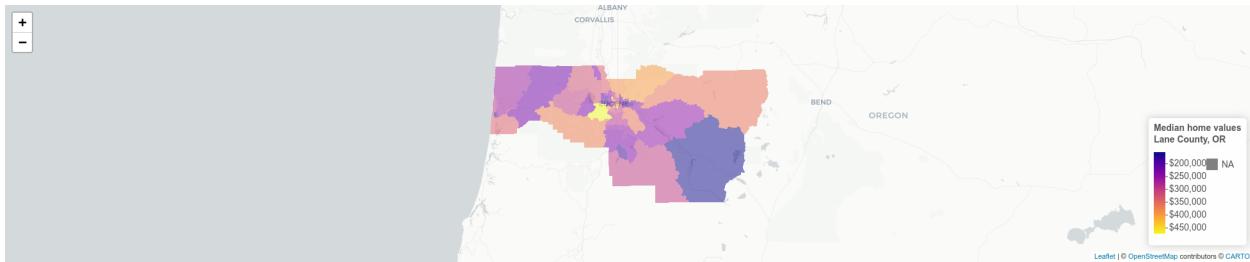
lane %>%
  mutate(tract = gsub(",.*", "", NAME)) %>% ## Get rid of everything after the first comma
  st_transform(crs = 4326) %>%
  leaflet(width = "100%") %>%
  addProviderTiles(provider = "CartoDB.Positron") %>%
  addPolygons(
    # popup = ~tract,
    popup = ~paste0(tract, "<br>", "Median value: $", 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
)

```



BONUS 3: 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**. 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 familiar 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")

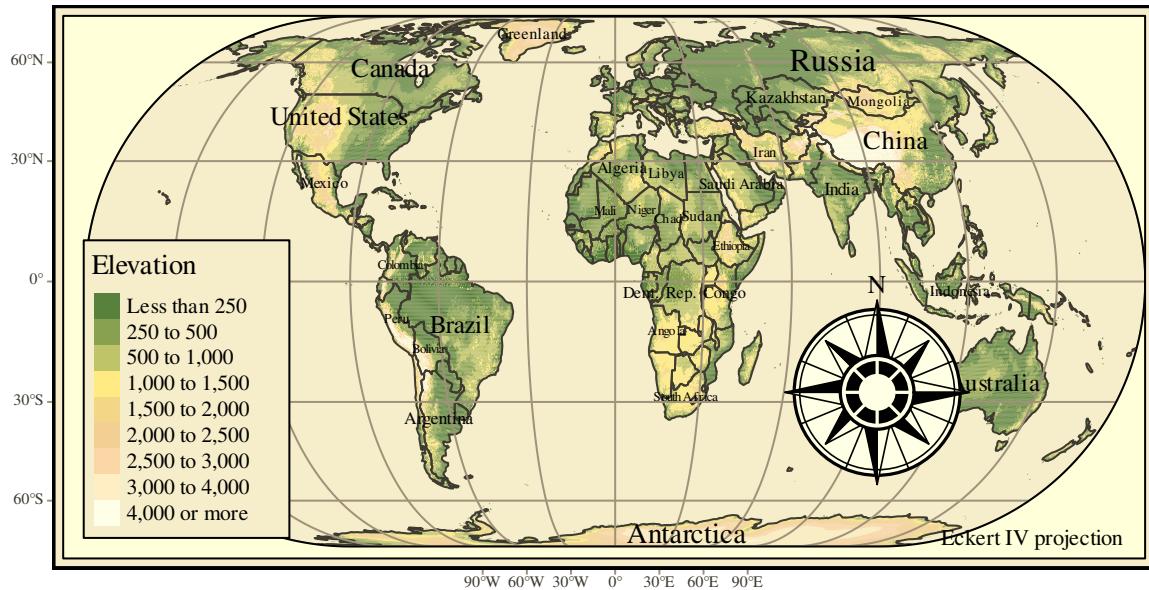
## Plot
tm_shape(land_eck4) +
  tm_raster(
    "elevation",
    breaks=c(-Inf, 250, 500, 1000, 1500, 2000, 2500, 3000, 4000, Inf),
    palette = terrain.colors(9),
    title="Elevation"
  ) +
  tm_shape(World) +
  tm_borders("grey20") +
  tm_graticules(labels.size = .5) +
  tm_text("name", size="AREA") +
  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. Importantly, 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.
- 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.