Geographic Data Visualization in R

Andrew McCormack

2019-01-10

Contents

1 *Overview of workshop*

In this workshop, you will learn how to visualize spatial data in R. Spatial data are widely available for many different political contexts, making them an excellent resource for communicating information that is geographic in nature. Despite their abundance as a data source, working with specialized Shapefiles can be daunting to the uninitiated. The aim of this workshop is disentangle many of challenges of working with spatial data in R. We begin with the basics of importing and manipulating shapefiles. Next, we cover how to create beautiful visualizations with spatial data in the ggplot2 package. In the remainder of the workshop, participants will learn how generate interactive maps using the leaflet package.

1.1 Getting started

There are a number of packages for working with spatial data in R. For importing and manipulating spatial data, we use functions from the `rgdal`, `rmapshaper`, and `sp` packages. For plotting static maps, we will use the `ggplot2` package, which is included in the `tidyverse` suite of packages. Standard data manipulation will be performed with function from `dplyr` and `broom` (both part of the `tidyverse`). To create interactive maps, we will use the `leaflet` package.

You may have all, some, or none of these packages installed already. For this reason, we use the `p_load()` function from the `pacman` package, which will load the packages you have installed and install and load the packages you don't have installed:

```
library(pacman)

p_load(# Standard data manipulation packages
       tidyverse,
       # Spatial data packages
       rgdal,
       rmapshaper,
       sp,
       # Interactive mapping
       leaflet)
```

2 Shapefiles

2.1 What is a Shapefile?

A shapefile is a file format used for storing geographic vector data. While geographic features in a shapefile can be represented by points, lines, or polygons, we will work primarily with polygons. While shapefiles were created for use with geographic information system (GIS) software, they have become increasingly accessible in R thanks to a number of useful packages designed for working with spatial data.

To demonstrate how to work with shapefiles in R, we will work with a shapefile containing boundaries for Québec Provincial electoral divisions. These files can be downloaded here.[1] Make sure that

[1] I have obtained this Shapefile from Élections Québec.

all these files are located in a dedicated folder. I have named this folder `quebec_prov_ridings`.

The folder `quebec_prov_ridings` contains 5 different files:

```
## [1] "qc_ridings.CPG" "qc_ridings.dbf"
## [3] "qc_ridings.prj" "qc_ridings.shp"
## [5] "qc_ridings.shx"
```

If this is your first time working with a shapefile, you may be surprised to notice that there are several individual files named `qc_ridings`, each with a different file extension. Though its name suggests a single file, a shapefile is actually a group of files containing feature geometry (in our case, these are the polygons that make up the map of Quebec provincial ridings) and feature attribute data. While different data sources will provide different types of constituent files, three of these files are essential: `.shp`, `.shx`, and `.dbf`:

- `.shp`: the file that contains the geometry for all features
- `.shx`: indexes the geometry, it allows GIS systems to find features within the .SHP file more quickly
- `.dbf`: contains feature attributes in tabular format

2.2 *How to load a shapefile into R*

R has a number of packages containing functions for importing shapefiles. We will use the `readOGR` function from the `rgdal` package. `readOGR` takes two main arguments. The first argument, `dsn` (short for data source name), specifies the folder where our shapefile is located.[2] The second argument, `layer`, specifies the file name without an extension (i.e. I use `"qc_ridings"`, not `"qc_ridings.shp"`). Using the assignment operator `<-`, I assign the Shapefile to an object named `rd`:

[2] Because my working directory is set to the folder containing the `quebec_prov_ridings` folder, I simply type `quebec_prov_ridings`. If you are working from within the directory where the Shapefile is located, specify `dsn = "."`.

```
rd <- readOGR(dsn = "quebec_prov_ridings", layer = "qc_ridings")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "/Users/andrewmccormack/Desktop/geoviz_workshop/quebec_prov_ridings", layer: "qc_ridings"
## with 125 features
## It has 3 fields
```

After reading in the shapefile, we receive a comment telling us our shapefile contains 125 features (one for each riding) and 3 fields (3 variables, discussed below). Now that we've imported the Shapefile into our environment, let's see what we are working with:

```
summary(rd)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##         min     max
## x  79569.03 1697055
## y 113908.06 2105873
## Is projected: TRUE
## proj4string :
## [+proj=lcc +lat_1=50 +lat_2=46 +lat_0=44
## +lon_0=-70 +x_0=800000 +y_0=0
## +datum=NAD83 +units=m +no_defs
## +ellps=GRS80 +towgs84=0,0,0]
## Data attributes:
##      CO_CEP                  NM_CEP
##  Min.   :104.0   Abitibi-Est     :  1
##  1st Qu.:300.0   Abitibi-Ouest   :  1
##  Median :476.0   Acadie          :  1
##  Mean   :491.1   Anjou-Louis-Riel:  1
##  3rd Qu.:702.0   Argenteuil      :  1
##  Max.   :938.0   Arthabaska      :  1
##                  (Other)         :119
##           NM_TRI_CEP
##  ABITIBIEST    :  1
##  ABITIBIOUEST  :  1
##  ACADIE        :  1
##  ANJOULOUISRIEL:  1
##  ARGENTEUIL    :  1
##  ARTHABASKA    :  1
##  (Other)       :119
```

The first line of output tells us that `rd` is of the class `SpatialPolygonsDataFrame`. This suggests that the shapefile has a polygon layer[3] (`SpatialPolygons`) and also that is has an attributes table (`DataFrame`). Under `Coordinates:`, we are given the bounding box of the map in latitude and longitude. This is a rectangle that encompasses the entire map:

[3] In our case, the polygon layer contains the outlines of Québec provincial electoral boundaries. It contains a number of geographic points that, when connected with lines, will plot our map of Québec.

```
        min     max
x  79569.03 1697055
y 113908.06 2105873
```

The next few lines give us the projection information:

```
Is projected: TRUE
```

```
+proj=lcc +lat_1=50 +lat_2=46 +lat_0=44 +lon_0=-70 +x_0=800000 +y_0=0 +datum=NAD83 +units=m +no_defs +ellp
```

As you may know, the Earth is round. In order to plot maps in a two-dimensional space, spherical coordinates (such as latitude and longitude) must be transformed to planar coordinates (x and y). This is a complicated topic subject to much debate. For the purposes of this workshop, we will accept Élections Québec's projection.

The last few lines of the ouput from `summary(rd)` gives us a summary of each column in the attributes table. This is the dataframe associated with the `qc_ridings` Shapefile. Because they don't have very intuitive names, here is a description of each variable:

- `CO_CEP`: the electoral district code
- `NM_CEP`: the official name of the electoral district
- `NM_TRI_CEP`: electoral district names transformed into non-accented capital letters, removing all characters that are not letters

These variables will be very useful when it comes to merging in other sources of data.

2.3 *How to access elements in a `SpatialPolygonsDataFrame`*

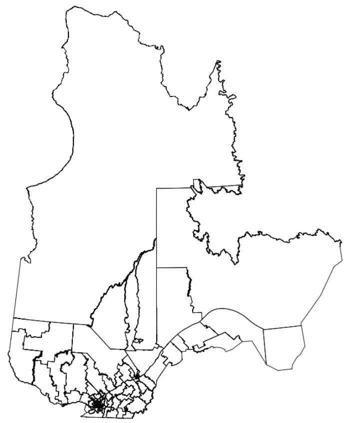
If you are used to working with data frames and lists in R, then you will be familiar with using the dollar-sign operator (`$`) as well as square brackets (`[` or `[[`) to extract content from objects. To access the contents of `SpatialPolygonsDataFrames`, we use the `@` operator. For instance, if we want to access the data in `rd`:

```
head(rd@data)
```

```
##   CO_CEP          NM_CEP    NM_TRI_CEP
## 0    360 Bourassa-Sauvé BOURASSASAUVE
## 1    918      Chicoutimi    CHICOUTIMI
## 2    648     Abitibi-Est    ABITIBIEST
## 3    212      Saint-Jean     SAINTJEAN
## 4    720      La Peltrie     LAPELTRIE
## 5    238         Chambly       CHAMBLY
```

Let's take a look at this Shapefile with the `plot()` function:

```
plot(rd)
```



While the `plot()` function is a good method for making sure that the shapefile is looking and working as it should, below we will construct our plots using the `ggplot2` package, and not `plot()` (which is part of base R graphics).

2.4 *Simplifying shapefiles*

Often, shapefiles will be very large and it will take your computer a while to load them into R and even longer to manipulate and plot them. When creating static maps, we often don't need the level of precision that large shapefiles provide. For this reason, it is often desirable to simplify our shapefile after importing it into R.

To do this, we can use the `ms_simplify()` function from the `rmapshaper` package. The `ms_simplify()` function will samples points from the polygons in `rd` to reduce it's size while preserving the polygons' (ridings') shapes. While this function has a number arguments, we will focus on the essentials: `input`, `keep`, and `keep_shapes`:

- The `input` argument is where we specify the Shapefile we want to simplify
- The `keep` argument is the proportion of points from the original Shapefile we want to retain. We can get away with a surprising amount of reduction here without substantially altering the appearance of the map. We specify `keep = 0.05`, which means we will keep only 5 percent of the original points.
- The `keep_shapes` argument, when `TRUE`, prevents small polygons from disappearing at high levels of simplifications. Because we don't want to lose any of our ridings in the simplification process, we specify `keep_shapes = TRUE`

```
rd_simple <- ms_simplify(input = rd, keep = 0.05,
    keep_shapes = TRUE)

# How does size of simplified Shapefile
# compare to original Shapefile?
(object.size(rd_simple)/object.size(rd))[1]

## [1] 0.08054286
```

After simplifying with `ms_simplify()`, the Shapefile is now only 8 percent of its original size. This will make visualizing these data much more efficient!

Let's plot this simplified Shapefile to see how it compares to our original plot:

```
plot(rd_simple)
```

As you can see, the map looks essentially the same. We were able to greatly reduce the size of the original Shapefile while not sacrificing a lot of detail. Depending on your system, you may also have noticed that this plot loaded much faster than the plot we created with the original shapefile.

2.5 *Converting a `SpatialPolygonsDataFrame` object into to data frame*

While the generic `plot()` function can use `SpatialPolygonsDataFrame` objects directly, `ggplot2` works better with data frames. Therefore we need to transform our `SpatialPolygonsDataFrame` object, `rd_simple`, into a data frame.

The `tidy()` function from the `broom` package comes in handy here. This function converts a variety of different R objects into data frame objects.[4] Most imporantly for us, `tidy()` can coerce a `SpatialPolygonsDataFrame` into a data frame. To do so, we input the object we want converted, which is our `SpatialPolygonsDataFrame` object, `rd_simple` into the `tidy()` function. We also need to specify a `region` from the attribute data of `rd_simple` so that we know which coordinates go with which provincial ridings:

```
rd_df <- broom::tidy(x = rd_simple, region = "NM_CEP")

# Examine the first few rows of the data frame
head(rd_df)
```

```
##       long      lat order  hole piece
## 1 387411.5 442553.4     1 FALSE     1
## 2 383820.0 442794.0     2 FALSE     1
## 3 372141.0 443647.0     3 FALSE     1
## 4 371718.0 442975.0     4 FALSE     1
## 5 371056.0 442660.0     5 FALSE     1
## 6 370695.0 441870.1     6 FALSE     1
##           group          id
## 1 Abitibi-Est.1 Abitibi-Est
## 2 Abitibi-Est.1 Abitibi-Est
## 3 Abitibi-Est.1 Abitibi-Est
## 4 Abitibi-Est.1 Abitibi-Est
## 5 Abitibi-Est.1 Abitibi-Est
## 6 Abitibi-Est.1 Abitibi-Est
```

This data frame has a number of variables that require our attention. To create maps in `ggplot()`, three of these are crucial:

- `long`: longitude, a measure of east-west position
- `lat`: latitude, a measure of north-south position
- `group`: an identifier that is unique for each region (in this case, Quebec provincial riding)

Unfortunately, in the process of converting `rd_simple` into a data frame, we lost all the attribute data (aside from the `id` column, which was originally `NM_CEP`) associated with the provincial ridings. To remedy this, we can merge the attribute data from the `SpatialPolygonsDataFrame` (`rd_simple`) with the data frame we just created (`rd_df`) using `dplyr`'s merge functions. By default `dplyr`'s merge functions look for a common variable, or common variables, between the two data frames we want to merge.[5] In our case, although we have two common variables, they have different names. For this reason, we need to explicity specify the variables we want to merge on:

[4] More specifically, `tidy()` convert a number of different model outputs into *tibbles*, which are a type of data frame. This distinction isn't important for our purposes, but you can read more here

[5] For more information on `dplyr`'s merge functions, I suggest this cheatsheat and/or this guide

```
rd_df <- right_join(rd_df, rd_simple@data, by = c(id = "NM_CEP"))
```

Now all of the variables from the original Shapefile have returned. In this case, the original Shapefile only had three variables, all of which were identifiers for the provincial riding. Other Shapefiles may contain a greater amount of attribute data, making it important to merge these two data frames.

We are now ready to use this data frame in `ggplot`. For those unfamiliar with the `ggplot2` package, I provide a brief overview below.

3 *`ggplot` refresher*

3.1 *The `ggplot2` package*

`ggplot2` is a powerful package for data visualization that allows you to create many types of plots with a great deal of flexibility. It is especially useful for making maps in R. `ggplot2` is based on the *Grammar of Graphics*—quantitative plots are composed of elements (data, aesthetics, geometries, scales, etc.) that convey precise and clear messages much like the grammatical elements of sentences. To create quantitative plots, we work with a number layered elements. The strength of `ggplot2` is that each of these elements can be added iteratively (i.e. we can add one element at a time to create highly customized plots). Let's start from scratch with the first and most important element: data.

3.2 *Geographic data*

To plot maps in ggplot, we need 3 basic elements: geographic coordinates, a grouping variable, and some variable we wish to represent geographically. To illustrate, we will create some fictional data:

```
df <- data.frame(x = c(2, 3, 1, 4, 5, 6), y = c(1,
    2, 3, 5, 8, 9), group = factor(c("t1", "t1",
    "t1", "t2", "t2", "t2")), variable = c("blue",
    "blue", "blue", "red", "red", "red"))
df
```

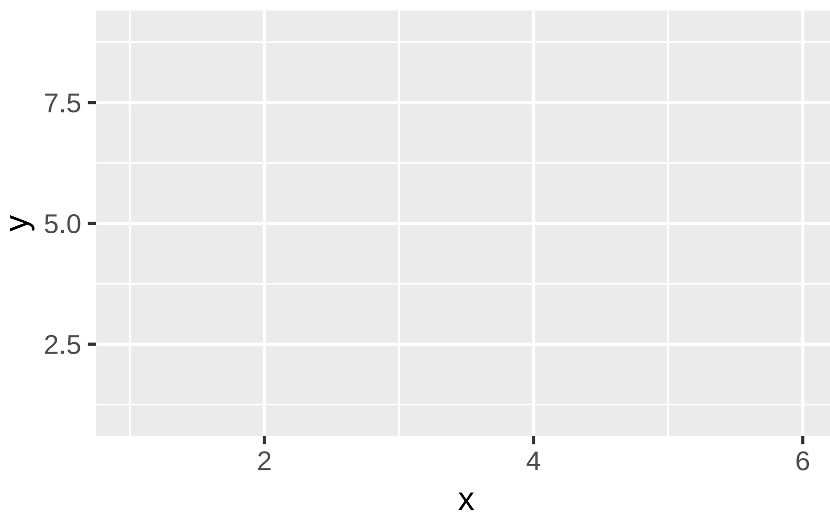
```
##   x y group variable
## 1 2 1    t1     blue
## 2 3 2    t1     blue
## 3 1 3    t1     blue
## 4 4 5    t2      red
## 5 5 8    t2      red
## 6 6 9    t2      red
```

The x and y variables are our geographic coordinates, which in this case are just two triangular "islands". Our grouping variable, group, will be used to tell ggplot that these triangles are two discrete objects. Our variable, variable, is simply the variable we want to visualize on our map.

3.3 *Aesthetic mapping*

Aesthetics refer to the variables we want to present. Aesthetic mappings in ggplot2, which go inside the aes() argument, define the variables that will be represented on our horizontal (x) and vertical (y) axes as well as how the data will be grouped. Let's initialize a ggplot object with the data we just created with an aesthetic mapping:

```
ggplot(data = df, mapping = aes(x = x, y = y))
```

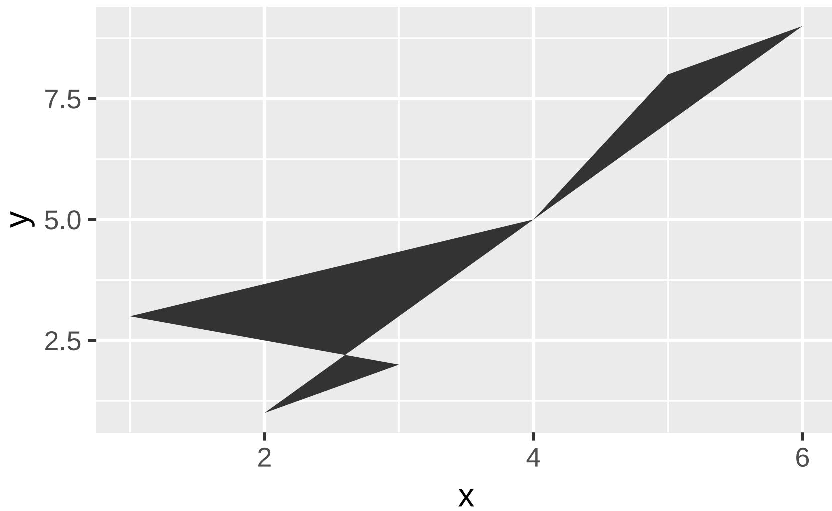


There's not much going on here. We need to tell ggplot the type of visual elements we want to plot—which in this case are geographic coordinates.

3.4 *Geometries*

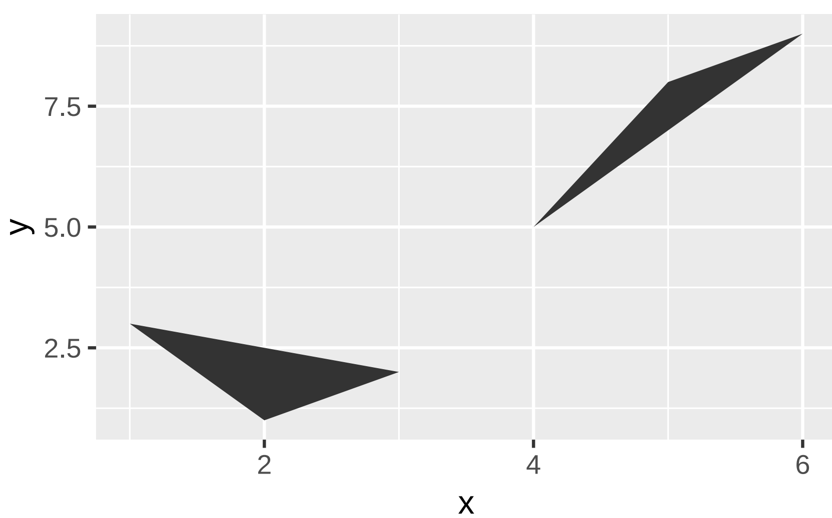
Visual elements in ggplot2 are called geoms (as in geometric objects). The appearance and location of these geoms are controlled by the aesthetic properties. There are many different geoms to choose from in ggplot, but for creating maps, geom_polygon() will be the most useful:

```
ggplot(data = df, mapping = aes(x = x, y = y)) +
    geom_polygon()
```



You will notice that this plot does not contain the two triangular islands that I promised above. The reason for this is that if we feed `geom_polygon()` only x and y coordinates, it just connects the points with no regard for how the points are grouped. This is where the grouping variable comes in. The grouping variable will tell `geom_polygon()` that each triangle is its own distinct region:

```
ggplot(df, aes(x, y, group = group)) + geom_polygon()
```

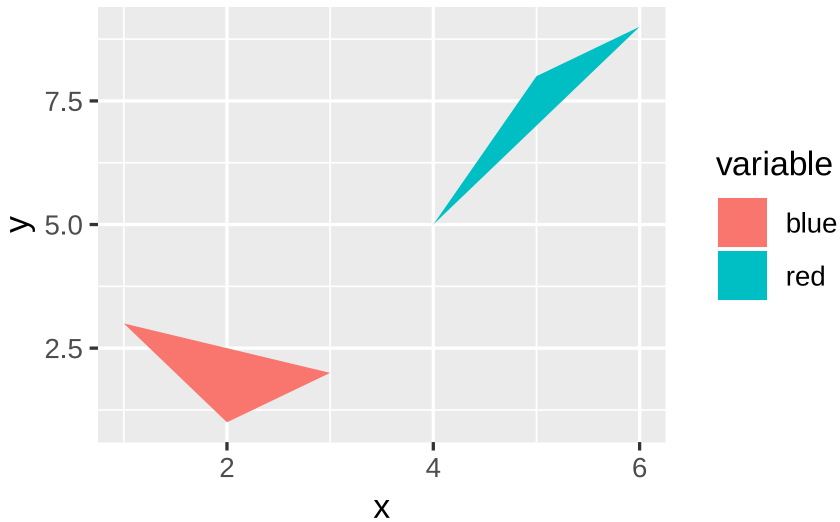


[6]

This looks better, but it doesn't tell us much about the triangles. From the `variable` column in our data, `df`, we know that triangle one (`t1`) is categorized as `red` and triangle two (`t2`) is categorized as `blue`. We can illustrate this in the plot with a `fill` aesthetic:

```
ggplot(df, aes(x, y, group = group, fill = variable)) +
    geom_polygon()
```

[6] Notice that I no longer explicitly specify `data = df` or `mapping = aes(x, y, group = group)`. If these arguments are in the correct order, `ggplot` (and all R functions for that matter) will know what we are referring to and pick this up implicitly.

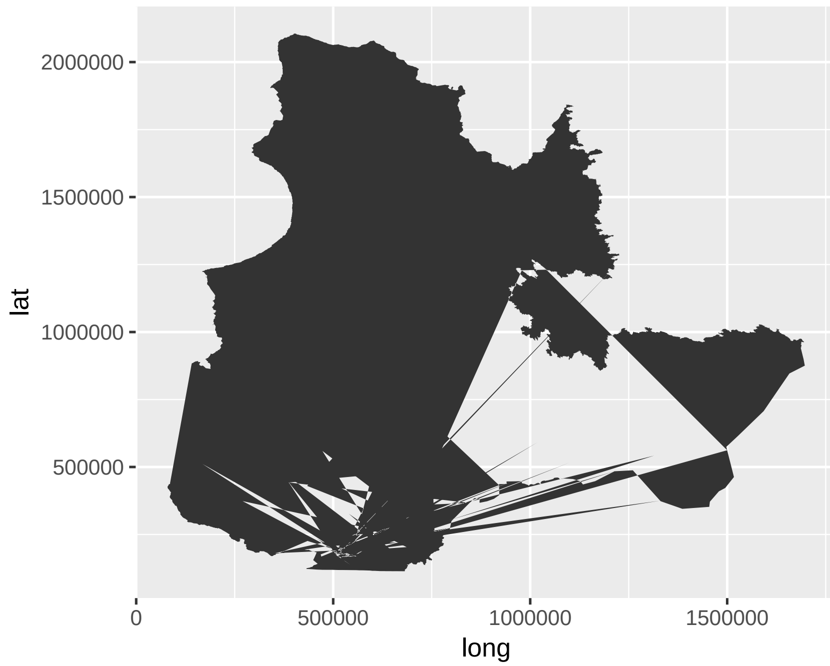


This basic example provides the foundation for plotting geographic data in `ggplot`. We will follow similar procedures throughout the workshop, though the polygons we plot will be Quebec provincial ridings, not triangles. Moreover, we will work with actual latitudinal and longitudinal values rather than fictional `x` and `y` coordinates.

4 *Plotting spatial data in `ggplot2`*

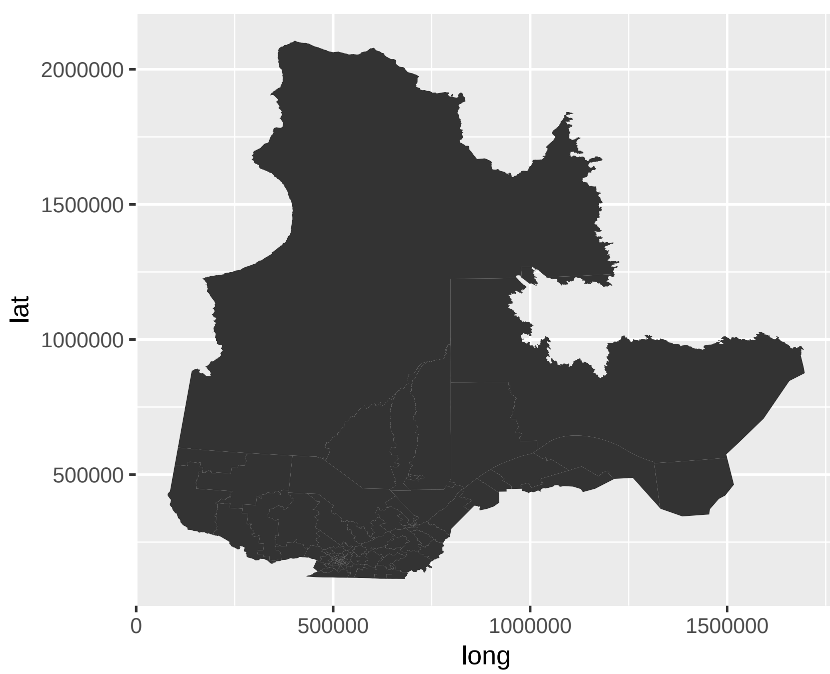
To create a map with the `rd_df` data frame we just created, the following aesthetic mappings are required: `x = long` (longitude), `y = lat` (latitude), and `group = group` (this tells `geom_polygon()` how to group observations—in this case, provinces).

```
ggplot(rd_df, aes(x = long, y = lat)) + geom_polygon()
```



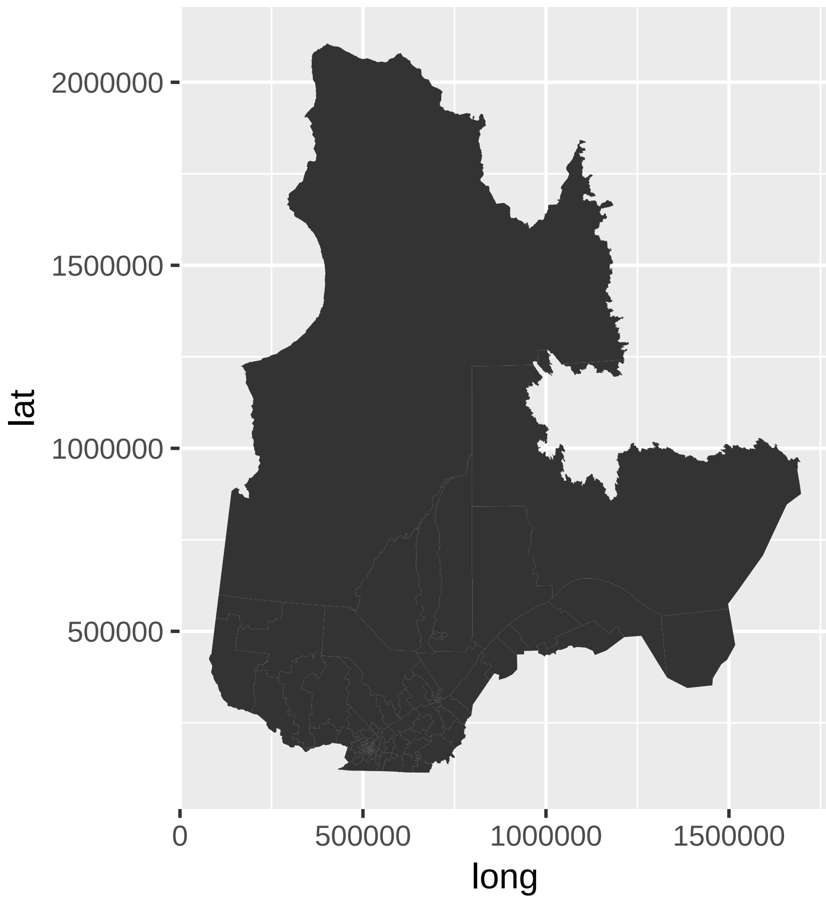
This map does not look very good. What did I miss here? Yes, you are right: we need a grouping variable so that `geom_polygon()` knows that each riding is its own polygon.

```
ggplot(rd_df, aes(x = long, y = lat, group = group)) +
    geom_polygon()
```



This map looks better, but there are still a number of issues. First, it looks like `ggplot` alters the shape of the map based on the width and length of the plot. While this may work for other types of plots, it does not work for maps. To get an accurate map of Quebec, we need to specify `coord_fixed()`. This fixes the relationship between the axes such that one unit on the x-axis (longitude) is the same length as one unit on the y-axis (latitude):

```
ggplot(rd_df, aes(x = long, y = lat, group = group)) +
    geom_polygon() + coord_fixed()
```



You will also notice that the axis text has no substantive significance. You can remove it, along with the axis ticks and background grid using a theme of your choice. For a simple presentation, I suggest `theme_void()`.[7]

```
ggplot(rd_df, aes(x = long, y = lat, group = group)) +
    geom_polygon() + coord_fixed() + theme_void()
```

[7] Two other ggplot theme options that are useful for maps are `theme_map()` from the `hrbrthemes` package and `theme_map()` from the `ggthemes` package.

5 Incorporating data into the map

So far, we have just plotted raw coordinates, which is not very exciting. Our map will be far more informative if we incorporate some Quebec provincial riding-level statistics. To do so, we need to find and then merge riding-level data with our geographic coordinates (`rd_df`). I have included some example data with this workshop, which can be loaded with the `read.csv()` function below:

```
qc_val <- read.csv("https://raw.githubusercontent.com/mccormackandrew/geoviz_workshop/master/r_data/qc_val
    stringsAsFactors = FALSE)
head(qc_val)
```

```
##        riding_name winning_party vote_share
## 1      Abitibi-Est           CAQ       42.2
## 2    Abitibi-Ouest           CAQ       34.1
```

```
## 3          Acadie          LIB       53.8
## 4 Anjou-Louis-Riel          LIB       39.1
## 5       Argenteuil          CAQ       38.9
## 6       Arthabaska          CAQ       61.8
##   riding_code mother_tongue_english
## 1         648           0.024341486
## 2         642           0.009239192
## 3         338           0.072998163
## 4         366           0.040098338
## 5         520           0.122285216
## 6         144           0.006391514
##   mother_tongue_french immigrant_share
## 1            0.9439087     0.019955911
## 2            0.9810654     0.007155026
## 3            0.3526771     0.530036234
## 4            0.6195320     0.331981836
## 5            0.8554732     0.037876836
## 6            0.9780377     0.023284148
##   aboriginal_share vismin_population
## 1      0.083884441       0.018795684
## 2      0.036229415       0.005791506
## 3      0.003305785       0.476350922
## 4      0.005270840       0.311628284
## 5      0.014858713       0.014515159
## 6      0.006972179       0.018005821
##   low_income_share unemployment_rate
## 1            0.058               7.3
## 2            0.052               7.9
## 3            0.195              11.4
## 4            0.131               9.5
## 5            0.068               7.2
## 6            0.075               5.8
##                  region
## 1 Abitibi-Témiscamingue
## 2 Abitibi-Témiscamingue
## 3              Montréal
## 4              Montréal
## 5           Laurentides
## 6      Centre-du-Québec
```

These data come from Quebec's chief returning officer (DGEQ) and are based on the 2016 census. For the purpose of this workshop, I narrowed the data down to a number of relevant socio-demographic indicators.

Let's merge these data with our riding coordinates data.

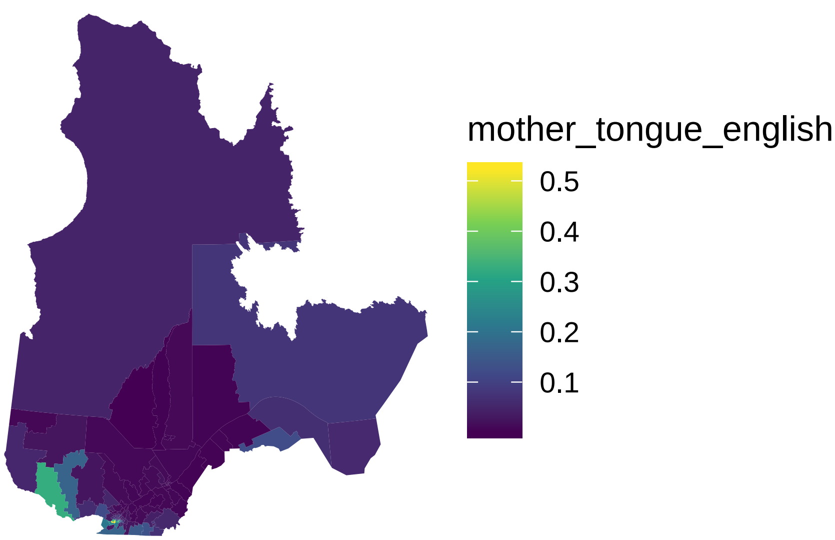
```
rd_df <- left_join(rd_df, qc_val, c(id = "riding_name"))
head(rd_df)
```

```
##       long      lat order  hole piece
## 1 516813.0 182778.1  2263 FALSE     1
## 2 516184.9 181996.2  2264 FALSE     1
## 3 515150.2 182931.0  2265 FALSE     1
## 4 515815.5 184227.9  2266 FALSE     1
## 5 515501.8 184610.4  2267 FALSE     1
## 6 516101.0 185963.0  2268 FALSE     1
##              group             id CO_CEP
## 1 Bourassa-Sauvé.1 Bourassa-Sauvé    360
## 2 Bourassa-Sauvé.1 Bourassa-Sauvé    360
## 3 Bourassa-Sauvé.1 Bourassa-Sauvé    360
## 4 Bourassa-Sauvé.1 Bourassa-Sauvé    360
## 5 Bourassa-Sauvé.1 Bourassa-Sauvé    360
## 6 Bourassa-Sauvé.1 Bourassa-Sauvé    360
##      NM_TRI_CEP winning_party vote_share
## 1 BOURASSASAUVE           LIB       46.2
## 2 BOURASSASAUVE           LIB       46.2
## 3 BOURASSASAUVE           LIB       46.2
## 4 BOURASSASAUVE           LIB       46.2
## 5 BOURASSASAUVE           LIB       46.2
## 6 BOURASSASAUVE           LIB       46.2
##   riding_code mother_tongue_english
## 1         360            0.04399775
## 2         360            0.04399775
## 3         360            0.04399775
## 4         360            0.04399775
## 5         360            0.04399775
## 6         360            0.04399775
##   mother_tongue_french immigrant_share
## 1            0.4993674       0.4220611
## 2            0.4993674       0.4220611
## 3            0.4993674       0.4220611
## 4            0.4993674       0.4220611
## 5            0.4993674       0.4220611
## 6            0.4993674       0.4220611
##   aboriginal_share vismin_population
## 1      0.006919034         0.4994345
## 2      0.006919034         0.4994345
## 3      0.006919034         0.4994345
## 4      0.006919034         0.4994345
```

```
## 5      0.006919034        0.4994345
## 6      0.006919034        0.4994345
##   low_income_share unemployment_rate
## 1            0.221              12.5
## 2            0.221              12.5
## 3            0.221              12.5
## 4            0.221              12.5
## 5            0.221              12.5
## 6            0.221              12.5
##     region
## 1 Montréal
## 2 Montréal
## 3 Montréal
## 4 Montréal
## 5 Montréal
## 6 Montréal
```

Now that we have the variables we need, let's create a more informative plot. To do this, we need to specify our variable of interest as a `fill` aesthetic inside ggplot. The `fill` aesthetic will colour the provincial ridings according to variable we feed it. I will feed it the `mother_tongue_english` variable, which is the proportion of the total population whose first language is English.

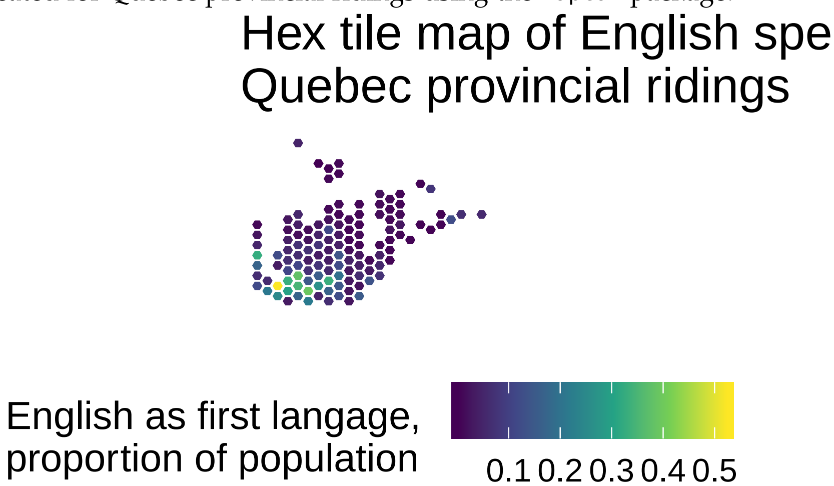
```
ggplot(rd_df, aes(x = long, y = lat, group = group,
    fill = mother_tongue_english)) + geom_polygon() +
    coord_fixed() + theme_void() + scale_fill_viridis_c()
```



Although this plots exactly what we had intended—mother tongue by provincial riding in Quebec—this map is not a very useful visual aid for understanding the proportion of Anglophones in Quebec.

It overemphasizes larger geographical units by assigning them a stronger visual weight. Because it is not land area itself we are interested in, this leads to a distorted impression of the data.

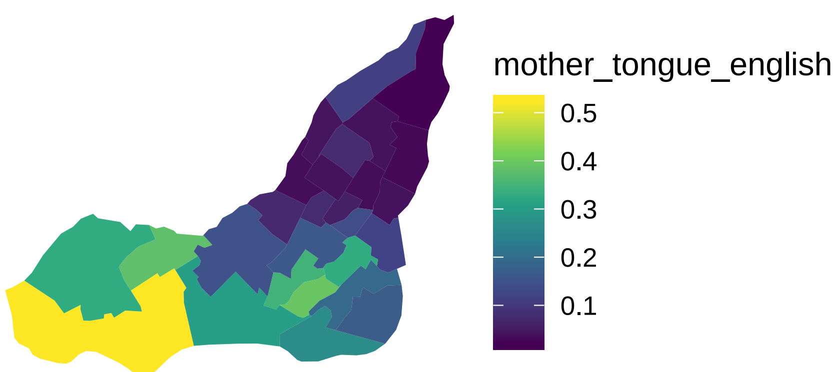
There are a number of solutions for this. First, we can use an alternative mapping technique, such as a hexagonal grid map. Though we don't cover hexagonal grid maps in this workshop, they can be created for Quebec provincial ridings using the `mapcan` package.



An overview of how `mapcan` works can be found here.

Another solution is to create a map that focuses on a smaller region of Quebec, like Montréal. To do this, we can subset `rd_df` to include only ridings from Montréal within `ggplot`:

```
ggplot(rd_df[rd_df$region == "Montréal", ], aes(x = long,
    y = lat, group = group, fill = mother_tongue_english)) +
    geom_polygon() + coord_fixed() + theme_void() +
    scale_fill_viridis_c()
```



By focusing on Montréal, we reveal some of the heterogeneity we missed out on with the full map of Quebec.

One last solution, which will be covered in the remainder of the workshop, is to create an interactive map that allows us to zoom in and out of the map as we desire.

6 Creating interactive maps in R

There are a few packages in R that provide tools for creating interactive maps. We will focus on one of the most powerful and user-friendly of these packages—`leaflet`. Leaflet is a Javascript library for interactive maps that can be used in R with the `leaflet` package.[8]

[8] Two other packages for interactive maps that are worth mentioning are `plotly`, and `rbokeh`

6.1 Preparing the data

Leaflet takes a `SpatialPolygonsDataFrame` (and other types of spatial data) as input, so there is no need to convert our `SpatialPolygonsDataFrame` to a data frame when using Leaflet. However, we do need to make some alterations before our data is ready for use in Leaflet.

First, the map projection that Élections Québec has provided will not work in Leaflet. We can easily change the projection to be more Leaflet-friendly with the `spTransform()` function from the `sp` package. We will use the `EPSG:4362` projection:[9]

[9] We choose `EPSG:4362` specifically because this is the projection that OpenStreetMap uses. Because we will overlay our `SpatialPolygonsDataFrame` onto a map created with OpenStreetMap data, we want the projections to match. Why OpenStreetMap and not Google Maps? Recently, Google announced that a billing account and API key would be mandatory for using Google Maps. OpenStreetMap is a free alternative.

```
# Transform CRS of Shapefile and assign to new
# SPDF object
rd_leaf <- spTransform(rd_simple, CRS("+init=epsg:4326"))
```

Next, we need to merge in some provincial riding-level data to plot. Remember that the only data our Shapefile contains are riding names. After we converted the `SpatialPolygonsDataFrame` to a data frame in the section above, we merged in relevant socio-demographic indicators found in the `qc_val` data frame. We can also merge these values right into the `@data` slot of `rd_simple` (a `SpatialPolygonsDataFrame`):

```
rd_leaf@data <- left_join(rd_leaf@data, qc_val,
    by = c(NM_CEP = "riding_name"))
```

6.2 Using Leaflet to plot a Shapefile

Now that our data is ready, we can put it to work in Leaflet. We will use the pipe operator extensively here.[10]

[10] The pipe operator allows us to structure sequences of operations from left-to-right, instead of from inside and out (nesting functions). This makes code much more flexible and readable. Read more here

In the first line, we simply initialize Leaflet (`leaflet()`). We then pipe this into the `addPolygons()` function, where we input our `SpatialPolygonsDataFrame` and add a few additional arguments to specify line width (`weight`), line colour (`color`)[11] and fill color (`fillColor`):

[11] Note that, unlike in `ggplot`, you must use the American spelling "color" for this argument. Do not use "colour".

```
leaflet() %>% addPolygons(data = rd_leaf, weight = 1,
    color = "red", fillColor = "bisque")
```

This creates a simple map of the provincial ridings that allows us to zoom in and out as we see fit.

6.3 *Overlaying a shapefile on a street map*

We can easily overlay our `SpatialPolygonsDataFrame` onto a street map using the `addTiles` function. This function will automatically place an OpenStreetMap map on the Leaflet plot:

```
leaflet() %>% addPolygons(data = rd_leaf, weight = 1,
    color = "red", fillColor = "bisque") %>% addTiles()
```

If the default OpenStreetMap does not suit your tastes or needs, there are many third-party maps available. These can be accessed with the `addProviderTiles` function. In our case, a black and white map may be a better option, as it will allow us to see the red riding division lines more clearly:

```
leaflet() %>% addPolygons(data = rd_leaf, weight = 1,
    color = "red", fillColor = "bisque") %>% addProviderTiles(providers$OpenStreetMap.BlackAndWhite)
```

Aside from riding names, this doesn't tell us much. Let's incorporate some riding level information!

Using the `label` argument inside `addPolygons()`, we can make the following information appear when we hover over a riding: (1) the name of the riding (`NM_CEP`), (2) the winning party of that riding in the 2018 provincial election (`winning_party`), and (3) the winning party's share of the vote (`vote_share`). We use the `str_c()` function from the `stringr` package to combine all of this information:

```
leaflet(rd_leaf) %>% addPolygons(label = ~stringr::str_c(NM_CEP,
    ", ", winning_party, ", ", vote_share, ", "),
    weight = 1, color = "red", fillColor = "bisque") %>%
    addProviderTiles(providers$OpenStreetMap.BlackAndWhite)
```

We can make this map more informative by shading the ridings according to the winning party of the 2018 election (or any other variable of our choice). To do this, we use the `colorFactor()` function from the `leaflet` package, which maps colours to a variable of our choice.[12] With this function, you specify the colours you want to use as well as the range of inputs (in this case, the parties in the `winning_party` variable) you want to match them to. Inputting a vector of values (the `winning_party` variable in our case) into the function returns a vector of colours in `#RRGGBB` format. We can then use this function directly inside the `addPolygon()` function:

[12] `colorFactor()` is one of a number of `color*` helper functions included in `leaflet` package. Depending on the data, you may also with to use `colorNumeric`, `colorBin`, or `colorQuantile`. Read more here

```
# Create partycol function to colour ridings
# by party colour
partycol <- colorFactor(palette = c("deepskyblue1",
    "red", "royalblue4", "orange"), domain = c("CAQ",
```

```
    "LIB", "PQ", "QS"))

# Create leaflet plot
leaflet(rd_leaf) %>% addPolygons(label = ~stringr::str_c(NM_CEP,
    ", ", winning_party, ", ", vote_share, ", "),
    weight = 1, color = ~partycol(winning_party)) %>%
    addProviderTiles(providers$OpenStreetMap.BlackAndWhite)
```

Lastly, we can also add a legend to the plot using the `addLegend()` function:

```
leaflet(rd_leaf) %>% addPolygons(label = ~stringr::str_c(NM_CEP,
    ", ", winning_party, ", ", vote_share, ", "),
    weight = 1, fillOpacity = 0.5, color = ~partycol(winning_party)) %>%
    addProviderTiles(providers$OpenStreetMap.BlackAndWhite) %>%
    addLegend(pal = partycol, values = ~winning_party,
        opacity = 0.5)
```

These are the basics of working with Shapefiles in leaflet. There are many more things that can be done with `leaflet` that were not covered in this workshop, such as adding markers, popup boxes, and Shiny integration.

7 *Conclusion*

This workshop has introduced the basics of static and interactive geographic data visualization in R. From here, you should be in strong position to use your own shapefiles to produce powerful and informative data visualizations. Note that we have only skimmed the surface of working with spatial data in R. There are a variety of online guides and tutorials available if you wish to learn more. I suggest the following sources (many of which I consulted while putting together this workshop):

- To learn more about `leaflet`, I suggest you visit the Leaflet for R website. In addition, this post provides an overview of how to use `leaflet` to create and publish web maps.
- For a more comprehensive overview on working with shapefiles in R, Robin Lovelace has an excellent guide.
- Bhaskar V. Karambelkar has created a large number of tutorials on many different aspects of GeoSpatial Data Visualization, available here.
- Though we used the `rgdal`, `rmapshaper`, and `sp` packages, the newer `sf` package is becoming increasingly popular for working with spatial data and plotting maps in R