2-2, Lesson 2 - D3 Projections and Path Generators

2.1 Creating a D3 Projection

Now that you have imported your geographic data, the next step is to project it. Here it may be helpful to step back and review what a projection is, since projections were one of the most complex topics covered in your earlier Cartography and GIS courses. To get down to basics, a projection is a set of mathematical equations used to stretch and distort geographic coordinates based on a rounded ellipsoid so that they can be displayed on a planar (two-dimensional) surface, such as your computer screen. All projections necessarily distort some visual property of the geography: areas, distances, directions, and/or angles. They vary by the distorted property/properties, the shape of the plane (**class**), by how many times the plane intersects the ellipsoid (**case**), and by rotation angle of the plane from north (**aspect**). Figure 2.1 demonstrates the distortion that occurs in even the simplest of projections, the Plate Carrée, an equidistant cylindrical projection. This projection is produced by the set of equations [x = λ, y = ϕ], where x and y are horizontal and vertical coordinates on a two-dimensional Cartesian grid, λ (lamda) is longitude, and ϕ (phi) is latitude.

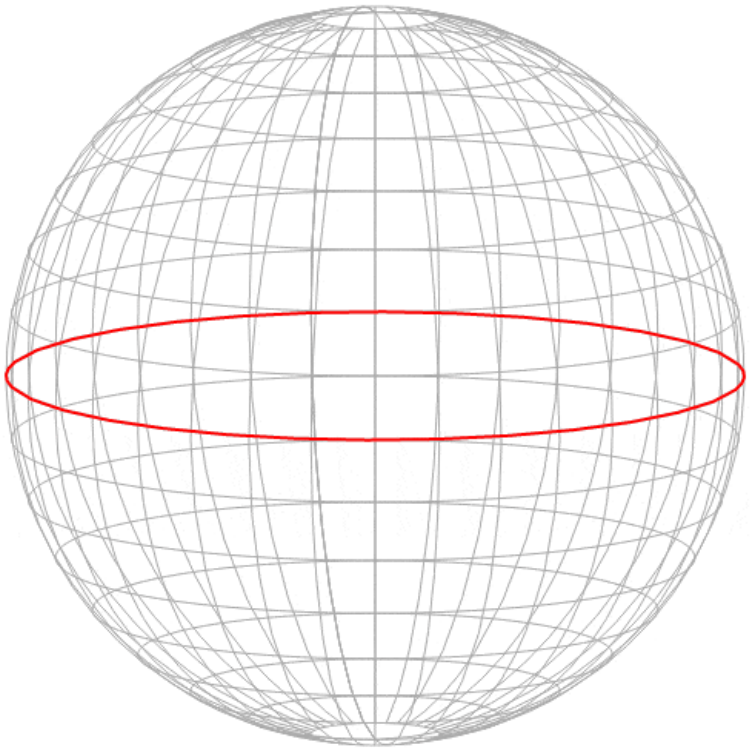

*Figure 2.1: Projecting the globe onto a two-dimensional surface using the Plate Carrée projection (**original graphic** **(http://bl.ocks.org/mbostock/5731632)** by Mike Bostock)*

Fortunately for us, modern desktop mapping software does the dirty work of applying projections to our chosen spatial datasets so we don't have to deal with the heavy math. All we have to do is pick out a projection that is cartographically appropriate given the type and scale of the map we want to make.

Unfortunately for cartographers, with the advent of tile-based slippy maps—such as the one you created for your Leaflet lab assignment—one projection became dominant on the Web: so-called **Web Mercator** **(https://en.wikipedia.org/wiki/Web_Mercator)**. This projection was created and popularized by Google in the mid-2000's (before it was assigned an official EPSG code (now 3857), it was unofficially referenced using the code EPSG:900913—a clever pun). It was chosen by Google for its technical advantages: it is a relatively simple equation, a cylindrical projection that can be made infinitely continuous to the east and west, and conformal so it preserves angles at high latitudes, making it good for navigation at high zoom levels anywhere on the planet. But for thematic mapping, it suffers from the disadvantage of severe area distortion at high latitudes, exaggerating the land area of the northern hemisphere (e.g., Greenland appears to be twice the size of Australia when in reality it is much smaller). While it is possible to make slippy map tilesets in other projections, it remains rare. Like it or not, for a slippy map, Web Mercator is likely to remain the dominant projection for the forseeable future.

D3 presents an entirely different scene. One of its great advantages for cartographers is that it supports hundreds of different map projections, thanks to the collaboration between Mike Bostock and data visualization artist **Jason Davies** **(https://www.jasondavies.com/)**. Several common projections are included in D3 through the **Geo Projections** **(https://github.com/d3/d3-geo/blob/master/README.md#projections)** portion of the library (Figure 2.2). But many others can be added through the **Extended Geographic Projections** **(https://github.com/d3/d3-geo-projection/)** and **Polyhedral Geographic Projections** **(https://github.com/d3/d3-plugins/tree/master/geo/polyhedron)** plugins. Not only can you choose which projection to use with your spatial data; you can change virtually any parameter that goes into each projection. D3 even enables you to smoothly transition between **different projections** **(http://bl.ocks.org/mbostock/3711652)** and **projection parameters** **(https://www.jasondavies.com/maps/transition/)**.

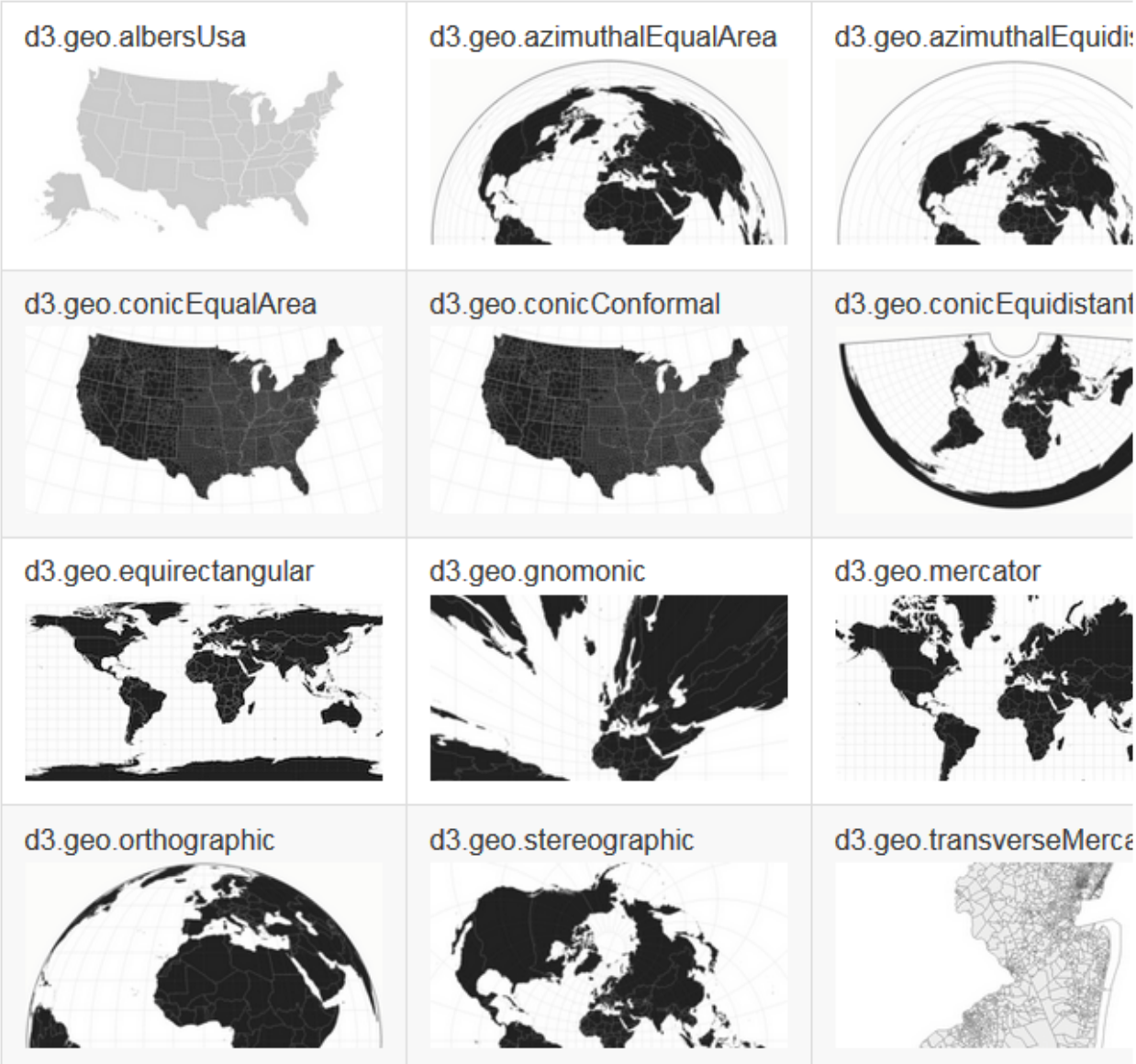

Figure 2.2: Projections included in D3's Geo Projections module

In the script, D3 implements projections using projection generators. Recall from previous Module that a D3 generator is a function that is returned by a D3 generator method and stored in a local variable. Any D3 projection method will return a **projection generator**, which must then be fed into the `d3.geo.path()` (https://github.com/d3/d3-geo/blob/master/README.md#geoPath) method to produce *another* generator —the path generator. Finally, the path generator is accessed within a selection block to draw the spatial data as path strings of the `d` attributes of SVG `<path>` elements. This process will become clearer as we build our generators below.

Let's start with the projection generator (Example 2.1). Since you will be creating a choropleth map, you should choose an equal-area projection so as to avoid incorrect visual interpretations of the mapped data. In this example, we will work with an **Albers equal-area conic projection** (https://en.wikipedia.org/wiki/Albers_projection) centered on France. You may wish to follow the example at first, then choose a different projection and/or edit the parameters to make the projection appropriate for your data.

Example 2.1: Creating an Albers projection generator in *main.js*

JavaScript

```
//Example 1.4 line 1...set up choropleth map
function setMap(){

    //map frame dimensions
    var width = 960,
        height = 460;

    //create new svg container for the map
    var map = d3.select("body")
        .append("svg")
        .attr("class", "map")
        .attr("width", width)
        .attr("height", height);

    //create Albers equal area conic projection centered on France
    var projection = d3.geoAlbers()
        .center([0, 46.2])
        .rotate([-2, 0, 0])
        .parallels([43, 62])
        .scale(2500)
        .translate([width / 2, height / 2]);

    //Example 1.4 line 3...use d3.queue to parallelize asynchronous data loading
    d3.queue
        .defer(d3.csv, "data/unitsData.csv") //load attributes from csv
        .defer(d3.json, "data/EuropeCountries.topojson") //load background spatial data
        .defer(d3.json, "data/FranceRegions.topojson") //load choropleth spatial data
        .await(callback);
```

In Example 2.1, before we can create the projection, we first write a `map` block to append the `<svg>` container that will hold the map and give it dimensions of 960 pixels by 460 pixels (lines 4-13). To create the projection, we use the `d3.geoConicEqualArea()` [(https://github.com/d3/d3-geo/blob/master/README.md#geoAlbers)](https://github.com/d3/d3-geo/blob/master/README.md#geoAlbers) projection method (line 16; note this is an alias of the `d3.geo.conicEqualArea()` method shown in Figure 2.2). The four operators on lines 17-20 are D3's way of implementing mathematical **projection parameters** [(https://github.com/mbostock/d3/wiki/Geo-Projections#_projection)](https://github.com/mbostock/d3/wiki/Geo-Projections#_projection):

- `.center()` [(https://github.com/d3/d3-geo/blob/master/README.md#projection_center)](https://github.com/d3/d3-geo/blob/master/README.md#projection_center) specifies the [longitude, latitude] coordinates of the center of the plane.
- `.rotate()` [(https://github.com/d3/d3-geo/blob/master/README.md#projection_rotate)](https://github.com/d3/d3-geo/blob/master/README.md#projection_rotate) specifies the [longitude, latitude, and roll] angles by which to **rotate the globe** [(http://bl.ocks.org/mbostock/4282586)](http://bl.ocks.org/mbostock/4282586).
- `.parallels()` [(https://github.com/d3/d3-geo/blob/master/README.md#conic_parallels)](https://github.com/d3/d3-geo/blob/master/README.md#conic_parallels) specifies the two standard parallels of a conic projection. If the two array values are the same, the projection is a **tangent** case (the plane intersects the globe at one line of latitude); if they are different, it is a **secant** case (the plane intersects the globe at two lines of latitude, slicing through it).
- `.scale()` [(https://github.com/d3/d3-geo/blob/master/README.md#projection_scale)](https://github.com/d3/d3-geo/blob/master/README.md#projection_scale) is a factor by which distances between points are multiplied, increasing or decreasing the scale of the map.

The fifth parameter, `.translate()` (https://github.com/d3/d3-geo/blob/master/README.md#projection_translate) (line 21), offsets the pixel coordinates of the projection's center in the `<svg>` container. Keep these as one-half the `<svg>` width and height to keep your map centered in the container.

Note that D3's projection parameters differ somewhat from the **projection parameters** (http://help.arcgis.com/en/geodatabase/10.0/sdk/arcsde/concepts/geometry/coordref/coordsys/projected/mapprojections.htm) commonly used by GIS software. Rather than treating the projection centering holistically, D3 breaks it down into the position of the ellipsoid and the position of the planar surface (also called the **developable surface**). The first two values given to `.rotate()` specify the globe's central meridian and central parallel, while `.center()` specifies the latitude and longitude of the plane's center. For conic projections, in order to keep north "up" in the center of the map and minimize distortion in your area of focus, you should keep the `.center()` longitude and `.rotate()` latitude each as `0` and assign the center coordinates of your chosen area as the `.center()` latitude and `.rotate()` longitude (Example 1.4 lines 17-18). If the geometric reasons for this are hard to grasp, you can experiment with different parameter values and see their effects using the Albers projection demonstration web app linked below.

> **Practice!** Experiment with the UW-Cart **D3 Albers Projection Demo** (http://uwcart.github.io/d3-projection-demo/) web application to see how different D3 parameter values affect the Albers projection. Then, visit the **D3 Geo-Projections** (https://github.com/d3/d3-geo/blob/master/README.md#projections) page and the **Extended Geographic Projections** (https://github.com/d3/d3-geo-projection/) page and choose a projection to implement that is cartographically appropriate given your chosen data. Write the projection block for your chosen projection in *main.js*.

2.2 Drawing Projected Data

Having created a projection function, we can now apply it to our spatial data to draw the represented geographies. In order to do this, we need to use `d3.geoPath()` to create a **path generator**(Example 2.2):

Example 2.2: Creating a path generator in *main.js*

JavaScript

```
//Example 2.1 line 15...create Albers equal area conic projection centered on France
var projection = d3.geoAlbers()
    .center([0, 46.2])
    .rotate([-2, 0])
    .parallels([43, 62])
    .scale(2500)
    .translate([width / 2, height / 2]);

var path = d3.geoPath()
    .projection(projection);
```

Creating the path generator is simple—we just create a two-line block, first calling `d3.geoPath()`, then using the `.projection()` operator to pass it our projection generator as the parameter (lines 9-10). The variable `path` now holds the path generator. Now let's apply it to draw the geometries from our spatial data (Example 2.3):

Example 2.3: Drawing geometries from spatial data in *main.js*

JavaScript

```
//Example 1.5 line 1
function callback(error, csvData, europe, france){
    //translate europe TopoJSON
    var europeCountries = topojson.feature(europe, europe.objects.EuropeCountries),
        franceRegions = topojson.feature(france, france.objects.FranceRegions).features;

    //add Europe countries to map
    var countries = map.append("path")
        .datum(europeCountries)
        .attr("class", "countries")
        .attr("d", path);

    //add France regions to map
    var regions = map.selectAll(".regions")
        .data(franceRegions)
        .enter()
        .append("path")
        .attr("class", function(d){
            return "regions " + d.properties.adm1_code;
        })
        .attr("d", path);
};
```

In Example 2.3, we add two blocks: one for the background countries (lines 8-11) and one for the regions that will become our choropleth enumeration units (lines 14-21). Because the `countries` block takes the `europeCountries` GeoJSON FeatureCollection as a single datum, all of its spatial data is drawn as a single feature. A single SVG `<path>` element is appended to the map container, and its `d` attribute is assigned the `path` generator. This automatically passes the datum to the `path` generator, which returns an SVG path coordinate string to the `<path>` element's `d` attribute. (Do not confuse the `<path> d` attribute with the variable `d` that iteratively holds each datum in a `.data()` block, such as on line 18 of Example 2.3). To recall what a path coordinate string looks like, see Figure 2.3.

To create our enumeration units, we use the `.selectAll().data().enter()` sequence to draw each feature corresponding to a region of France separately (lines 14-16. Recall that `.data()` requires its parameter to be in the form of an array, whereas `topojson.feature()` converts the TopoJSON object into a GeoJSON FeatureCollection object. For our `regions` block to work, we need to pull the array of features out of the FeatureCollection and pass that array to `.data()`, so we tack on `.features` to the end of line 5 to access it. Once that is done, a new `<path>` element is appended to the map container for each region (line 17). Two class names are assigned to each `<path>`: the generic class name `regions` for all enumeration units and a unique class name based on the region's `adm1_code` attribute (lines 18-20). Each `<path>` is then drawn with the region geometry by the `path` generator (line 21).

Now we can see our geometries in the browser and use the Inspector to distinguish each individual `<path>` element (Figure 2.3):

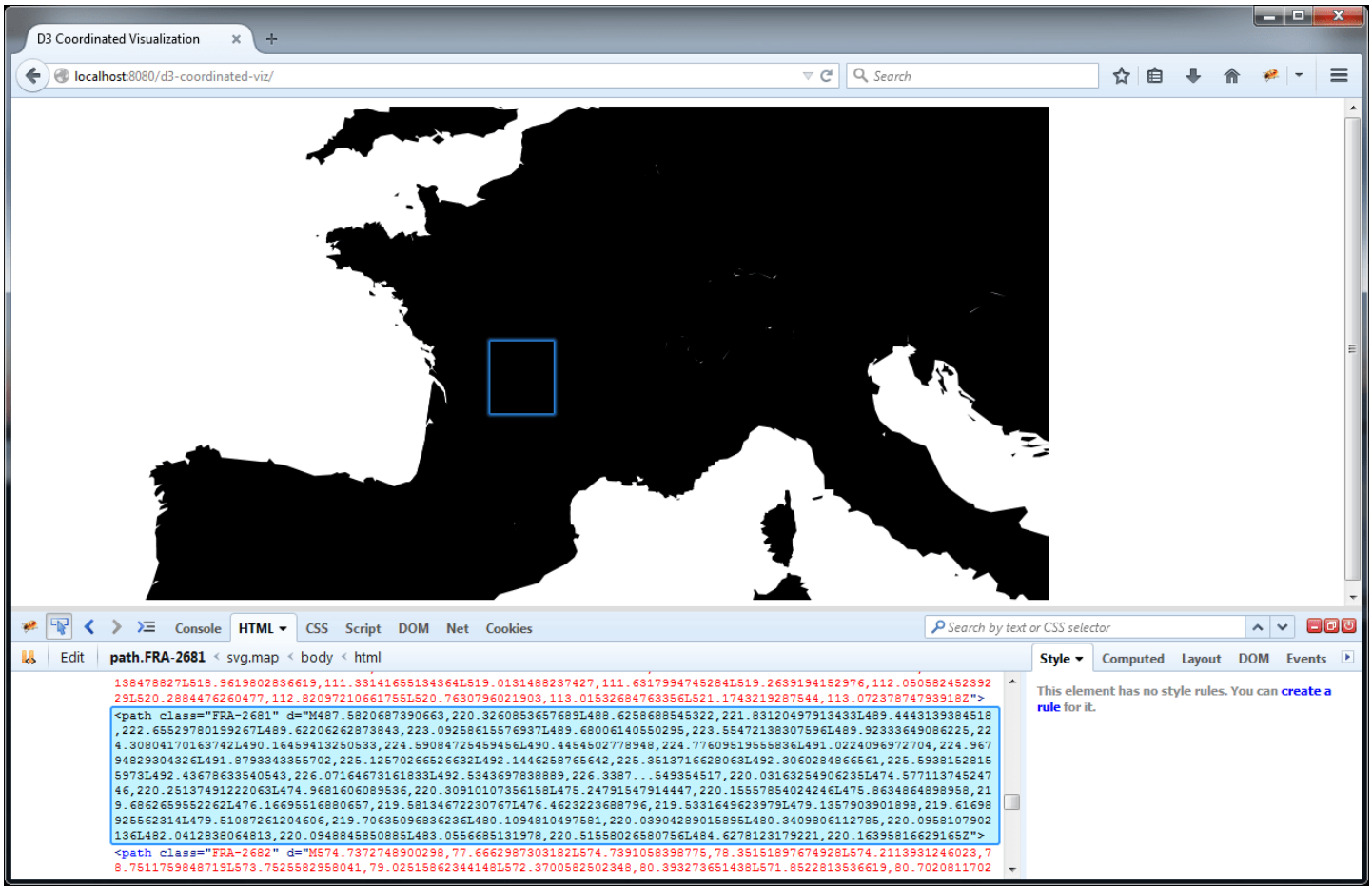

Figure 2.3: Spatial geometries drawn in the browser

If you think you have done everything right so far but you do *not* see your geometries in the browser—particularly if you get a bunch of seemingly random lines or polygons, or just a black map container—it is likely that your spatial data has been projected. In this case, D3 attempts to project the already-projected data, resulting in visual chaos. If your data is projected, you will need to return to Lesson 1 and "reproject" your data to unprojected WGS 84 before you can continue.

Obviously, we don't just want our map to be colored default black-and-white. As choropleth enumeration units, the regions will eventually be colored dynamically by the script, so we don't really have to worry about them for now. But it would be nice to outline the countries of Europe for reference. That we can do with some simple styles added to *style.css* (Example 2.4):

Example 2.4: Styling country borders in *style.css*

CSS

```
.countries {
    fill: #FFF;
    stroke: #CCC;
    stroke-width: 2px;
}
```

We can immediately see the result in the browser (Figure 2.4):

Figure 2.4: Styled country borders

Feel free to get more creative than this default style in your own D3 lab assignment.

> **Practice!** Create a path generator and use it to draw your background geometry and enumeration units in the browser. Style your background geometry in *style.css*.

III. Drawing a Graticule

With our geometries drawn, we could add a flat background color to the `<svg>` map container. But on a small-scale map such as our example, it is helpful to include a graticule to show your users the distortion of the projection. Providing a graticule is optional for your D3 lab assignment. If you want to include one, though, D3 provides a convenient `d3.geo.graticule()` (https://github.com/d3/d3-geo/blob/master/README.md#geoGraticule) method for creating a graticule generator. To use it, first create the graticule generator (Example 2.5):

Example 2.5: Creating a graticule generator in *main.js*

JavaScript

```
//Example 2.3 line 1
function callback(error, csvData, europe, france){
    //create graticule generator
    var graticule = d3.geoGraticule()
        .step([5, 5]); //place graticule lines every 5 degrees of longitude and latitude
```

The `.step()` (https://github.com/d3/d3-geo/blob/master/README.md#graticule_step) operator (line 5) tells the generator to place a graticule line every five degrees of longitude and latitude. Next, we can use our `graticule` generator to give us the spatial data for the graticule lines we will place on the map, and our `path` generator to draw the `<path>` element `d` strings for them (Example 2.6):

Example 2.6: Drawing graticule lines in *main.js*

JavaScript

```
//Example 2.5 line 3...create graticule generator
var graticule = d3.geoGraticule()
    .step([5, 5]); //place graticule lines every 5 degrees of longitude and latitude

//create graticule lines
var gratLines = map.selectAll(".gratLines") //select graticule elements that will be created
    .data(graticule.lines()) //bind graticule lines to each element to be created
    .enter() //create an element for each datum
    .append("path") //append each element to the svg as a path element
    .attr("class", "gratLines") //assign class for styling
    .attr("d", path); //project graticule lines
```

In Example 2.6, we use the `.selectAll().data().enter()` sequence to create a separate `<path>` element for each line of the graticule. The data is provided by the `graticule.lines()` (https://github.com/d3/d3-geo/blob/master/README.md#graticule_lines) method (line 7), which builds and returns a GeoJSON features array with all of the graticule lines selected by the `.step()` operator (line 3). To actually see the lines instead of a default black fill, we need to add another set of styles to *style.css* (Example 2.7):

Example 2.7: Graticule line styles in *style.css*

CSS

```
.gratLines {
    fill: none;
    stroke: #999;
    stroke-width: 1px;
}
```

We should now be able to see our graticule lines (Figure 2.5):

Figure 2.5: Europe with background graticule lines

Finally, we can add contrast between land and water by coloring the background of the graticule. For this, we can use the `graticule.outline()` (https://github.com/d3/d3-geo/blob/master/README.md#graticule_outline) method to create a single GeoJSON polygon the size of the graticule extent (most of the Earth's surface) and build an SVG `<path>` element for that polygon (Example 2.8):

Example 2.8: Drawing a graticule background in *main.js*

JavaScript

```
        //Example 2.6 line 1...create graticule generator
        var graticule = d3.geoGraticule()
            .step([5, 5]); //place graticule lines every 5 degrees of longitude and latitude

        //create graticule background
        var gratBackground = map.append("path")
            .datum(graticule.outline()) //bind graticule background
            .attr("class", "gratBackground") //assign class for styling
            .attr("d", path) //project graticule

        //Example 2.6 line 5...create graticule lines
        var gratLines = map.selectAll(".gratLines") //select graticule elements that will be created
```

We can then style the `gratBackground <path>` element to symbolize water (Example 2.9):

Example 2.9: Graticule background style in *style.css*

CSS

```
.gratBackground {
    fill: #D5E3FF;
}
```

Note that separating the `gratBackground` and `gratLines` blocks allows us to reorder the drawing of our graticule and spatial data if we so choose. If we wanted our graticule lines to appear on top of our other geometries, we could leave the `gratBackground` block where it is and move the `gratLines` block below the `countries` and `regions` blocks. The interpreter will add the `<path>` elements from each of these blocks in the order they appear in the script.

One final touch we will add to the map background is a frame to neaten the map (Example 2.9).

Example 2.9: Framing the map in *style.css*

CSS

```
.map {
    border: medium solid #999;
}
```

Below is our final basemap (Figure 2.6). Note that the enumeration units will be colored to create our choropleth in the next module.

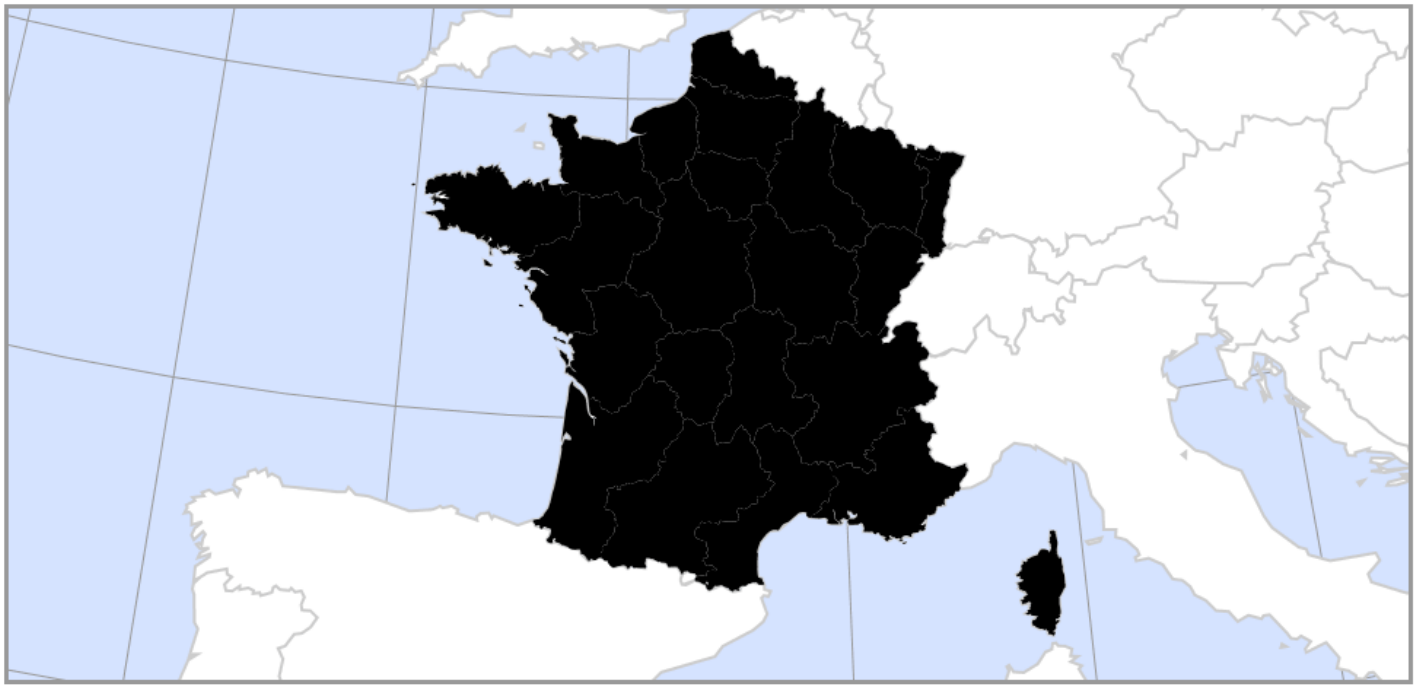

Figure 2.6: D3 map of France

You may choose to add your own stylistic touches to your overall map. Do not feel limited by the rather simple approach we have taken in this example. Consider whether your geometries warrant the kind of background reference and framing we have added here, or if they can **stand on their own (http://bost.ocks.org/mike/map/)** within your geovisualization.

Self-Check:

1. **Which of the following D3 projection parameters assigns the central meridian, centering the map projection longitudinally while keeping north "up" in the middle of the frame?**
   a. .center([0, 46.2])
   b. rotate([-2, 0])
   c. .parallels([43, 62])
   d. .scale(2500)
   e. .translate([width / 2, height / 2])
2. **True/False:** Projected spatial data cannot be drawn correctly by D3.

Module Reminder

1. Create a *d3-coordinated-viz* web directory and Git repository for your D3 lab assignment. Sync this repository with GitHub.
2. Simplify your spatial data and convert it to TopoJSON format.

3. Use *d3.queue* to load your spatial data TopoJSON files and multivariate attribute CSV file into your *main.js*script.
4. Choose a projection to use with your choropleth map and create the appropriate D3 projection generator.
5. Use your projection generator to create a path generator and draw your spatial data as geometries on the map. Add appropriate styles in *style.css*.
6. With all of the above tasks completed, commit changes to your d3-coordinated-viz web directory and sync with GitHub.