Geography 575
Lab #2: Coordinated Visualization w/ D3

Lab Objectives:

- Use the D3 library to coordinate interactions across multiple visualizations
- Learn about the GeoJSON and TopoJSON formats
- Implement *sequence* and *retrieve* for coordinated, multivariate visualization

Evaluation:

This lab is worth **50 points** toward the Lab Assignments evaluation item. A grading rubric is provided at the end of the lab to inform your work.

Schedule of Deliverables:

- **October 20-21st:** Lab #2 Assigned //collaboration begins
- **October 27-28th:** Multivariate Dataset Due //progress report to collaborator
- **November 3-4th:** Basemap Due // progress report to collaborator
- **November 10-11th:** Attribute Sequencing Due // progress report to collaborator
- **November 17-18th:** Lab #2 Due //submission deadline

Challenge Description

You have decided to compete for an ENGAGE Innovation Award, a UW-Madison program run by DoIT promoting the use of technology for research and teaching (http://engage.wisc.edu/). This specific Innovation Award cycle addresses the use of information visualization for the purpose of scientific exploration. Parameters of the Innovation Award require you to work with a domain expert to develop a visualization of complex and multivariate information in order to facilitate the generation of new insights about your collaborator's core research interests. Given your expertise in engineering successful user experiences with map-based visualizations, you plan to team up with a colleague of yours in Geography to design a highly interactive and coordinated geovisualization application, with the goal of supporting hypothesis generation and knowledge construction about your colleague's spatial research. The visualization will load enumerated information, allowing for the interactive identification, comparison, ranking, association, and delineation of multiple attributes as they vary across space. The selection of winners is based on a proof-of-concept application allowing for exploration of a sample information set that you have assembled. The proof-of-concept application should reveal new insights regarding notable outliers, anomalies, patterns, trends, correlations, and clusters; submissions will be chosen based on the potential for expanding these proof-of-concept interfaces to unlock additional geographic insights.

Editor's Notes from ENGAGE

Your visualization must include a choropleth map with *at least 15* enumeration and *at least 5* numerical variables collected for these units. The enumeration units cannot be at the same geographic location and/or cartographic scale as your Lab #1/#2 application.

Coordinated, Multivariate Visualization on the Web

Getting Started With D3

D3, or ***Data-Driven Documents***, presents a different philosophy of web mapping than the majority of technologies that currently produce web maps. Leaflet and most other web mapping libraries produce *slippy maps* based on sets of tiled raster images loaded dynamically into the browser when needed. A common complaint from cartographers about slippy maps is their virtually universal reliance on cylindrical projections, most commonly the conformal Web Mercator projection, which is highly distorted across latitude and inappropriate for data visualization at small map scales. The D3 alternative utilizes vector graphics rendered in the browser, allowing for dynamic map projection and direct feature interaction.

D3 is an open-source JavaScript library pioneered and maintained by Mike Bostock of the New York Times (http://bost.ocks.org/mike). Increasingly recognized as a leading data visualization library, D3 simplifies loading and interacting with information. It draws all graphics as client-side (in the browser) vectors using the SVG (Scalable Vector Graphics) standard. Maps created using D3 can be transformed into a multitude of projections thanks to the work of freelance developer Jason Davies and the *proj4.js* library of map projections (http://trac.osgeo.org/proj4js/).

The goal of this tutorial is to provide you with a broad introduction to using D3 for Web Cartography. The following tutorial extends two excellent online learning materials: (1) Mike Bostock's "Let's Make a Map" tutorial (http://bost.ocks.org/mike/map), and (2) developer Scott Murray's e-book "Interactive Data Visualization for the Web" (http://chimera.labs.oreilly.com/books/1230000000345); the web version is free as of this writing). Refer to these materials for additional background and guidance as you complete the tutorial.

1. Finding and Formatting Multivariate Information

The first step is assembly of an appropriate ***multivariate*** (i.e., multiple attributes enumerated over the same set of spaces) information to portray in a choropleth map. Because choropleth maps shade the color of a region according to an attribute value, your data should be numerical and tabular, with attributes (or fields) organized as separate table columns and each enumeration unit (or region) represented by a single table row. Because enumeration units typically vary in size and shape, it is important to normalize the attribute information according to a relevant variable (i.e., divide by area, population, etc.).

Although they could be combined into one file, this tutorial maintains the attribute data and geographic data (i.e., the linework) as separate files in order to demonstrate how to join data from disparate sources together using JavaScript. This is useful when drawing on data from dynamic web services. Prepare your attribute information as a *.csv* file using spreadsheet software (e.g., Microsoft Excel, Google Sheets, OpenOffice Calc). In addition to attribute columns, the file should include columns for a unique ID, the name of each enumeration unit, and a code that can be linked to the geographic dataset.

Figure 1 provides an example multivariate attribute dataset for the provinces of France. The attribute "admin1_code" will be used to link the attribute dataset to the geographic dataset in the code. This is a dummy dataset with no meaning in the real world; you should replace it with data from a phenomenon that is of interest to you. To avoid problems in your code later on, be sure the column names are logical and do not contain spaces or start with a number or special character. These headers will be used as keys to reference the data values in your code.

	A	B	C	D	E	F	G	H
1	id	name	adm1_code	varA	varB	varC	varD	varE
2	1	Alsace	FRA-2686	85	38	75	30	9
3	2	Aquitaine	FRA-2665	28	29	38	26	15
4	3	Auvergne	FRA-2670	18	59	22	60	82
5	4	Basse-Normandie	FRA-2661	35	45	31	26	14
6	5	Bourgogne	FRA-2671	12	31	15	22	28
7	6	Bretagne	FRA-2662	25	50	25	25	25
8	7	Centre	FRA-2672	88	46	56	15	12
9	8	Champagne-Ardenne	FRA-2682	52	51	46	68	75
10	9	Corse	FRA-2666	7	12	18	11	9
11	10	Franche-Comte	FRA-2685	23	18	16	24	26
12	11	Haute-Normandie	FRA-2673	32	28	29	25	22
13	12	Ile de France	FRA-2680	8	15	22	25	29
14	13	Languedoc-Roussillon	FRA-2668	82	74	72	10	85
15	14	Limousin	FRA-2681	9	16	14	23	45
16	15	Lorraine	FRA-2687	33	45	68	96	102
17	16	Midi-Pyrenees	FRA-2669	15	38	85	96	82
18	17	Nord Pas de Calais	FRA-2683	12	10	9	4	74
19	18	Pays de la Loire	FRA-2664	42	18	28	30	22
20	19	Picardie	FRA-2684	9	15	15	8	29
21	20	Poitou-Charentes	FRA-2663	38	31	28	32	85
22	21	Provence-Alpes Cote D'Azur	FRA-2667	25	100	88	74	45
23	22	Rhone-Alpes	FRA-1265	25	46	66	85	45
24								

Figure 1: An Example Multivariate Attribute Dataset

Next, prepare the geographic dataset in GIS software. For this tutorial, you will convert your geography to ***TopoJSON*** format (https://github.com/mbostock/topojson/wiki). TopoJSON is similar to GeoJSON, introduced in the Leaflet tutorial, but significantly reduces the file size and stores ***topology,*** or the spatial relationships of features. Both are variants on ***JSON***, which stands for JavaScript Object Notation. TopoJSON structures each map feature as a series of ***arcs,*** or lines connecting sets of nodes, with the node sets stored in a separate object that indexes the arcs. This format greatly reduces the data volume and improves rendering efficiency by storing each arc only once, rather than duplicating arcs along shared edges. The lightweight TopoJSON library, available from http://github.com/mbostock/topojson, is required to translate the TopoJSON format for use by D3 or any other web mapping library that can utilize GeoJSON files.

The sample geography dataset used in this tutorial (*europe.topojson*, included in the tutorial source code) was prepared using two shapefiles downloaded from the Natural Earth website (http://www.naturalearthdata.com): one with the nations of Europe as a whole, and one with the provinces of France. A number of technologies may be used to convert data to TopoJSON format,

including the TopoJSON command line tool (available from the wiki cited above), MapShaper (http://mapshaper.com), GeoJSON.io (http://geojson.io), and Shape Escape (http://www.shpescape.com). The Natural Earth shapefiles for this tutorial were combined into a single TopoJSON file using Shape Escape.

The object structure of the TopoJSON format includes an outer-level object with the key `objects` that is not included in the GeoJSON specification. For this tutorial, you will need to ensure that each of the `GeometryCollection` objects (analogous to the `FeatureCollection` level of a GeoJSON, or layers in a shapefile) has a key that is a logical name for the feature layer represented by that `GeometryCollection`, as shown in **Figure 2**. If you use Shape Escape, it will be necessary to open the TopoJSON file in a plain text editor and manually change these layer names.

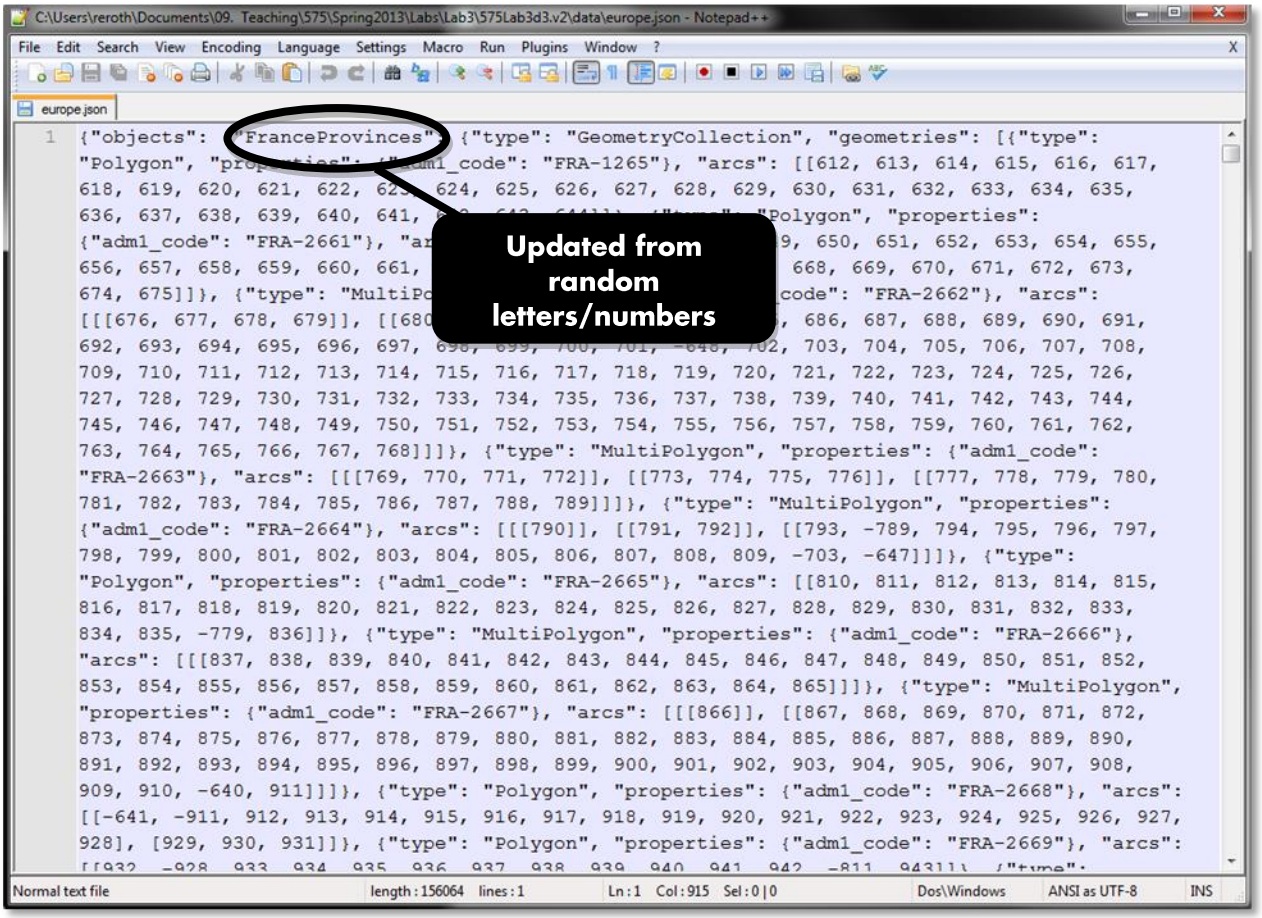

Figure 2. Manually rename the `objects` in your TopoJSON file for reference in your code.

2. Preparing Your Directory Structure and Boilerplate HTML

With your *.csv* and TopoJSON (*.json*) files prepared, it is now time to start building your map! Create a directory that includes folders named "data", "css", "js", and "lib". Because you will be loading data from the local directory asynchronously (after the first code has executed), you should place your website directory within your development server's root so that it can be accessed via http://localhost/. In the website directory, create three new files named *index.html* (root level),

style.css (*css* folder), and *main.js* (*js* folder). Copy your newly created *.csv* and *.json* files into the *data* folder. Add the boilerplate HTML code provided in **Code Bank 1** to the *index.html file.*

After configuring your directory, acquire three *.js* files from Bostock's GitHub account: (1) *d3.v3.js* (https://github.com/mbostock/d3) containing the D3 visualization library, (2) *topojson.v1.min.js* for parsing your TopoJSON file (https://github.com/mbostock/topojson/), and (3) *queue.js* (https://github.com/mbostock/queue), which will help with asynchronously loading the data. Save these files to your *lib* folder and link to them in *index.html* (**CB1: 11-13**).

```
1       <!DOCTYPE HTML>
2       <html>
3               <head>
4                       <meta charset="utf-8">
5                       <title>My Coordinated Visualization</title>
6
7                       <!--main stylesheet-->
8                       <link rel="stylesheet" href="css/style.css" />
9               </head>
10              <body>
11                      <!--libraries-->
12                      <script src="lib/d3.v3.js"></script>
13                      <script src="lib/topojson.v1.min.js"></script>
14                      <script src="lib/queue.js"></script>
15
16                      <!--link to main JavaScript file-->
17                      <script src="js/main.js"></script>
18              </body>
19      </html>
```

Code Bank 1: Basic HTML5 Boiler Plate (in: *index.html*).

3. Loading Your *.json* into the Browser

The next step is loading the geographic information assembled in *europe.json*. One of the excellent code classes provided by D3 includes several methods for loading various data formats using ***AJAX*** (Asynchronous JavaScript and XML) and parsing the contained information into JavaScript arrays. The loaders used here are *d3.csv* and *d3.json.* These methods work even better when used in conjunction with the *queue.js* plug-in, as explained below.

Because the data is loaded ***asynchronously***—i.e., separately from the rest of the script so the browser does not have to wait for that data to load before displaying the web page—all code that manipulates the asynchronous data must be contained within a ***callback*** function that is triggered only after the data is loaded. A callback function can be specified for each D3 data loader. The downside of this is that each loader requires a separate callback, requiring you to nest loaders and callbacks in order to manipulate data from multiple files.

Code Bank 2 provides the logic needed to initialize the webpage and print the *europe.json* file to the console. The `queue()` method (**CB3: 12-15**) , which accesses the *queue.js* plug-in, allows data from multiple files to be loaded in parallel rather than in series, thus speeding up the process, and allows you to specify a single callback function for all data sources. Load the *index.html* page in your browser; you now will see the TopoJSON loaded to the DOM (**Figure 3**).

```
1       //begin script when window loads
2       window.onload = initialize();
3
4       //the first function called once the html is loaded
5       function initialize(){
6               setMap();
7       };
8
9       //set choropleth map parameters
10      function setMap(){
11              //use queue.js to parallelize asynchronous data loading
12              queue()
13                      .defer(d3.csv, "data/unitsData.csv") //load attributes from csv
14                      .defer(d3.json, "data/europe.json") //load geometry from topojson
15                      .await(callback); //trigger callback function once data is loaded
16
17              function callback(error, csvData, europe){
18                      console.log(europe);
19              };
20      }
```

Code Bank 2: Loading data files and printing *europe* data to the console (in: *main.js*).

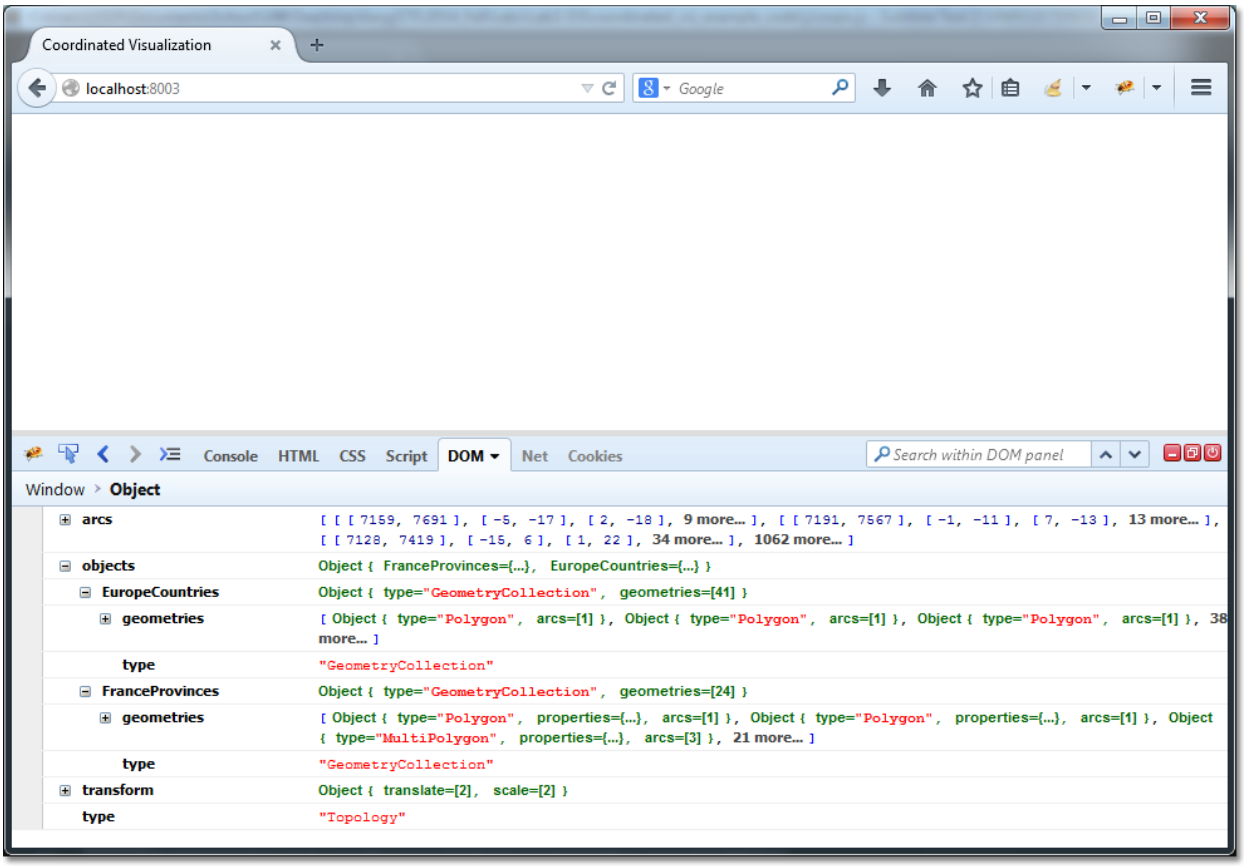

Figure 3. Printing the TopoJSON to the DOM. The object names should match the names given in **Figure 2**, which in turn should match the original shapefiles.

4. Drawing Your Basemap

Now that your information is loading properly, it is time to draw your basemap. The first step in creating a map or any other visualization using the D3 library is creation of an HTML element in which to draw the map. You need not create any elements in *index.html;* instead you will use D3 to create a blank `<svg>` element for the map and populate its content (first review the SVG specification at http://www.w3.org/TR/SVG/).

Start creation of the choropleth map by drawing its basemap, or geographic context. The basemap makes use of the geometry included in *EuropeCountries.shp* and converted to the `EuropeCountries` object in your TopoJSON. All graphics associated with the choropleth view are drawn within the `setMap()` function, created in **Code Bank 2** and extended in **Code Bank 3**. First, set the size of the map view in pixels (**CB3: 3-5**). Note the absence of `px` after each number, as used in stylesheets; D3 takes integers for dimension values and translates them to pixels.

```
1     function setMap(){
2
3         //map frame dimensions
4         var width = 960;
5         var height = 460;
6
7         //create a new svg element with the above dimensions
8         var map = d3.select("body")
9             .append("svg")
10            .attr("width", width)
11            .attr("height", height);
12
13        //create Europe Albers equal area conic projection, centered on France
14        var projection = d3.geo.albers()
15            .center([-8, 46.2])
16            .rotate([-10, 0])
17            .parallels([43, 62])
18            .scale(2500)
19            .translate([width / 2, height / 2]);
20
21        //create svg path generator using the projection
22        var path = d3.geo.path()
23            .projection(projection);
24
25        //use queue.js to parallelize asynchronous data loading
26        queue()
27            .defer(d3.csv, "data/unitsData.csv") //load attributes from csv
28            .defer(d3.json, "data/europe.json") //load geometry from topojson
29            .await(callback); //trigger callback function once data is loaded
30
31        function callback(error, csvData, europe){
32            //add Europe countries geometry to map
33            var countries = map.append("path") //create SVG path element
34                .datum(topojson.feature(europe,
                                    europe.objects.EuropeCountries))
35                .attr("class", "countries") //class name for styling
36                .attr("d", path); //project data as geometry in svg
37        };
38    }
```

Code Bank 3: Extending `setMap()` to Draw the Basemap.

Next, create an `<svg>` element to contain the choropleth map using the `d3.select()` method (**CB3: 7-11**). A D3 ***selection*** creates an array with one or more DOM elements to be operated on by subsequent methods. The string parameter passed to `.select()` references the DOM element to be selected and is therefore called the ***selector***. The statement `d3.select("body")` is essentially the same as `$("body")` in jQuery; it passes the selector `"body"` and returns the first matching element in the DOM. Like jQuery, D3 uses dot syntax to string together methods, an approach known as ***method chaining***. This code selects the `<body>` element of the DOM and adds an `<svg>` element, then sets the size to the values already stored in the `width` and `height` variables. This new `<svg>` element essentially is a container that holds the map geometry. Another method, `selectAll()`, can be used to select *every* matching element in the DOM as well as create new elements from data. It will be evoked later in the tutorial.

After creating the `<svg>` container, you then need to indicate how the geographic coordinates should be projected onto the two-dimensional plane (the computer screen). As stated in the introduction, one of the exciting things about D3 for cartographers is its support for an extensive and growing library of map projections. The list of projections currently supported by D3, either natively or through the extended projections plug-in, is available at: https://github.com/mbostock/d3/wiki/Geo-Projections. Choose an equal-area projection, given the choropleth mapping context.

The following example applies the Albers Equal Area Conic projection using `d3.geo.albers()`, with a centering on France (**CB3: 13-19**); this projection is native to *d3.v3.js*. The projection parameters following the method call apply mathematical transformations to the default Albers projection:

- `.center` recenters the map at a given `[lon, lat]` coordinate;
- `.rotate` rotates the globe counter-clockwise (from the North Pole) given angles of `[lon, lat, roll]` away from the geographic center;
- `.parallels` sets the standard parallels of the projection, given as `[lat1, lat2]`;
- `.scale` is the scale of the map, set using an arbitrary scale factor;
- `.translate` adjusts the pixel coordinates of the map's center, and always should be set as half the width and height to keep the map's center in the center of the SVG area.

Next, you need to project your TopoJSON according to these projection parameters. D3 uses a "geo path" SVG element to render the geometry included in a *.json* as SVG. The `d3.geo.path()` method creates a new `path` generator with a default projection of Albers USA. This logic may run counter to previous experience with JavaScript; if D3 worked like Leaflet, you might expect that calling `d3.geo.path()` will return an object or an array. Instead, `d3.geo.path()` calls a D3 ***generator function*** that creates a new function (the ***generator***) based on the parameters you send it. You then can store this generator function as a variable, and access the variable like you would call a function, passing it parameters to manipulate. Note that the `d3.geo.path()` method requires that you specify the previously created projection. Each time the `path` generator is used to create a new SVG element (i.e., a new graphical layer in the map), the SVG graphics will be drawn using the projection indicated in the call to the `d3.geo.path()` generator function. Hopefully, the idea and usage of generator functions will become clearer as you proceed through the tutorial.

First, make use of the `d3.geo.path()` generator function to define a `path` generator that creates projected SVG paths from the TopoJSON geometry based on your map projection (**CB3: 21-23**).

Then, make use of this `path` generator through the `append()` method to add an SVG `<path>` element containing the geometry derived from your TopoJSON, projected according to the generator definition (**CB3: 32-36**); note that this code should be part of the callback function, replacing the console.log statement. The first line adds the `<path>` element to the DOM and to the `map` selection and assigns the new selection to a variable `countries` (**CB3: 33**). The second line specifies the `datum()` that will be attached to the `countries` selection (**CB3: 34**). In D3 terms, a ***datum*** is a unified chunk of information that can be expressed in SVG form (as in the singular form of *data,* not a geometric model of Earth's surface).

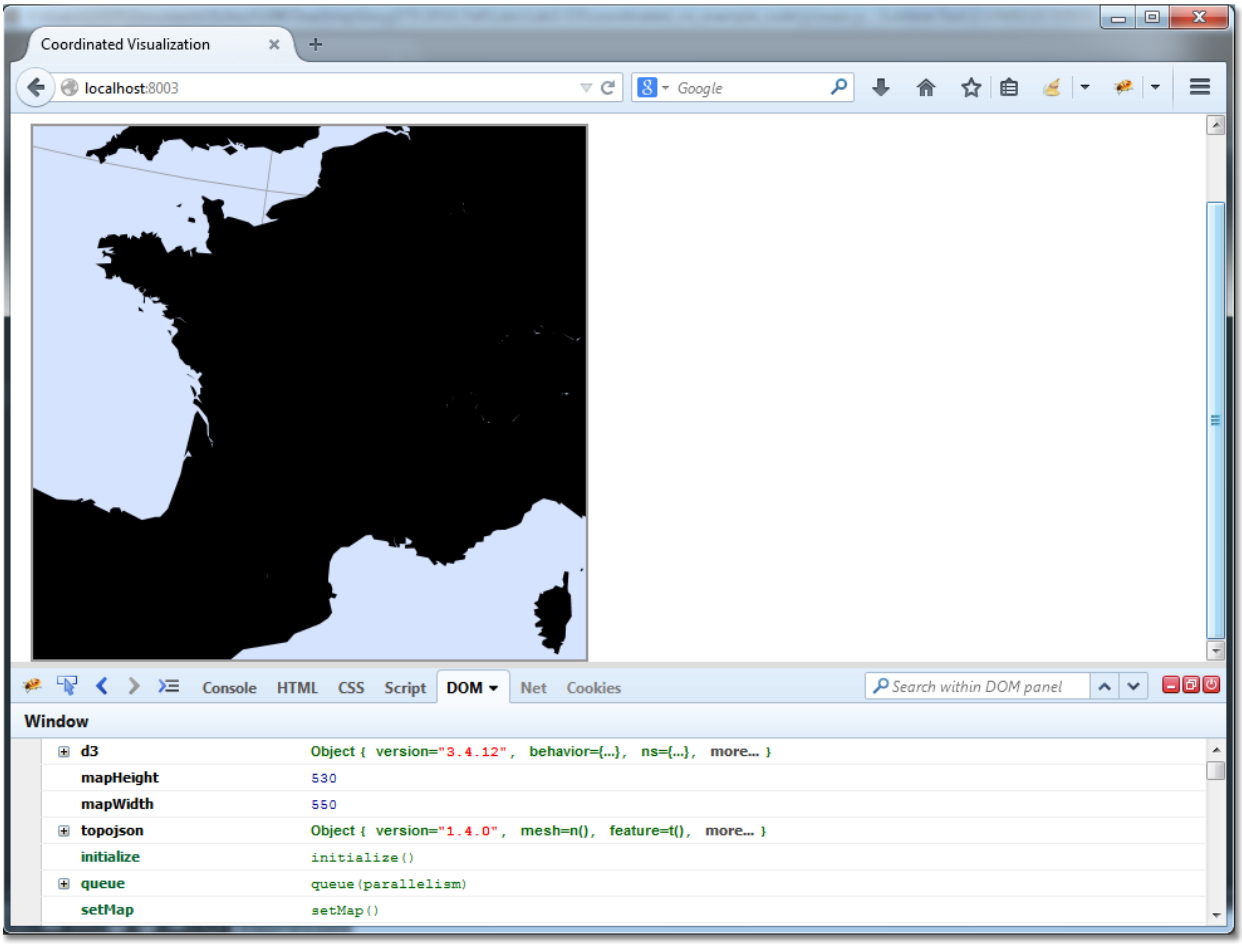

Figure 4. Drawing the Basemap.

At this point, it is acceptable to treat the polygons in the `EuropeCounties` JavaScript object altogether, as this is the background context for the choropleth map and will not be interactive. This is why the `append()` method is used, rather than first calling the `selectAll()`, `data()`, and `enter()` methods (described below). The `datum()` method expects a JSON or GeoJSON; to use the newer TopoJSON format, access the `topojson.feature()` method from *topojson.js*, indicating the object (`EuropeCounties`) in the TopoJSON you want translated. The third line assigns the `countries` `<path>` element the class name `countries` so that it can be styled in *styles.css* (**CB3: 35**). In the fourth line, the `d` attribute contains a string of information that describes the `<path>` (see: https://developer.mozilla.org/en-US/docs/SVG/Attribute/d) (**CB3: 36**). It is for

this purpose that the `path` generator is so useful: it projects the `EuropeCounties` geometry and translates it into an SVG `<path>` description string.

Altogether, the revisions in **Code Bank 3** result in four D3 ***blocks*** of chained methods connected by dot syntax to minimize the file size (**8-11**; **14-19**; **22-23**; **32-36**). It is important that you do not place a semicolon between lines of a block, as this interrupts the block and results in a syntax error. To maximize clarity, the following instructions designate a variable for each block that creates at least one new element, with the same variable name as the class attribute designated for that element, even if that variable is not accessed again. Now refresh your browser and you should see a map (**Figure 4**).

5. Styling Your Basemap

With the basemap geometry drawing in the browser, it is now time to style the basemap. Style rules can be applied to the SVG element containing the projected map using the `countries` class reference (**CB3: 35**). Return to *style.css* and add the basic style rules provided in **Code Bank 4**. These styles add default gray outlines to the `countries` as a starting point; continue to improve the applied basemap styles as you progress through the tutorial.

```
1       .countries {
2               fill: #fff;
3               stroke: #ccc;
4               stroke-width: 2px;
5       }
```

Code Bank 4: Basic Styles for the `countries` class (in *style.css*).

A nice cartographic function supported by D3 is the ability to add graticule lines to any map (**Code Bank 5**). Graticule methods are included in the D3 Geo Paths documentation (https://github.com/mbostock/d3/wiki/Geo-Paths), and an example is available at: http://bl.ocks.org/mbostock/3734308.

To add the graticule to your basemap, begin by creating a generator called `graticule` (**CB5: 1-3**). Where you place this code matters, as you are conceptually building your visual hierarchy from the bottom-up in the map as you add new code from the top-down in the `setMap()` function. The `graticule` generator should be placed after creating the `path` generator, but before loading and processing the TopoJSON with `queue()`; this order will place the countries above the graticule.

Next, use the `path` generator to add two SVG elements named `gratBackground` (i.e., the water) and `gratLines` to the map. First, add `gratBackground` using the `append()` method and configure its attributes (**CB5: 5-9**). Then, add `gratLines` to the `map` using `selectAll()`, `data()`, and `enter()` and configure its attributes (**CB5: 11-17**). The sequence of these three methods is used to create multiple new `<path>` elements at once, and thus to draw each desired graticule line individually. This is required by D3 for graticule lines, but also is useful when individual features are styled differently or are interactive.

Consider carefully the code provided in **Code Bank 5**, particularly the final block. It might appear as though D3 warped the space-time continuum to select DOM elements before they were created.

Really, D3 just 'sets the stage' for them. The `selectAll()` method creates an ***empty selection***, or a blank array into which will be placed one element for each graticule line (**CB5: 12**). The `data()` method operates like `datum()`, but creates `undefined` placeholders in the selection array for future elements associated with each datum (each value or object within the overall data array) (**CB5: 13**). The `enter()` method adds each datum to its placeholder in the selection array, changing the placeholders from `undefined` to objects, each of which has a `__data__` property that holds the associated datum (**CB5: 14**). Every method placed below `enter()` will be executed once for each item in the array; you can think of this as a kind of "for" loop. The `append()` method adds a new `<path>` element for each object in the selection, binding the datum to that element (**CB5: 15**). The first `attr()` call assigns each `<path>` element a class for styling purposes (**CB5: 16**); note that you must assign this class, as the only purpose of passing `".gratLines"` to `selectAll()` is to select elements that do not exist yet in the DOM. The second `attr()` call projects each datum through the `path` generator into the `d` attribute, just as `gratBackground` and `countries` are projected.

```
1       //create graticule generator
2       var graticule = d3.geo.graticule()
3               .step([10, 10]); //place graticule lines every 10 degrees
4
5       //create graticule background
6       var gratBackground = map.append("path")
7               .datum(graticule.outline) //bind graticule background
8               .attr("class", "gratBackground") //assign class for styling
9               .attr("d", path) //project graticule
10
11      //create graticule lines
12      var gratLines = map.selectAll(".gratLines") //select graticule elements
13              .data(graticule.lines) //bind graticule lines to each element
14              .enter() //create an element for each datum
15              .append("path") //append each element to the svg as a path element
16              .attr("class", "gratLines") //assign class for styling
17              .attr("d", path); //project graticule lines
```

Code Bank 5: Adding a Graticule to `setMap()` (in: *main.js*).

Finally, style the `gratBackground` and `gratLines` in *style.css* using the class names `"gratBackground"` and `"gratLines"` (**Code Bank 6**); you are encouraged to tweak these styles as your design evolves. Refresh your *index.html* page in your browser to view your basemap (**Figure 5**).

```
1       .gratBackground {
2               fill: #D5E3FF;
3       }
4
5       .gratLines {
6               fill: none;
7               stroke: #999;
8               stroke-width: 1px;
9       }
```

Code Bank 6: Styling the Graticule (in *style.css*).

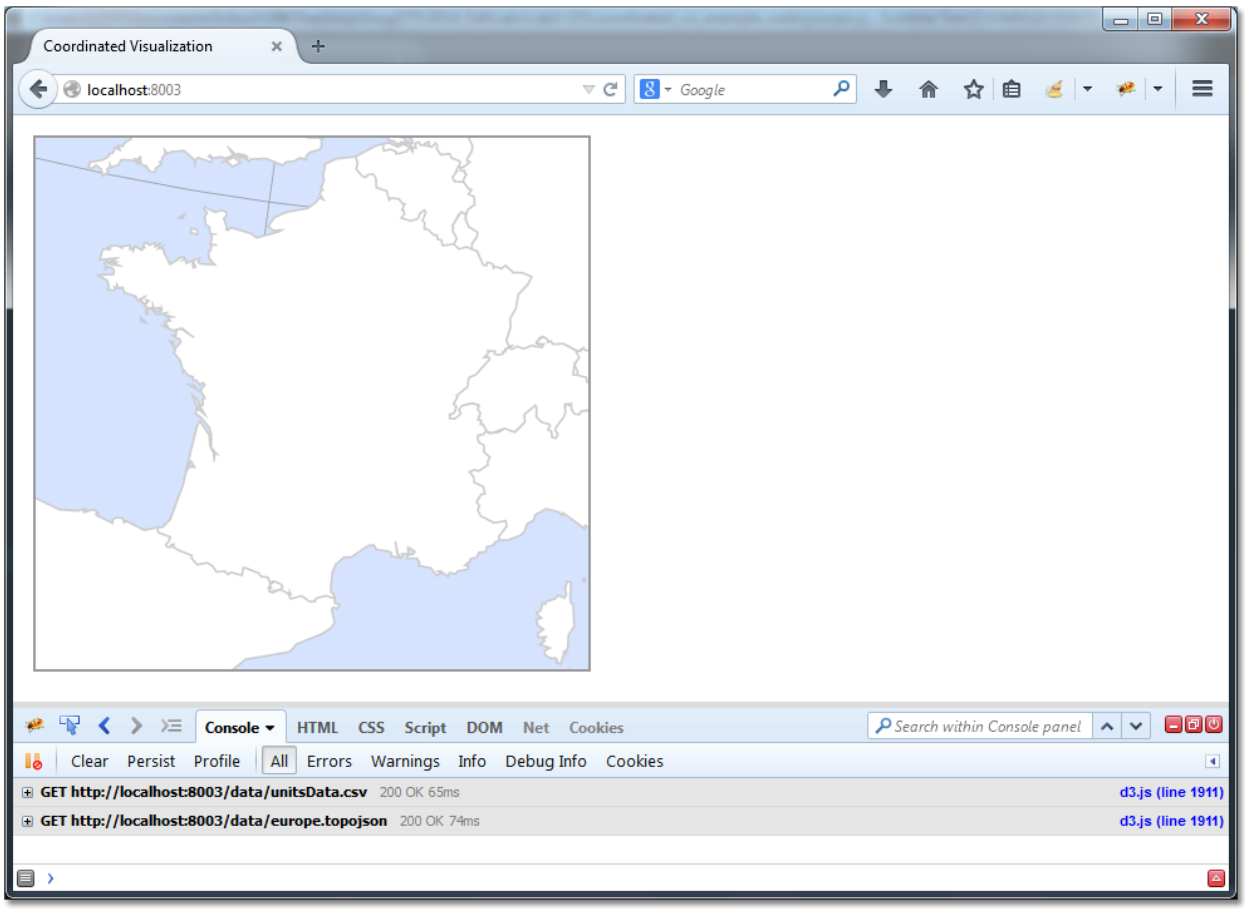

Figure 5. Styling the Basemap.

6. Drawing Your Choropleth Map

With the basemap context in place, you are ready to draw your choropleth map. Again, the choropleth maps uses the geometry included in *FranceProvinces.shp* and converted to the `FranceProvinces` object in your TopoJSON. The `FranceProvinces` object could be added through the `path` generator using using `datum()`, as with the `EuropeCountries` above. Unlike the basemap, however, each province is unique in both representation and interaction. You need to add each province separately in order to set different properties and attach different event listeners to each individual province. Therefore, you need to use the `selectAll()`, `data()`, and `enter()` methods—much like you did to draw each graticule lines—rather than the simpler `append()` and `datum()` methods—like you did for drawing the basemap countries and graticule background (**Code Bank 7**). A new selection named `provinces` is created using the `selectAll()` method and each province is added to the map by the `path` generator (**CB7: 10-18**). It is important to note that **Code Bank 7** must be added within the `callback` function, as the TopoJSON must first be processed before adding the `provinces` element.

Reload your *index.html* file in your browser; you now should see your enumeration units plotted atop the basemap and graticule with a default black fill (**Figure 6**).

```
1       //retrieve and process europe json file
2       function callback(error, csvData, europe){
3
4               //add Europe countries geometry to map
5               var countries = map.append("path") //create SVG path element
6                       .datum(topojson.feature(europe,
                                        europe.objects.EuropeCountries))
7                               .attr("class", "countries") //class name for styling
8                               .attr("d", path); //project data as geometry in svg
9
10              //add provinces to map as enumeration units colored by data
11              var provinces = map.selectAll(".provinces")
12                      .data(topojson.feature(europe,
                                        europe.objects.FranceProvinces).features)
13                      .enter() //create elements
14                      .append("g") //give each province its own g element
15                      .attr("class", "provinces") //assign class for additional styling
16                      .append("path")
17                      .attr("class", function(d) { return d.properties.adm1_code })
18                      .attr("d", path) //project data as geometry in svg
19
20      };
```

Code Bank 7: Drawing the Choropleth Map in `setMap()` (in: *main.js*).

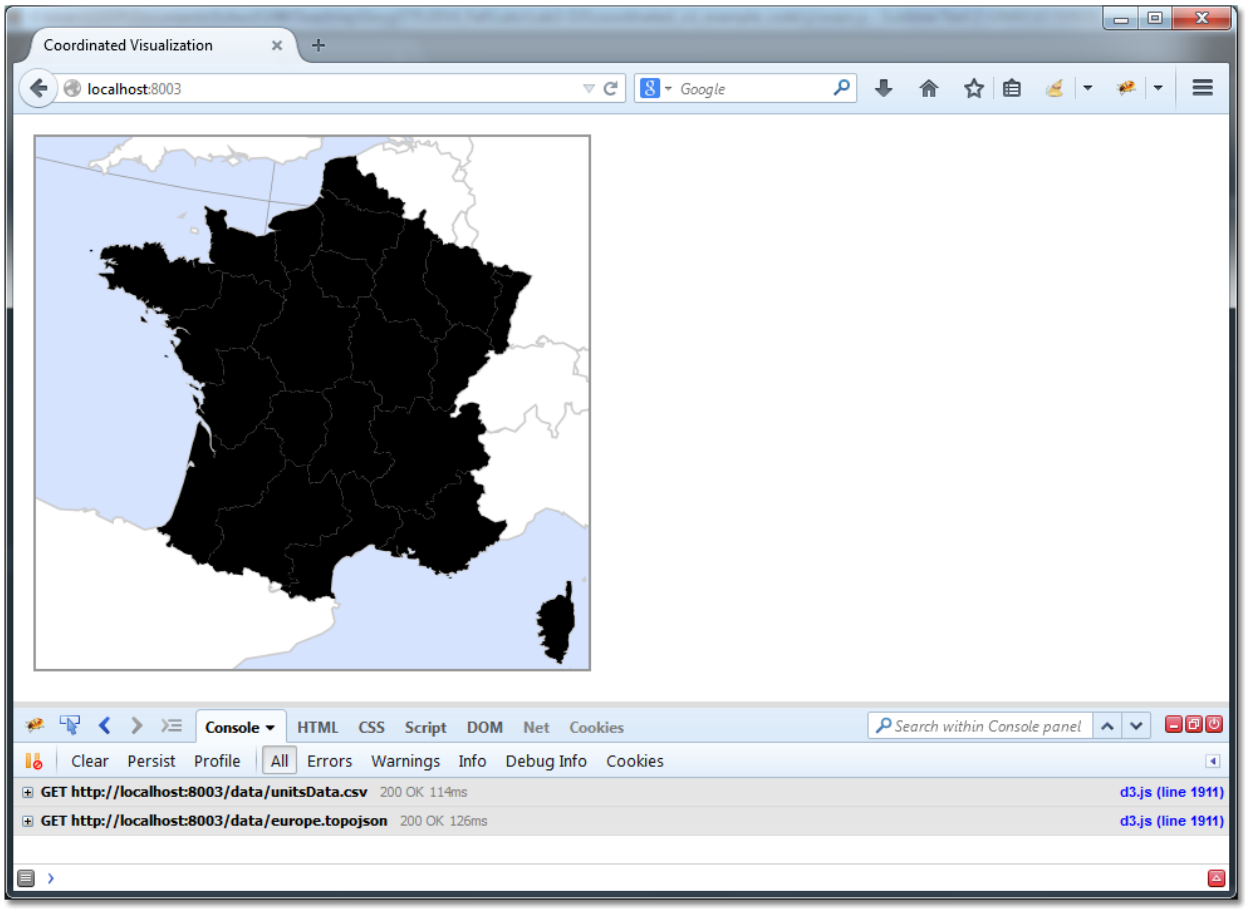

Figure 6. Drawing the Enumeration Units.

7. Relating Your *.csv* and *.json* Information

You now are ready to load the *.csv* file containing your multivariate information so that you can color the enumeration units according to their unique attribute values. **Code Bank 8** makes use of a file named *unitsData.csv*. Again note that this file includes a column with the `adm1_code` header for each province (**Figure 1**), which can be used to join each province's multivariate information in the *.csv* file to its geographic information in the *.json*.

Code Bank 8 includes part of **Code Bank 2**, which shows the use of the `d3.csv` method with `queue()` to load and parse *unitsData.csv*. The `d3.csv` method parses each row into an object using the column headings as ***keys,*** while `queue().await(callback)` passes the object to the callback function (**CB8: 4**). In order to attach the multivariate information from the *unitsData.csv* to the geographic information in *europe.json*, both sets of data must be accessed from inside the callback function, which must in turn be fully contained in `setMap()` to make use of the generator functions previously created.

In **Code Bank 8**, a set of nested loops is used to attach the multivariate information contained in `csvData` to the `FranceProvinces` topojson object as properties of each topojson `geometry` (feature). First, an array is created containing the attribute names for the attributes to be transferred, and the desired topojson geometries array is assigned to a variable for neatness (**CB8: 8-9**). An outer loop then loops through each of the province objects in the `csvData` array, assigning each object to the variable `csvProvince` and assigning the province's `adm1_code` to the variable `csvAdm1` (**CB8: 11-14**). An inner loop then cycles through each topojson `geometry` object, testing whether that object's `adm1_code` matches the `adm1_code` from the `csvData` province (**CB8: 16-20**). If the province codes match, a final loop runs through each `key` in the `keyArray` and assigns the corresponding key/value pair from the `csvData` province object to the `properties` object of the topojson `geometry` (**CB8: 22-27**). Also within the `if` statement, the topojson province is assigned the name of the `csvProvince` (**CB8: 29**). Once the right match has been found and attribute values transferred, the `jsonProvs` loop can be broken to save on processing time (**CB8: 30**). If this loop structure remains unclear, it is recommended that you add `console.log` statements line-by-line to inspect how the *.csv* and *.json* contents are being manipulated and combined through the nested `for` loops.

```
1       queue()
2               .defer(d3.csv, "data/unitsData.csv") //load attributes data from csv
3               .defer(d3.json, "data/europe.json") //load geometry from europe topojson
4               .await(callback);
5
6       function callback(error, csvData, europe){
7               //variables for csv to json data transfer
8               var keyArray = ["varA","varB","varC","varD","varE"];
9               var jsonProvs = europe.objects.FranceProvinces.geometries;
10
11              //loop through csv to assign each csv values to json province
12              for (var i=0; i<csvData.length; i++) {
13                      var csvProvince = csvData[i]; //the current province
14                      var csvAdm1 = csvProvince.adm1_code; //adm1 code
15
16                      //loop through json provinces to find right province
17                      for (var a=0; a<jsonProvs.length; a++){
18
```

```
                        //where adm1 codes match, attach csv to json object
                        if (jsonProvs[a].properties.adm1_code == csvAdm1){

                                // assign all five key/value pairs
                                for (var key in keyArray){
                                        var attr = keyArray[key];
                                        var val = parseFloat(csvProvince[attr]);
                                        jsonProvs[a].properties[attr] = val;
                                };

                        jsonProvs[a].properties.name = csvProvince.name; //set prop
                        break; //stop looking through the json provinces
                        };
                };
        };

        //add Europe countries geometry to map
        var countries = map.append("path") //create SVG path element
                .datum(topojson.feature(europe,europe.objects.EuropeCountries))
                .attr("class", "countries") //assign class for styling countries
                .attr("d", path); //project data as geometry in svg

        //add provinces to map as enumeration units colored by data
        var provinces = map.selectAll(".provinces")
                .data(topojson.feature(europe,
                                europe.objects.FranceProvinces).features)
                .enter() //create elements
                .append("g") //give each province its own g element
                .attr("class", "provinces") //assign class for additional styling
                .append("path")
                .attr("class", function(d) { return d.properties.adm1_code })
                .attr("d", path) //project data as geometry in svg
};
```

Code Bank 8: Relating Your *.csv* and *.json* Information within `setMap()` (in: *main.js*).

8. Styling Your Choropleth Map

Now that the multivariate information in the *unitsData.csv* file is attached to the `FranceProvinces` topojson object, you can color each province according to its unique attribute value. The example *csvData.csv* file contains five variables using the column headers "varA" through "varE" (**Figure 1**). Before implementing the choropleth styling solution, you first need to implement a method for determining which of the five variables should be represented in the choropleth map.

First, move the `keyArray` created within the callback in **Code Bank 8** to the top of main.js to make it a global variable (**CB9: 2**). Since the keys contained by `keyArray` are hard-coded strings, they do not actually need to be inside of the callback. Be sure to move this variable from within the `setMap()` function to the top of the *main.js* document, rather than duplicating it in both places in your code. Declare a second global variable, `expressed`, which indicates which of the keys in the `keyArray` is currently in use for coloring the choropleth map. Set the default index to `0`, or the first attribute in the *.csv* file. This index value can be changed later on to sequence through the different attributes.

```
1       //global variables
2       var keyArray = ["varA","varB","varC","varD","varE"]; //array of property keys
3       var expressed = keyArray[0]; //initial attribute
```

Code Bank 9: Global Variables for Setting the Choropleth Variable (in: *main.js*).

```
1       function colorScale(csvData){
2
3               //create quantile classes with color scale
4               var color = d3.scale.quantile() //designate quantile scale generator
5                       .range([
6                               "#D4B9DA",
7                               "#C994C7",
8                               "#DF65B0",
9                               "#DD1C77",
10                              "#980043"
11                      ]);
12
13              //build array of all currently expressed values for input domain
14              var domainArray = [];
15              for (var i in csvData){
16                      domainArray.push(Number(csvData[i][expressed]));
17              };
18
19              //pass array of expressed values as domain
20              color.domain(domainArray);
21
22              return color; //return the color scale generator
23      };
24
25      function choropleth(d, recolorMap){
26
27              //get data value
28              var value = d.properties[expressed];
29              //if value exists, assign it a color; otherwise assign gray
30              if (value) {
31                      return recolorMap(value);
32              } else {
33                      return "#ccc";
34              };
35      };
```

Code Bank 10: Functions for Styling the Choropleth Map (in: *main.js*).

Next, you need to add two new functions providing the logic for styling the choropleth map (**Code Bank 10**): (1) `colorScale()` and (2) `choropleth()`. These functions are external to the `setMap()` function. The `colorScale()` function (**CB10: 1-23**) provides the logic for setting the class breaks using a ***quantile*** classification, which divides a variable into a discrete number of classes with each class containing approximately the same number of items. Quantile classification is supported natively by D3 through the `d3.scale.quantile()` generator function. Importantly, the `colorScale()` function takes the `csvData` array from the callback function as a parameter (**CB10: 1**), demonstrating the value of the AJAX solution. Because of this, the call to `colorScale()` must be added within the callback function, but before the `provinces` block where the resulting scale will be used (**CB11: 2**). The `colorScale()` function first creates a `d3.scale.quantile()` generator named `color` and indicates the color scheme for the choropleth using the `range()` method, which specifies the scale's output (**CB10: 3-11**); a five-class, purple color scheme from ColorBrewer is used here. The `d3.scale.quantile()` method also requires an array of input values, specified using the `domain()` method. To determine the quantile class breaks properly, the domain array must include all of the attribute values for the currently expressed attribute (note: an equal-interval classification can be created by instead passing a two-value array to `domain()` with just the minimum and maximum values of the

expressed attribute). A loop through the `csvData` array is used to push the expressed attribute value for each enumeration unit into a single array, which is then passed to the `domain()` method (**CB10: 13-20**). The `color` generator is returned to `setMap()` and stored locally in `recolorMap` (**CB10: 19** -> **CB11: 2**).

```
1       function callback(error, csvData, europe){
2               var recolorMap = colorScale(csvData);
3
4               //CODE BANK 8 SUPPRESSED FOR SPACE
5
6               var provinces = map.selectAll(".provinces")
7                       .data(topojson.feature(europe,
                                europe.objects.FranceProvinces).features)
8                       .enter() //create elements
9                       .append("g") //give each province its own g element
10                      .attr("class", "provinces") //assign class for additional styling
11                      .append("path")
12                      .attr("class", function(d) { return d.properties.adm1_code })
13                      .attr("d", path) //project data as geometry in svg
14                      .style("fill", function(d) { //color enumeration units
15                              return choropleth(d, recolorMap);
16                      });
17      };
```

Code Bank 11: Styling the Choropleth Map in `setMap()` (in: *main.js*).

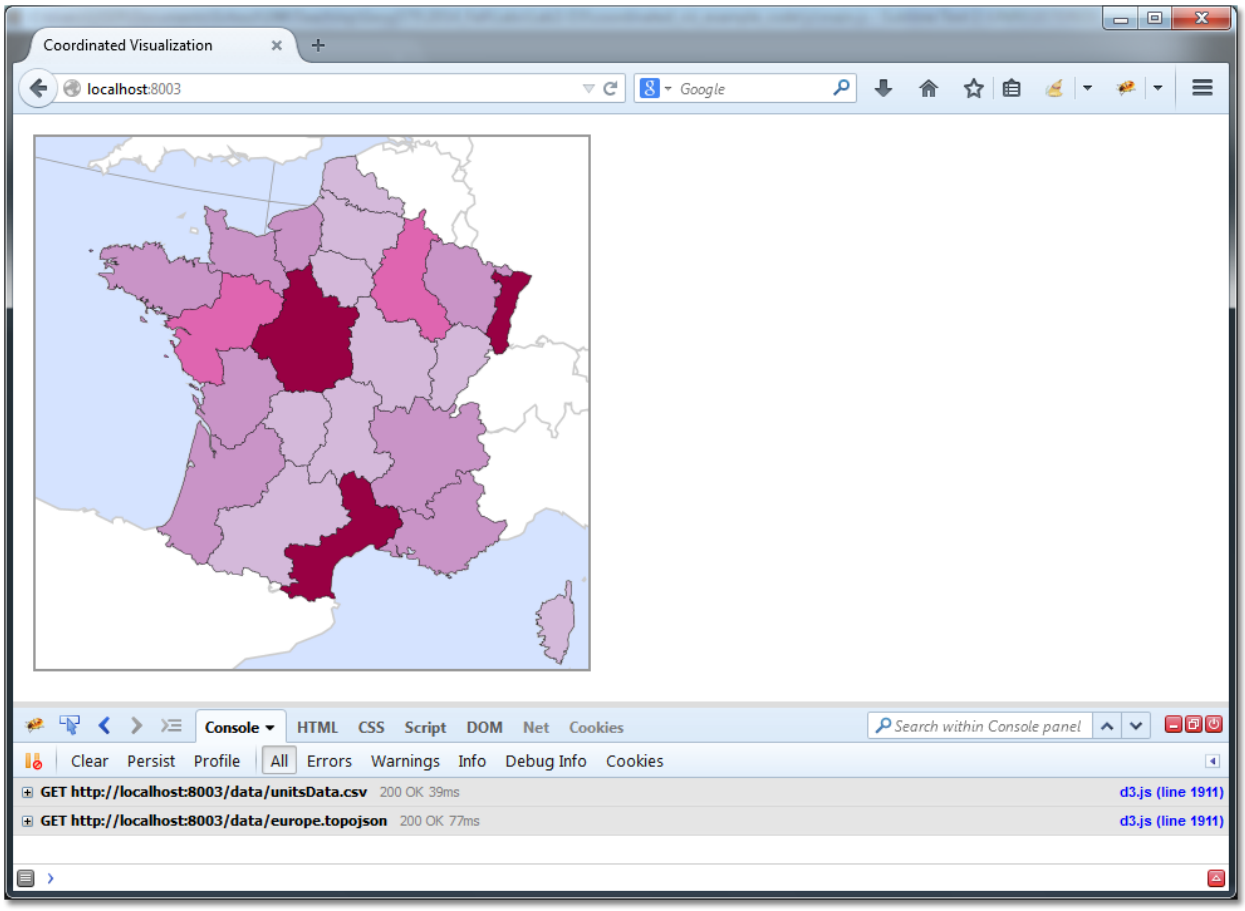

Figure 7. Styling the Choropleth Map.

The `choropleth()` function then colors the enumeration units according to this quantile classification. The `choropleth()` function is called on the `style()` method in the `provinces` block, within the callback function in `setMap()` (**CB11: 14-16**). The choropleth function takes two parameters: (1) a `datum` from the *europe.json* `FrancePovinces` object associated with a given province (passed through the selection) and (2) the `color` generator, stored locally in the `recolorMap` variable. (**CB11: 15**). The choropleth function identifies the attribute value of the province under investigation (**CB10: 28**) and then checks if a value is valid (**CB10: 30**). If the value exists, the class color associated with that value's quantile is returned (**CB10: 31**); if it does not, a default grey is returned (**CB10: 33**).

Refresh *index.html* in your browser. Bingo! You now have a choropleth map (**Figure 7**).

9. Adding a Coordinated Data Visualization

After implementing the choropleth map, your next task is to create a linked information visualization that provides users with a richer, exploratory experience. D3 enables the drawing of limitless visual isomorphs that can be coordinated with your map. A variety of common visualizations—along with D3 sample code—is provided in the D3 Gallery. This tutorial makes use of a simple histogram or bar chart to represent each of your enumeration units in attribute space, with the enumeration units ordered from least to greatest value in the currently mapped attribute. However, you may experiment with implementing a different visual isomorph; in this case, make use of the Gallery examples and the D3 API Reference to guide you. For a more extensive look at bar charts and scatterplots, see Chapter 6 of Scott Murray's book.

The first step is to create a second SVG container to hold the bar chart. Define a new function called `setChart()`, which should accept two parameters: the `csvData` array and the color generator held in `colorize` (**CB12: 6**). Call this function at the end of the AJAX `callback`, after the `provinces` code block, passing it the two parameters. At the top of the script, assign two global variables for the width and height of the chart container (**CB12: 2**). Then, create a new variable called `chart` and assign it a code block that: (a) selects the body of the web page, (b) appends a new SVG element, *(c) assigns it the chart width and height, and (d) assigns a class attribute that can be used to select the chart container in future code (**CB12: 9-13**).

You can add a title for the chart within the SVG container itself by appending a `<text>` element, assigning it `x` and `y` attributes to position it in relation to the top-left corner of the SVG container (**CB12: 16-18**). Also assign a class name of `chartTitle` for reference in CSS (**CB12: 19**). In *style.css*, add styles to the `chart` SVG container and `chartTitle` `<text>` element to give it an appearance consistent with that of the map container (**CB13**). Use margin rules to adjust the positioning of the chart on the page (**CB13: 4-5**).

The final task in creating the bar chart is to add the bars, assigning the height of each dynamically to represent the attribute values and coloring them according to the classes symbolized on the map. Begin by using the `.selectAll()` method to create an empty selection that will accept the `csvData` array of attribute data (**CB12: 22**). Next, pass the array through `.data()`, then join the attribute data to the selection with `.enter()`, and append a new SVG `<rect>` element for each feature in the dataset (**CB12: 23-25**). To sort the data from smallest to largest attribute value, call the `.sort()` method, which takes an anonymous ***comparator*** function that uses two data

elements, `a` and `b` (**CB12: 26**). The parameter `a` represents the current data element the block is handling, and `b` represents the next element; their expressed attribute values are subtracted to determine which is smaller. In this way, the data array is re-arranged in ascending order of current attribute values before any further methods operate on it.

```
1       //global variables—at top of main.js
2       var chartWidth = 550, chartHeight = 450;
4
5       //new function below setMap()
6       function setChart(csvData, colorize){
7
8               //create a second svg element to hold the bar chart
9               var chart = d3.select("body")
10                      .append("svg")
11                      .attr("width", chartWidth)
12                      .attr("height", chartHeight)
13                      .attr("class", "chart");
14
15              //create a text element for the chart title
16              var chartTitle = chart.append("text")
17                      .attr("x", 20)
18                      .attr("y", 40)
19                      .attr("class", "chartTitle");
20
21              //set bars for each province
22              var bars = chart.selectAll(".bar")
23                      .data(csvData)
24                      .enter()
25                      .append("rect")
26                      .sort(function(a, b){return a[expressed]-b[expressed]})
27                      .attr("class", function(d){
28                              return "bar " + d.adm1_code;
29                      })
30                      .attr("width", chartWidth / csvData.length - 1);
31      };
```

Code Bank 12: Adding a bar chart (in: *main.js*).

```
1       .chart {
2               background-color: rgba(128,128,128,.2);
3               border: medium solid #999;
4               margin-left: 600px;
5               margin-top: 56px;
6       }
7
8       .chartTitle {
9               font-size: 1.5em;
10              font-weight: bold;
11      }
```

Code Bank 13: Styling the bar chart SVG container and title text (in: *style.css*).

The next method of the `bars` block assigns two class names—`bar`, which is common to every `<rect>` element, and the `adm1_code` associated with each individual province (**CB12: 27-29**). The final line assigns the width of each bar using a formula that distributes the full width of the CSV container between the bars, leaving what will become a one-pixel gap between each bar (**CB12: 30**). Note that the bar heights, the `x` and `y` positions of the bars, and the colors of the bars have yet to be assigned. Because the bars eventually will be resized and rearranged when the user changes the expressed attribute, these methods have been moved to a new code block in a separate function, `updateChart()`, that can be called whenever the expressed attribute changes. This function should be called at the end of `setChart()` and passed two parameters: the `bars` selection (**CB12: 22-30**) and the length of the `csvData` array (assigned the variable `numbars` in the function definition) (**CB14: 2**).

```
1       //function called at the end of setChart()
2       function updateChart(bars, numbars){
3               //style the bars according to currently expressed attribute
4               bars.attr("height", function(d, i){
5                       return Number(d[expressed])*3;
6                   })
7                   .attr("y", function(d, i){
8                       return chartHeight - Number(d[expressed])*3;
9                   })
10                  .attr("x", function(d, i){
11                      return i * (chartWidth / numbars);
12                  })
13                  .style("fill", function(d){
14                      return choropleth(d, colorize);
15                  });
16
17              //update chart title
18              d3.select(".chartTitle")
19                      .text("Number of "+
20                              expressed[0].toUpperCase() +
21                              expressed.substring(1,3) + " " +
22                              expressed.substring(3) +
23                              " In Each Province");
24      };
```

Code Bank 14: Function to update the bars when a new attribute is selected (in: *main.js*).

The `updateChart()` function starts by picking up where the first `bars` block left off, adding a `height` attribute and generating its value dynamically based on the value for each enumeration unit in the currently mapped attribute (**CB14: 4-6**). Each bar then is assigned a `y` attribute, which subtracts the value given for the height from the height of the SVG container so that the bars elongate from the bottom of the container rather than down from the top (keeping in mind that a `y` value of 0 is the top of the x-y plane and `y` values increase downward) (**CB14: 7-9**). The `x` attribute then is assigned using the index value of each feature in the `bars` selection to position each bar directly to the right of the previous one (**CB14: 10-12**). Finally, each bar is given a fill color using the same `choropleth()` function as the `provinces` block created in `setMap()` (**CB14: 13-15**).

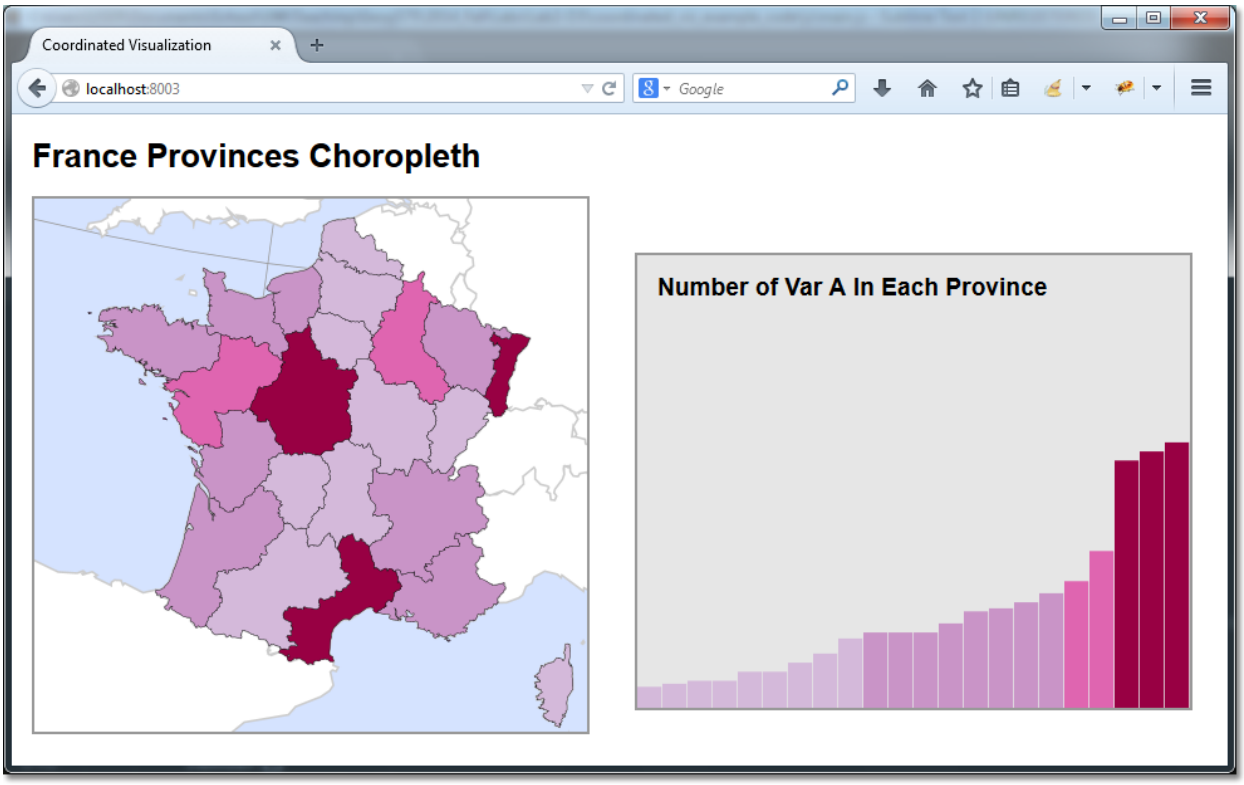

Figure 8. Styling the Choropleth Map.

It is good practice to update the title of the chart in order to reflect the currently displayed attribute. The second code block in the `updateChart()` function assigns the content of the `<text>` element that holds the title as a well-formatted string including the name of the currently expressed attribute (**CB14: 18-23**). The bar chart is now complete, but not yet interactive (**Figure 8**).

10. Implementing Dynamic Attribute Selection

With both the map and bar chart drawing properly, you are now ready to make it interactive. Interactivity changes a visualization from static to dynamic, exponentially increasing its utility while empowering the user to explore the multivariate dataset. D3 allows for an indefinite variety of map interactions, although implementing them is not automatic and requires some creativity on the part of the developer. This tutorial covers two dynamic interactions: user selection of the represented attribute in this section, and highlighting of individual enumeration units with retrieval of details about the enumeration unit using a dynamic label.

Dynamic attribute selection requires an input control allowing users to choose the attribute they would like to see represented in the choropleth map. A simple and appropriate HTML input tool is the `<select>` element, which provides a dropdown menu with a list of options. As described above, creating new HTML elements using D3 involves the `.append()` method. At the end of the `callback`, add a call to a new function called `createDropdown()` and pass it the `csvData` object as a parameter; define this function below the `setMap()` function (**Code Bank 15**).

The first block (**CB15: 2-7**) selects the HTML <body> element and appends a new element, giving it the class `dropdown` so that its position can be adjusted in *style.css*. The `.html()` method is used to simplify the creation of an <h3> element and its content (the title of the menu) within the dropdown . The final line of the block appends a new <select> element (the menu itself). The second block (**CB15: 9-20**) then uses `.selectAll()` to recursively create each menu item, feeding in the `keyArray` so that each string in the array (e.g., "varA") is used as a datum for a selection. For each selection, an <option> element is appended to the parent <select> element, and the selection's datum is assigned as the value of the <option> element's `value` attribute. The text content of the <option> element is assigned using the `.text()` method, which contains a function that uses JavaScript string methods to manipulate the datum for plain English display to the user (**CB15: 15-20**). In *style.css*, a `.dropdown` selector should be added with a `margin-left` property to adjust the position of the dropdown menu div on the page.

```
1       function createDropdown(csvData){
2               //add a select element for the dropdown menu
3               var dropdown = d3.select("body")
4                       .append("div")
5                       .attr("class","dropdown") //for positioning menu with css
6                       .html("<h3>Select Variable:</h3>")
7                       .append("select");
8
9               //create each option element within the dropdown
10              dropdown.selectAll("options")
11                      .data(keyArray)
12                      .enter()
13                      .append("option")
14                      .attr("value", function(d){ return d })
15                      .text(function(d) {
16                              d = d[0].toUpperCase() +
17                                      d.substring(1,3) + " " +
18                                      d.substring(3);
19                              return d
20                      });
21      };
```

Code Bank 15: Adding a dropdown menu in `setMap()` (in: *main.js*).

In order to enable the attribute selection dropdown, an event listener must be added that will update the choropleth map when the user changes the selected attribute. D3 uses `.on()` as the primary method for adding event listeners. This method specifies the type of event and a function that will execute when the event is fired. HTML <select> elements use the `"change"` event to determine when a user has selected a new menu item. Use the `.on()` function to add a `"change"` event listener and trigger the `changeAttribute()` function when this event is fired (**CB16: 8-10**). The `changeAttribute()` function contains the code that restyles the map according to the selected attribute (**CB16: 14-25**). First, the `expressed` variable is reassigned with the attribute option selected by the user (**CB16: 16**). The `colorize` variable must then be reset to a new `color` generator to correctly classify the new attribute values (**CB16: 17**). Recall that the `colorScale()` function sets the scale range using the *.csv* data values of the expressed attribute, as shown in **Code Bank 10**. The `path` elements of all of the existing provinces on the map are

selected and restyled by the `choropleth()` function using the new `color` generator (**CB16: 22-24**).

When the user selects a new attribute, the bars of the bar chart also should be rearranged and restyled to express the new attribute. The second block in the `changeAttribute()` function does this, using `.selectAll()` to select the existing bars and `.sort()` to reorder them from least to greatest current attribute values (**CB16: 27-30**). Animation helps to augment user feedback from the attribute selection and is aesthetically pleasing, so a D3 `transition` is added to animate the bars' change in position, height, and color (**CB16: 31**). D3 transitions *tween* two different states, changing from one to the other smoothly by interpolating the intermediate values using built-in interpolator functions for colors, strings, numbers, and arrays. For more information on transitions, see the D3 API Transitions page (https://github.com/mbostock/d3/wiki/Transitions). A slight delay is added to each bar to add a sequential appearance to the animation (**CB16: 32-34**). Finally, the `updateChart()` function is called to update the position, height, and fill color of each bar (**CB16: 37 → CB14**).

```
1     function createDropdown(csvData){
2           //add a select element for the dropdown menu
3           var dropdown = d3.select("body")
4                 .append("div")
5                 .attr("class","dropdown") //for positioning menu with css
6                 .html("<h3>Select Variable:</h3>")
7                 .append("select")
8                 .on("change", function(){
9                       changeAttribute(this.value, csvData);
10                });
11
12    //REMAINDER OF CODE BANK 12 AND CODE BANK 10 SUPPRESSED FOR SPACE
13
14    function changeAttribute(attribute, csvData){
15          //change the expressed attribute
16          expressed = attribute;
17          colorize = colorScale(csvData);
18
19          //recolor the map
20          d3.selectAll(".provinces") //select every province
21                .select("path")
22                .style("fill", function(d) { //color enumeration units
23                      return choropleth(d, colorize); //->
24                });
25
26          //re-sort the bar chart
27          var bars = d3.selectAll(".bar")
28                .sort(function(a, b){
29                      return a[expressed]-b[expressed];
30                })
31                .transition() //add animation
32                .delay(function(d, i){
33                      return i * 10
34                });
35
36          //update bars according to current attribute
37          updateChart(bars, csvData.length);
38    };
```

Code Bank 16: Adding an event listener and callback function (in: *main.js*).

11. Implementing Coordinated Highlighting & Dynamic Labels

There are two steps left for completing the interactive choropleth map: (1) providing visual feedback when probing an enumeration unit (i.e., ***highlighting***) and (2) activating a tooltip (i.e., a ***dynamic label***) supporting the retrieval of details about the probed enumeration unit. You will create three functions to make these work: (1) `highlight()`, which restyles the probed enumeration unit and populates the content for the dynamic label on `mouseover` (**CB17: 13**), (2) `dehighlight()`, which reverts the enumeration unit back to its original color and deactivates the dynamic label on `mouseout` (**CB17: 14**), and `(3) moveLabel()`, which updates the position of the dynamic label according to changes in the x/y coordinates of the mouse on `mousemove` (**CB17: 15**). The event listeners should be added at the end of the `provinces` block in `setMap()` (**CB11 → CB17: 13-18**) to make each enumeration unit in the choropleth interactive, making sure you update the position of the semicolon that ends the block.

When implementing highlighting across features that are styled differently (as with the varying color scheme in a choropleth map), it is necessary to store the original color of the highlighted feature for when the feature is subsequently "dehighlighted". This approach is faster than reprocessing the *.json* object. The solution in **Code Bank 17** appends the original color as a text string in a `<desc>` SVG element (**CB17: 17-19**). The contents of the `<desc>` element then can be referenced to extract this color when reverting the enumeration unit to its original choropleth styling upon `dehighlight()`. Note that for highlighting to work properly for all attributes, the `desc` element should be selected and reset at the end of the block in the `changeAttribute()` function, as shown in **Code Bank 18**.

```
1     var provinces = map.selectAll(".provinces")
2           .data(topojson.feature(europe,
                  europe.objects.FranceProvinces).features)
3           .enter() //create elements
4           .append("path") //append elements to svg
5           .attr("class", "provinces") //assign class for styling
6           .attr("class", function(d) {
7                 return "provinces " + d.properties.adm1_code;
8           })
9           .attr("d", path) //project data as geometry in svg
10          .style("fill", function(d) { //color enumeration units
11                return choropleth(d, colorize);
12          })
13          .on("mouseover", highlight)
14          .on("mouseout", dehighlight)
15          .on("mousemove", moveLabel)
16          .append("desc") //append the current color
17                .text(function(d) {
18                      return choropleth(d, colorize);
19          });
```

Code Bank 17: Adding Event Listeners to `provinces` in `setMap()` (in: *main.js*).

Highlighting and the *retrieve* operator should be coordinated between the map and chart, or other visualization of your choosing. As described in lecture, ***coordination*** is the application of an operator evoked in one view on associated information elements in all other views, and is fundamental to the success of exploratory geovisualization. To do this, simply add the same event

listeners to the end of the bars block in `setChart()` as were added to the provinces block in `setMap()` (**CB19: 11-13**). The event handers for these three functions will be the same as those used on the map: `highlight()`, `dehighlight()`, and `moveLabel()`, respectively. These functions will be called when the user interacts with either the map or the bar chart, activating a tooltip and affecting the styling of the matching elements in both.

```
1     function changeAttribute(attribute, csvData){
2           //change the expressed attribute
3           expressed = attribute;
4
5           //recolor the map
6           d3.selectAll(".provinces") //select every province
7                 .select("path")
8                 .style("fill", function(d) { //color enumeration units
9                       return choropleth(d, colorScale(csvData)); //->
10                })
11                .select("desc") //replace the color text in each desc element
12                      .text(function(d) {
13                            return choropleth(d, colorScale(csvData)); //->
14                      });
15    };
```

Code Bank 18: Resetting the `<desc>` element in `changeAttribute()` (in: *main.js*).

Once adding the event listeners, you then must define the event handler functions. First, add a new function named `highlight()` (**CB19**). The `highlight()` function receives the `data` object associated with highlighted enumeration unit or bar as the parameter. The `data` object's properties are stored in a local variable named `props` (**CB19: 4**). The `provinces` listener passes the complete JSON features, of which the `properties` are a subset, while the `bars` listener passes only the properties of each feature from the CSV data; hence this line tests whether the `properties` object exists within the `data`, and if so assigns it to `props`, and if not assigns the `data` object itself (this shorthand syntax can be used for any simple if-then statement in JavaScript). The `highlight()` function then calls `d3.selectAll()` to find both the `path` element in the map and the `rect` element in the chart with `class` attributes that match the selected feature's `adm1_code`, and changes their `fill` style to black (**CB19: 6-7**). There are many other possible highlighting solutions; this one was chosen for simplicity.

```
1           //set bars for each province
2           var bars = chart.selectAll(".bar")
3                 .data(csvData)
4                 .enter()
5                 .append("rect")
6                 .sort(function(a, b){return a[expressed]-b[expressed]})
7                 .attr("class", function(d){
8                       return "bar " + d.adm1_code;
9                 })
10                .attr("width", chartWidth / csvData.length - 1)
11                .on("mouseover", highlight)
12                .on("mouseout", dehighlight)
13                .on("mousemove", moveLabel);
14    };
```

Code Bank 19: Adding Event Listeners to `bars` in `setChart()` (in: *main.js*).

The `highlight()` function then designates two HTML strings used in the dynamic label, one with the attribute data (`labelAttribute`) and one with the province name (`labelName`) (**CB19: 9-11**). Note how the `labelAttribute` variable makes use of the global `expressed` variable to determine the attribute to include in the label. As above, you are encouraged to adjust the content of the dynamic label based on the purpose of your map. Finally, the `highlight()` function creates a new `<div>` element named `infolabel` to hold the dynamic label (**CB19: 14-20**). A child `<div>` named `labelname` is added to `infolabel` to position the province name within the label.

Add style rules to the dynamic label in *style.css*, using the `infolabel` and `labelname` class identifiers. **Code Bank 20** provides basic style rules to create a 200x50px dynamic label with white text and a black background. Note that the `<h1>` and `<b>` tags are styled for the `infolabel` text (**CB20: 17-26**), as these are used in the `labelAttribute` HTML string (**CB19: 9-10**); you are not limited to these tags in the design of your dynamic label.

```
1     function highlight(data){
2
3           //json or csv properties
4           var props = data.properties ? data.properties : data;
5
6           d3.selectAll("."+props.adm1_code) //select the current province
7                 .style("fill", "#000"); //set the enumeration unit fill to black
8
9           var labelAttribute = "<h1>"+props[expressed]+
10                              "</h1><br><b>"+expressed+"</b>"; //label content
11          var labelName = props.name; //html string for name to go in child div
12
13          //create info label div
14          var infolabel = d3.select("body").append("div")
15                .attr("class", "infolabel") //for styling label
16                .attr("id", props.adm1_code+"label") //for label div
17                .html(labelAttribute) //add text
18                .append("div") //add child div for feature name
19                .attr("class", "labelname") //for styling name
20                .html(labelName); //add feature name to label
21    };
```

Code Bank 19: Highlighting the Choropleth Map (in: *main.js*).

Reload your *index.html* page in your browser and inspect your work. At this point, mousing over an individual enumeration unit or bar should result in highlighting both the unit and bar and *retrieving* the associated attribute value in a label (**Figure 9a**). However, the enumeration units do not return to their original color on `mouseout`, as you have yet to implement the `dehighlight()` function (note the error in the browser console). **Use what you have learned up to this point to define the `dehighlight()` function on your own**. The `dehighlight()` function represents the conceptual opposite of the `highlight()` function, requiring you to revert the enumeration unit color and deactivate the dynamic label. Remember that the color text is stored in the `desc` variable of the `provinces` path being dehighlighted. You will need to investigate the `remove()` method in D3 to learn how to remove an element (the dynamic label) from the DOM.

Also, the dynamic label, while updating on `mouseover`, is positioned statically off to the side of the page rather than associating with the selected province or bar. To make the dynamic label follow the user's mouse cursor, first change the positioning of both `<svg>` elements to absolute in *style.css*

(**CB20: 1-3**); this should be defined early in the stylesheet so it may be overridden by individual class rules. Then, define the `moveLabel()` function in *main.js* that is called on `mousemove` atop an enumeration unit (**Code Bank 21**). The `moveLabel()` function uses the `d3.event()` method to access the current mouse event (`mousemove`), which includes mouse coordinate properties (`clientX` and `clientY`). The function simply accesses the mouse coordinates of the event and uses them to offset the label in relation to the `body` element, the lowest DOM element that is relatively positioned. If you changed the size of the dynamic label in *style.css*, you need to adjust the horizontal and vertical label coordinates according to the revised width and height (**CB21: 3-4**). If your label overflows the window in some areas, think about how you might modify these lines with `if-else` statements using the `window.innerWidth` and `window.innerHeight` properties to change the positioning of the label when it approaches the window edge to prevent it from going off the page.

```
1       svg {
2               position: absolute;
3       }
4
5       /*CODEBANK 4 & 6 SUPPRESSED FOR SPACE*/
6
7       .infolabel {
8               position: absolute;
9               width: 200px;
10              height: 50px;
11              color: #fff;
12              background-color: #000;
13              border: solid thin #fff;
14              padding: 5px;
15      }
16
17      .infolabel h1 {
18              margin: 0;
19              padding: 0;
20              display: inline-block;
21              line-height: 1em;
22      }
23
24      .infolabel b {
25              float: left;
26      }
27
28      .labelname {
29              display: inline-block;
30              float: right;
31              margin: -25px 0px 0px 40px;
32              font-size: 1em;
33              font-weight: bold;
34              position: absolute;
35      }
```

Code Bank 20: Styling the Dynamic Label (in: *style.css*).

Reload the *index.html* page; you now should have a functional *retrieve* operator that includes highlighting and a dynamic label that follows the mouse (**Figure 9b**).

```
1       function moveLabel() {
2
3               var x = d3.event.clientX+10; //horizontal label coordinate
4               var y = d3.event.clientY-75; //vertical label coordinate
5
6               d3.select(".infolabel") //select the label div for moving
7                       .style("margin-left", x+"px") //reposition label horizontal
8                       .style("margin-top", y+"px"); //reposition label vertical
9       };
```

Code Bank 21: Styling the Dynamic Label (in: *main.js*).

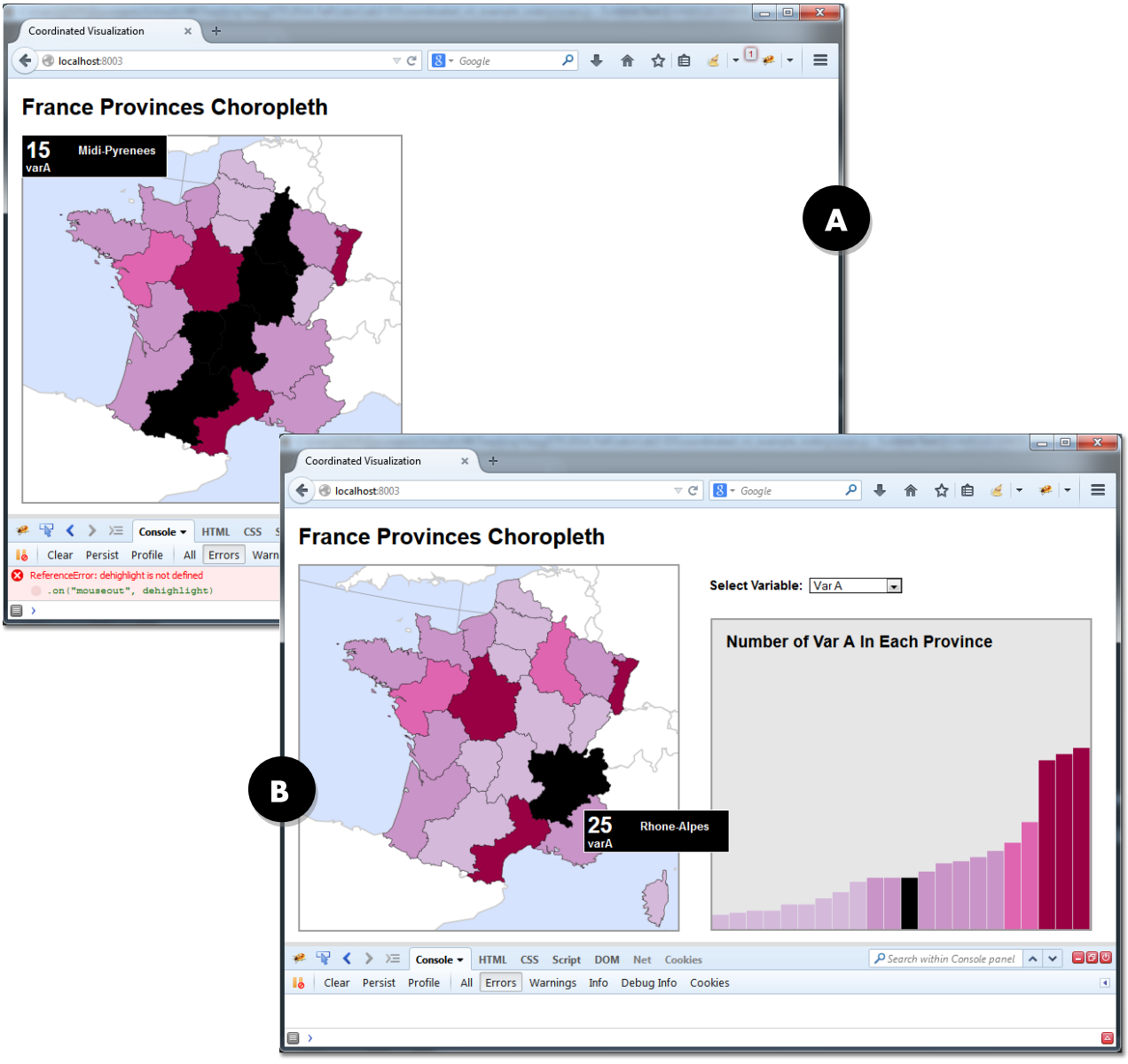

Figure 9. Implementing Highlighting and Tooltips in the Choropleth Map and Chart.

12. Transition What You've Learned: Be Awesome

You Coordinated Visualization Lab deliverable is an impressive display of modern web mapping using open source and mobile friendly technologies. You should be proud of the work you have accomplished; to think, JavaScript was introduced to you only 10 weeks ago! Push yourself to learn new features of D3 by extending the lab requirements. You are encouraged to continue to add functionality to your visualization; please see the rubric below for extra credit opportunities on your lab based on exploration and implementation of D3.

Evaluation Rubric: Interaction Challenge (50pts)

Delivery: You are required to publish a version of your map to the web through GitHub Pages or another service AND upload a *.zip* of your entire directory to the Learn@UW Lab #2 Dropbox at least one hour before your lab on **November 17 or 18**. While you may receive bonus points by implementing additional views, you cannot exceed 40 points overall on this assignment.

Weekly Check-In Points (14)

(4) October 27-28th: Multivariate Dataset Due (i.e., *at least* through Step #1)
- *submit dataset to Learn@UW dropbox*

(4) November 3-4th: Choropleth Basemap Due (i.e., *at least* through Step #4)
- *submit your zipped code & a GitHub URL to Learn@UW dropbox*

(6) November 10-11th: Attribute Sequencing Due (i.e., *at least* through Step #9)
- *submit your zipped code & a GitHub URL to Learn@UW dropbox*

Representation: Choropleth Map View (8)

(8 points) The basemap is appropriately designed for the map scenario with correct projection, level of generalization, visual hierarchy, etc.; the data is normalized; the choropleth has an appropriate color scheme and class structure.

(6 points) The basemap and/or color scheme have one or two minor design flaws.

(4 points) There are multiple design flaws with the basemap, color scheme, and class structure, or the data is not normalized.

(2 points or below) There are serious errors with the data and/or design scheme.

Representation: Coordinated View (8)

(8 points) The bar chart or other main data graphic draws correctly; visual elements (bars, etc.) are positioned logically to help the user create meaning from the data; explanatory elements (titles, labels, etc.) appear where they are necessary or appropriate.

(6 points) Visual elements could be more clearly positioned or labeled.

(4 points) The graphic is not overtly meaningful for the scenario, or there are multiple flaws with the positioning, styling, etc. of the visual and/or explanatory elements.

(2 points or below) The graphic fails to draw correctly, is buggy, and/or has several visual flaws.

Coordinated Interaction: *Sequence* (8)

(8 points) The map and additional data graphic(s) update correctly when sequencing by attribute; the user is given adequate visual affordances to show how to update the visualizations and feedback that the shows the update took place.

(6 points) Visual affordances and/or feedback are missing.

(4 points or below) There are bugs in the interaction that cause the visualizations not to update correctly.

(2 points or below) Attribute sequencing is not implemented.

Coordinated Interaction: *Retrieve* (8)

(8 points) Dynamic labels open on the map and additional data graphic(s); highlighting feedback is coordinated across views; the information window is well designed and contains information that supports the design scenario.

(6 points) Dynamic labels are not well designed or there are minor problems with how they display on the page.

(4 points) There are bugs in the dynamic labels and/or the feedback does not coordinate across views properly.

(2 points or below) The retrieve operator is not implemented.

Design for Scenario (4)

(4 points) The design is clear, creative, consistent across information views, and fits the scenario.

(2 points or below) There are inconsistencies in the design, a mismatch with the scenario, or it is bland.

Extra Credit (4+)

(+4) Submit a ~1 page write-up discussing the parts of this lab and the covered technologies that were unclear, places you were lost, and "Aha!" moments you may have had along the way.

(+2) Implement a choropleth legend that updates with the attribute sequencing.

(+2) Each additional coordinated view.

(+2) Each additional implemented operator.