Geography 575
Lab #3: Coordinated Visualization Challenge

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 **40 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:

- **March 21st:** Lab #3 Assigned //collaboration begins
- **April 4th:** Work Period //input & feedback from collaborator
- **April 11th:** Work Period //input & feedback from collaborator
- **April 18th:** Lab #3 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 application 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 units and a PCP that represents *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.

1. Coordinated, Multivariate Visualization on the Web

a. Overview of D3

In Labs #1 and #2, you learned how to use JavaScript to create a Leaflet slippy map, load and process external information, style and resize point markers, skin UI controls, and implement interaction operators using JavaScript, jQuery, and jQueryUI. While tile-based slippy maps speed map browsing considerably and are easy to make, they present significant constraints on cartographic design. Among the largest of issues is restriction in map projection. Slippy maps make use of cylindrical map projections to optimize the processing and serving of map tiles. Most existing tilesets are provided in the Web Mercator projection, which is conformal (acceptable for navigation-focused reference mapping) and not equivalent (a property that should be preserved for many thematic maps, such as choropleth, dot density, and isoline when using color shading between intervals). Because you are making a choropleth map as one of the coordinated views, you should not rely on a non-equivalent tile service for your basemap.

Additionally, maps used for exploration—or Geovisualization—require implementation of multiple, linked information graphics that can be hard to create with JavaScript or jQuery alone. Luckily, there are a growing number of information visualization libraries that can be used in concert with mapping libraries to create interesting, coordinated visualizations. (for some examples, see: http://www.idea.org/blog/2012/10/25/great-tools-for-data-visualization). In Lab #3, you will be using the D3 visualization library to build your first coordinated visualization (http://d3js.org). ***D3***, or ***data-driven documents***, is a JavaScript library pioneered and maintained by Mike Bostock of the NYTimes (http://bost.ocks.org/mike); the D3 library grew from prior collaborations on the Protovis library between Bostock and Professor Jeffrey Heer of Stanford University. Increasingly recognized as a leading data visualization library, D3 is robust, elegant, stable, and expanding rapidly. It simplifies loading and interacting with information, extending the dot syntax used by jQuery. It draws all graphics as client-side (in the browser) vectors using SVG (return to Lab #1 for details about SVG). The map visualizations in D3 can be reprojected using any class, case, aspect, and centering thanks to the work of freelance developer Jason Davies and the *proj4.js* library of map projections (http://trac.osgeo.org/proj4js/).

The D3 library is an open-source project on GitHub that seems to evolve daily (https://github.com/mbostock). The goal of Lab #3 is to provide you a broad introduction to key concepts in using the library; the great wealth of the library could cover a series of advanced courses on Interactive Cartography and Geovisualization. The following introduction to D3 extends from 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://ofps.oreilly.com/titles/9781449339739/index.html). Refer to these documents for additional background and guidance as you complete Lab #3.

b. Finding and Formatting Multivariate Information

As with Lab #1, the first step towards completing the challenge is assembly of an appropriate information set for geographic visualization; for Lab #3, the information set will be ***multivariate*** (i.e., multiple attributes enumerated over the same set of spaces) rather than spatiotemporal. While you will continue to use the *.csv* format to hold your attribute information (as with Labs #1 and #2), you will make use of the *.json* format to hold your geographic information. ***JSON*** stands for JavaScript Object Notation and quickly has become a standard format for information loaded into

and interpreted by a browser. The *.json* format is leveraged in Lab #3 due to the use of enumeration units (polygons) rather than markers (points) in the choropleth map (although *.json* can be used for markers as well).

First, prepare the attribute information in a *.csv* file. Common agencies that provide enumerated, multivariate information include FedStats (http://www.fedstats.gov/), many US Federal Bureaus, and the World Bank (http://data.worldbank.org/); consider consulting with Jaime Stoltenberg in the Robinson Map Library to acquire specific multivariate information. Regardless of source, the format of the multivariate information set should be similar to the spatiotemporal information set used for Labs #1 and #2; unique enumeration units should be included as rows and unique attributes should be included as columns. **Figure 1** provides an example multivariate information set for provinces in France. Remember that enumerated information must be normalized when depicted using a choropleth map in order to account for variation in the size/shape of enumeration units. After keying the raw information into the attribute table, consider how best to normalize it for the choropleth map (if it is not already normalized). Return to your G370 notes regarding discussion on normalizing information for choropleth mapping.

	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 Information Set. In the table, the enumeration units should be included as rows and the attributes included as columns; for simplicity, normalize the information in the *.csv*. Note: The above information is meaningless.

Next, prepare the geographic information in a *.shp* file for subsequent conversion to *.json*. Begin by finding a *.shp* file that provides geometry for your enumeration units as well as any additional context information you wish to overlay in the choropleth map. Web-based repositories containing enumeration units in the vector *.shp* format include the Esri Data Bank, Geocommons, and Natural Earth. The following instructions make use of the Admin 0 and Admin 1 small-scale cultural files included in Natural Earth (http://www.naturalearthdata.com/) to produce two *.shp* files for the choropleth map: (1) *FranceProvinces.shp* (extracted from Admin 1) containing the enumeration units for a choropleth map and (2) *EuropeCountries.shp* (extracted from Admin 0) for basemap context. One column in the *.csv* and one column in the *.shp* must match to join the files together; in the **Figure 1** example, "adm1_code" has been copied from Natural Earth into the *.csv.* Note: The column you use to join the *.csv* and *json* CANNOT include records that start with a number or reserved character, as these are used as object names when loaded into the browser.

It is recommended that you do some processing of the *.shp* file in ArcMap before converting it to *.json*. First, delete any unneeded geographic features and attribute columns from *.shp* file to reduce the overall file size. Next, simplify your linework as much as possible, as additional geometry nodes slow the loading and interaction of your geovisualization; you may use either the ArcToolbox simplification tools or MapShaper (http://mapshaper.com) to remove points from your *.shp* files (return to Lab #1 from G370 to refresh your memory on simplification/generalization). It is important to note that ArcGIS and the *.shp* file format can be bypassed altogether using the open-source GDAL/OGR python library (installed on all lab machines as part of OSGeo4W and free to download); for additional details, see: http://www.gdal.org/ogr/drv_geojson.html.

c. Converting Your Information from *.shp* to *.json*

There are two spatially-enabled variants of the *.json* format: GeoJSON and TopoJSON; both variants continue to make use of the *.json* extension. Both the GeoJSON and TopoJSON formats structure the geometry and attribute values of a given map feature (here, a polygon) as a set of properties associated with a JavaScript object; therefore, both geographic and multivariate information are placed into the DOM in a similar manner when loaded into the browser. The ***GeoJSON*** format structures the geographic information for each polygon as an array of ***nodes*** (lat/long coordinate pairs) defining the complete outer boundary of the polygon. In contrast, the ***TopoJSON*** format—pioneered by D3's Mike Bostock in early 2013—structures the geographic information for each polygon as a series of ***arcs*** (lines connecting a pair of nodes), and stores the pair of nodes constituting each arc in a separate JavaScript object. The result of the TopoJSON format is that ***topology***, or shared arcs/edges, is explicit. As you learned in G377, the topological format is more efficient, as the nodes for shared edges are defined in the file only once and therefore drawn in the browser only once. We therefore will use the TopoJSON format for Lab #3, although GeoJSON remains common across web mapping applications in part because many plugins do not yet support the newer TopoJSON format.

Convert your processed *.shp* files containing the polygon geometry to *.json* using the web service ***Shape Escape*** (http://www.shpesape.com). Initially developed by Josh Livni of Google to aid conversion of *.shp* files to Google Fusion Tables, the service recently was updated to include conversion of *.shp* files to GeoJSON and TopoJSON. To use Shape Escape, first compress all of the constituent files of your collected *.shp* files into a single *.zip* file. It is not necessary to separate different *.shp* files into different *.zip* files; in the following example, the processed *FranceProvinces.shp* and *EuropeCountries.shp* are compressed to a single *europe.zip* file.

Once compressed, navigate to Shape Escape, select the "shp2geoJSON/topoJSON" option (**Figure 2a**) and upload your *.zip* file when prompted (**Figure 2b**). Upon upload, you are redirected to a Google Map view with a sidebar on the left that lists the new files created from your *.shp* files (**Figure 2c**). There should be one GeoJSON listed for each *.shp* file included in the *.zip* and a TopoJSON at three levels of simplification (described as "precision"); to improve loading and interaction. Click "display" under any of these files to preview the geometry on the map. Click "download" to redirect to a page that contains the GeoJSON or TopoJSON definition (**Figure 1d**). Click the "with attributes" option if you want to preserve all attributes from the *.shp* files in the TopoJSON; see the Shape Escape FAQ if you want to select only particular attributes. Copy the contents of the page (Ctrl+A/Ctrl+C) and paste (Ctrl+P) into NotePad++, saving the file with the extension *.json* (e.g., *europe.json*).

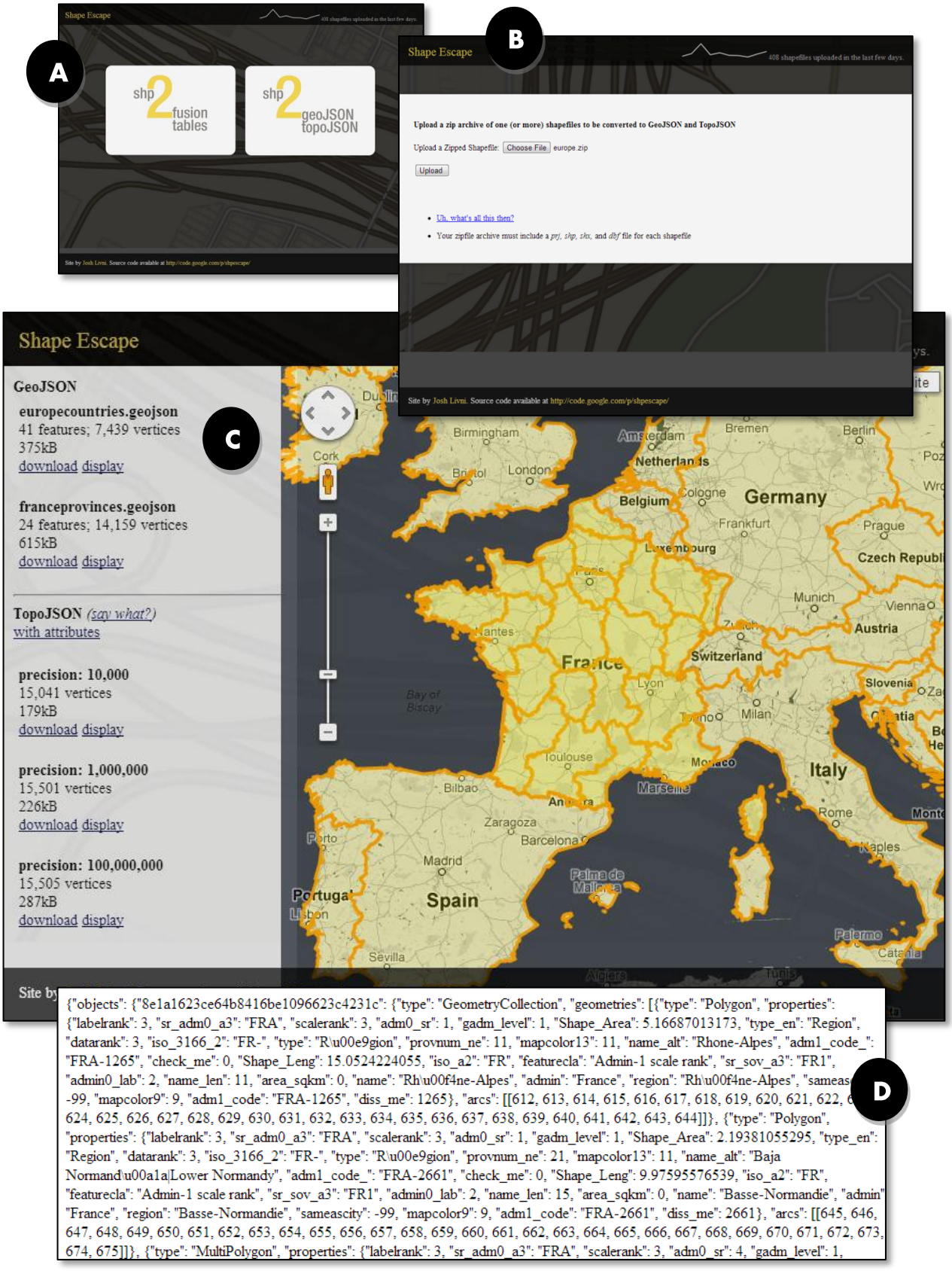

Figure 2. Using Shape Escape (http://www.shpescape.com) to convert *.shp* to *.json*

Before moving on, inspect the difference between the GeoJSON and TopoJSON formats. Starting with GeoJSON, notice that the first character is a curly brace ({), denoting the start of a JavaScript object (hence "JavaScript OBJECT Notation"). In each GeoJSON, the `type` is a `featureCollection`. Each object contains a `features` array of objects, the first object holding its `geometry` (an array of nodes) and the second holdings its `properties` or attributes. Incidentally, Leaflet also can create a feature group from a GeoJSON, a solution that may be useful for the final project (see http://leafletjs.com/examples/geojson.html).

Next, look at the TopoJSON format. The overall `type` is now `Topology`, and instead of `features` there are `geometries` with `arcs`. Instead of one `featureCollection`, there are one or more `geometryCollections` stored as `objects`, with each `object` corresponding to one of the input *.shp* files. Important: You need to rename manually each `object` from the random string of letters/numbers assigned to it by Shape Escape to names that you can reference in your code (e.g., `"FranceProvinces"` and `"EuropeCountries"` in **Figure 3**). For more on the TopoJSON specification, see https://github.com/mbostock/topojson/wiki.

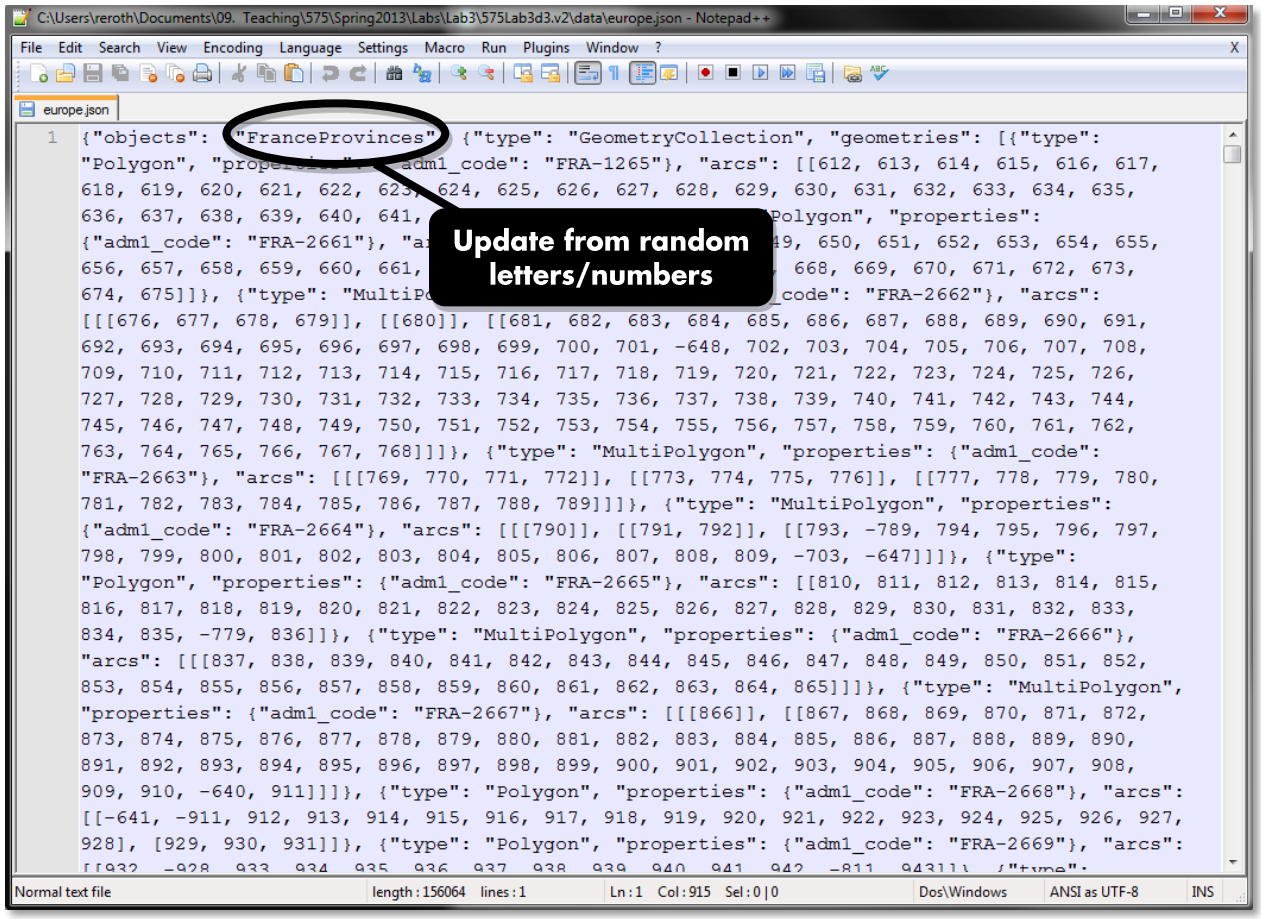

Figure 3. Manually rename the `objects` in your TopoJSON file for reference in your code.

2. The Choropleth Map View

a. Preparing Your Directory / Structure

With your TopoJSON processed, it is now time to start building your coordinated visualization! As in the first two labs, create a local directory for Lab #3 that includes folders named "css", "data", "img", and "js". Then, create three new files named *index.html* (root level), *style.css* (*css* folder), and *main.js* (*js* folder); save your newly created *.csv* and *.json* files into the *data* folder. Finally, add the Lab #1 basic HTML5 boilerplate into your *index.html* file, linking to your newly created stylesheets and scripts (**Code Bank 1**).

After configuring your directory, acquire two existing *.js* files from Bostock's Github account: (1) *d3.v3.js* (https://github.com/mbostock/d3) containing the D3 visualization library and (2) *topojson.js* for parsing your TopoJSON file (https://github.com/mbostock/topojson/). Save these files to your *js* folder and link to them in *index.html* (**CB1: 11-13**). Again return to Lab #1 for details about downloading source code from Github; remember, you only should include the required *.js* source files in your final build.

```
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="js/d3.v3.js"></script>
13                      <script src="js/topojson.js"></script>
14
15                      <!--link to main javascript file-->
16                      <script src="js/main.js"></script>
17              </body>
18      </html>
```

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

b. Loading Your *.json* into the Browser

The next step is loading the geographic information assembled in *europe.json*. Remember the `ProcessCSV` prototype function that you used to load and parse your spatiotemporal information for Lab #1? One of the beauties of D3 is that it provides for you a series of functions for processing information in various formats. The `d3.csv()` function performs the same action as the more cumbersome `ProcessCSV` prototype from Lab #1. For Lab #3, you first will use the `d3.json()` function, which places the information in your TopoJSON into the brower's memory. **Code Bank 2** provides the logic needed to initialize the webpage and print the *europe.json* file to the console. Load the *index.html* page in Firefox; you now will see your TopoJSON loaded to the DOM (**Figure 4**).

```
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              //retrieve and process europe json file
12              d3.json("data/europe.json", function(error, europe) {
13                      console.log(europe);
14              });
15      }
```

Code Bank 2: Loading *europe.json* and Printing to the Console (in: *main.js*).

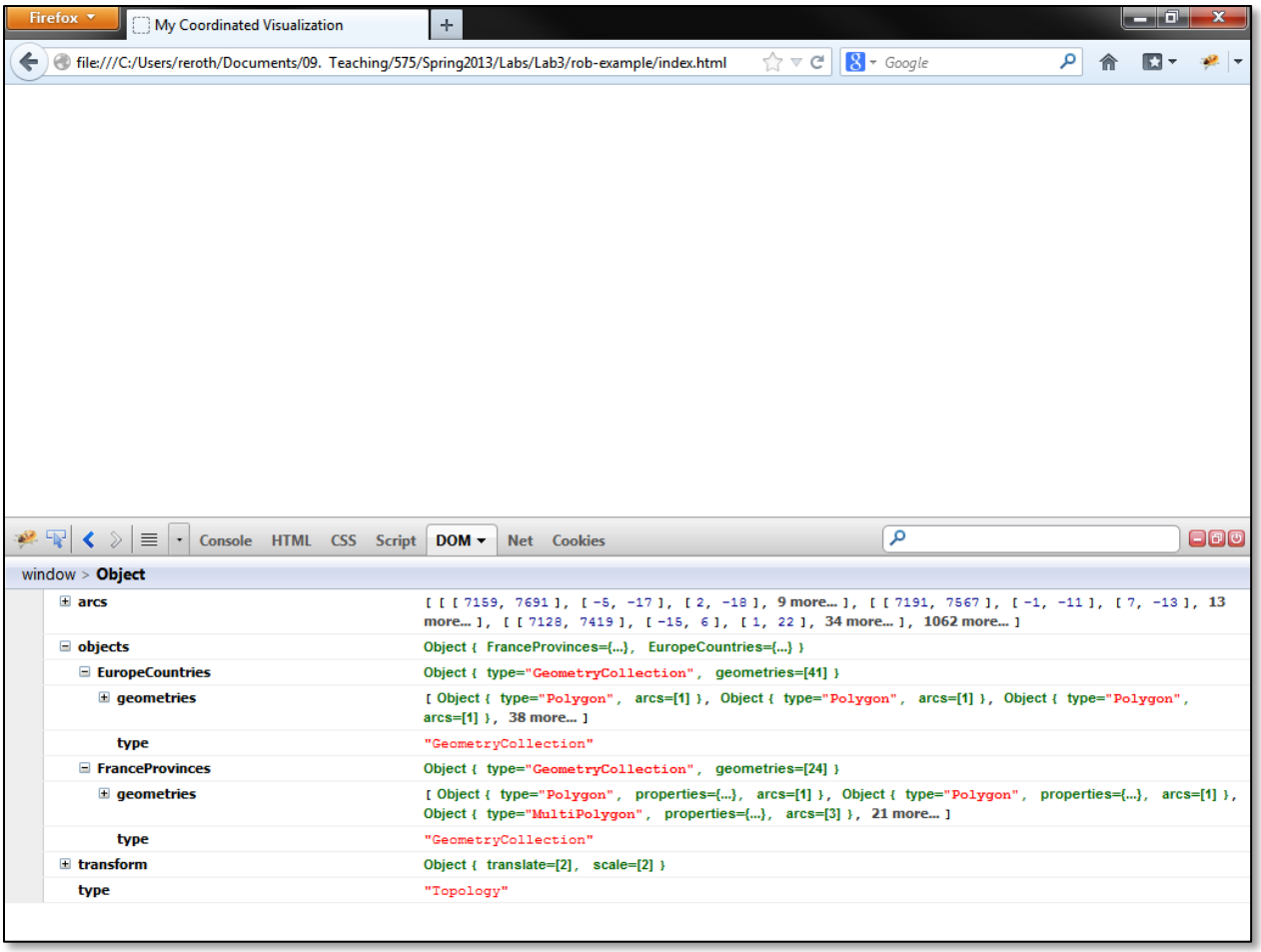

Figure 4. Printing the TopoJSON to the DOM. The object names should match names given in **Figure 3**, which in turn should match the original *.shp* files.

c. Drawing Your Basemap

Now that your information is loading properly, it is time to make a map! 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. In Labs #1 and #2, you added a `div` element to *index.html* to contain the Leaflet map. For Lab #3, you instead will create a blank `svg` element for each of the included visualizations and populate its content using JavaScript. As introduced in Lab #1, the *.svg* file format supports the drawing and styling of vector-based graphics. D3 is built atop existing browser capability to render SVG. It will be easier to interpret the following instructions in this subsection if you 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          //retrieve and process europe json file
26          d3.json("data/europe.json", function(error,europe){
27                //add Europe countries geometry to map
28                var countries = map.append("path") //create SVG path element
29                      .datum(topojson.object(europe,
                                    europe.objects.EuropeCountries))
30                      .attr("class", "countries") //class name for styling
31                      .attr("d", path); //project data as geometry in svg
32          });
33    }
```

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

Next, create an `svg` element to contain the choropleth map using the `d3.select()` function (**CB3: 7-11**). The statement `d3.select("body")` is essentially the same as `$("body")` in jQuery, except it only returns the first matching element in the DOM instead of all matching elements. Like jQuery, D3 uses dot syntax to string together function calls, 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()`, is evoked later in the instructions to select every element that matches the selector…even if those elements haven't been created yet!

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). You did not need to indicate a projection in Leaflet due to the reliance on a tileset already projected into Web Mercator. 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 plugin, is available at: https://github.com/mbostock/d3/wiki/Geo-Projections. Choose a projection that is cartographically appropriate for your mapped phenomena.

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 function 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) away from the geographic center;
- `.parallels` sets the standard parallels of the projection;
- `.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()` function creates a new `path` generator with a default projection of Albers USA. This logic may run counter to that which you experienced in Labs #1 and #2; if D3 worked like Leaflet, you might expect that calling `d3.geo.path()` will return a ***variable***, like an object or an array. Instead, the `d3.geo.path()` is a D3 ***generator function*** that creates a new function (the ***generator***) based on the parameters you send it. You can then store this generator function as a variable, and access the variable like you would call a function, passing it variables to manipulate. Note that the `d3.geo.path()` function 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 `d3.geo.path()` generator function. Hopefully, the idea and usage of generator functions will become clearer as you proceed through Lab #3.

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()` function to add an `svg` element

containing the geometry derived from your TopoJSON, projected according to the generator definition (**CB3: 25-32**); note that this code should be part of the function called by `d3.json()`, replacing the console log. The first line, `append("path")`, adds the referenced `svg` element (`countries`) using the `path` generator (**CB3: 28**). The second line specifies the `datum()` that will be attached to this `path` element (**CB3: 29**); in D3 terms, a ***datum*** is a unified chunk of information that can be expressed in SVG form.

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()` function is used, rather than first calling the `selectAll()` and `enter()` functions (described below). The `datum()` function expects a JSON or GeoJSON; to indicate to D3 that you are using the newer TopoJSON, access the `topojson.object()` method from *topojson.js*, indicating the object (`EuropeCounties`) in the TopoJSON you want translated. The third line assigns that `countries` element the class name "`countries`" so that it can be styled in *styles.css* (**CB3: 30**). 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: 31**). 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.

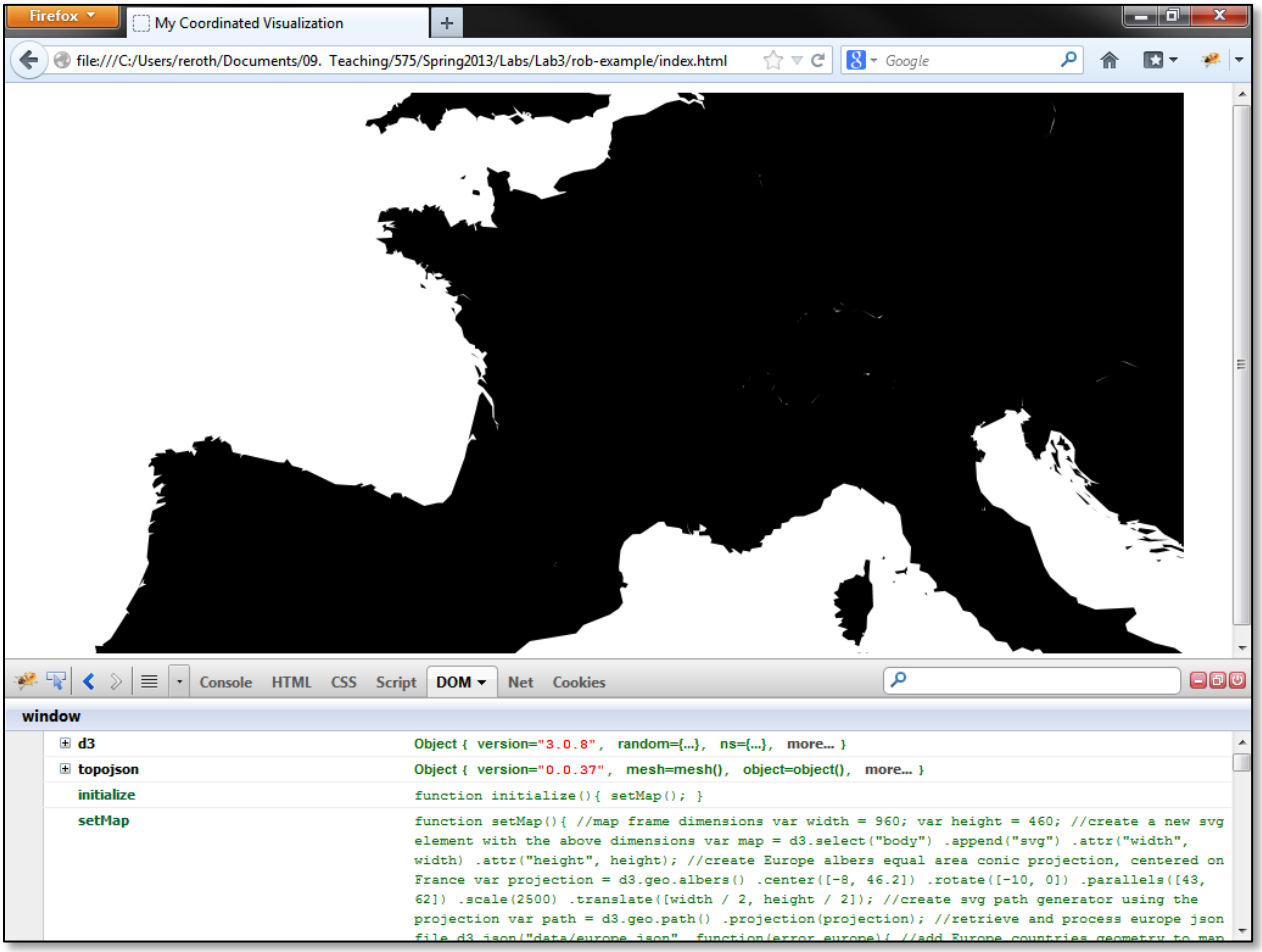

Figure 5. Drawing the Basemap.

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**; **26-32**). 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. A semicolon can be placed at the end of the block to denote its end, although this really only makes a difference if you plan to minify your script for deployment. To maintain legibility, 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. If a separate *main.js* function is called from a block, that function's annotation will refer to the variable name of the block that the function is called from. Now refresh your browser. Look, it's a map (**Figure 5**)!

d. 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: 30**). 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 lab.

```
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**). At the time of writing Lab #3, the `d3.geo.graticule()` function was not yet integrated into the D3 API; however, an example of the `d3.geo.graticule()` function is available at: http://bl.ocks.org/mbostock/3734308. (Note: `graticule()` was added into the API over spring break in v3.1!).

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 up your visual hierarchy from the buttom-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 using `d3.json()`; 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()` function and configure its attributes (**CB5: 5-9**). Then, add `gratLines` to the `path` element using `selectAll()` and `enter()` and configure its attributes (**CB5: 11-17**). As stated above, the use of `selectAll()`/`enter()` is different from the use of `append()` alone. D3's `selectAll()` and `enter()` functions are used to create multiple, new 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()` function creates an empty selection, which allows an element to appear for each graticule line (**CB5: 12**). The `data()` function operates like `datum()`, but computes a function that will join each datum (each data value or array within the overall data array) to its own element (**CB5: 13**). The `enter()` function compares the number of data values ("datums") to the number of DOM elements that match the selector, and counts each element that is needed but does not already exist (**CB5: 14**). The `append()` function adds a new `svg path` element for each element counted by `enter()`, 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 function of `selectAll(".gratLines")` is to select elements that do not exist yet in the DOM. The second `attr()` call projects the data through the `path` generator into the `d svg` 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 Firefox to view your basemap (**Figure 6**).

```
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*).

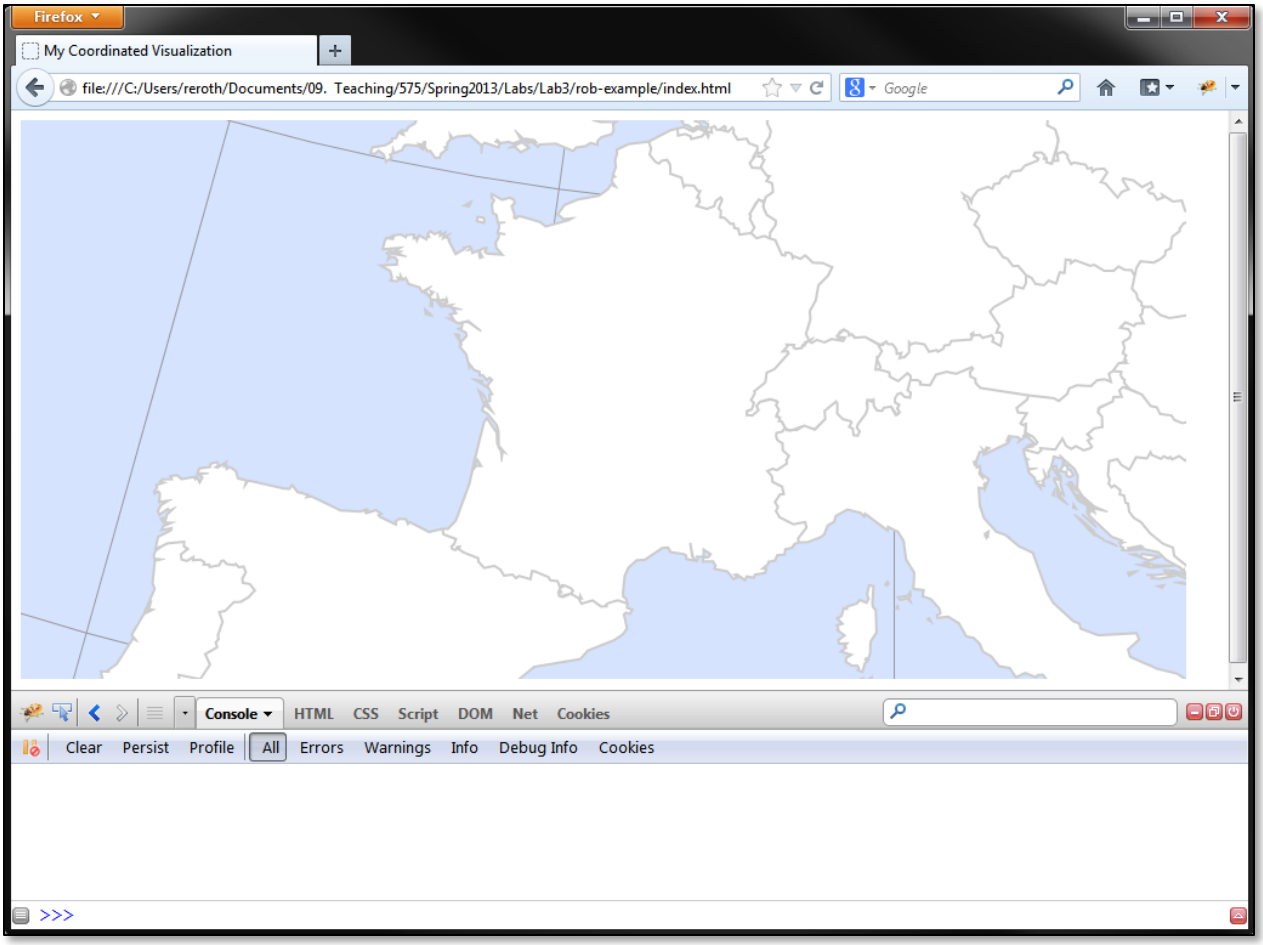

Figure 6. Styling the Basemap.

e. 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 the `append()` function, 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()` and `enter()` functions—much like you did to draw each graticule lines—rather than the simple `append()` function—like you did for drawing the basemap countries and graticule background (**Code Bank 7**). A new `svg` element named `provinces` is created using the `selectAll()` function and each province is added to the map by the `path` generator using the `enter()` function. It is important to note that **Code Bank 7** must be added within the `d3.json()` function, as the TopoJSON must first be processed before adding the `provinces` element.

```
1       //retrieve and process europe json file
2       d3.json("data/europe.json", function(error,europe){
3
4               //add Europe countries geometry to map
5               var countries = map.append("path") //create SVG path element
6                       .datum(topojson.object(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.object(europe,
                                              europe.objects.FranceProvinces).geometries)
13                      .enter() //create elements
14                      .append("path") //append elements to svg
15                      .attr("class", "provinces") //assign class for additional styling
16                      .attr("id", function(d) { return d.properties.adm1_code })
17                      .attr("d", path) //project data as geometry in svg
18
19      });
```

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

Reload your *index.html* file in Firefox; you now should see your enumeration units plotted atop the basemap and graticule with a default black fill (**Figure 7**).

f. Relating Your *.json* and *.csv* 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 uses the `d3.csv()` function and ***AJAX***, or ***Asynchronous JavaScript and XML***, to load and parse *unitsData.csv*. AJAX allows for communication between server (here the simple *.csv*) and the browser without interfering with the content and styles of the display itself (i.e., without refreshing). The `d3.csv()` function loads in the *unitsData.csv* file in the *data* folder (**CB8: 2**) and parses each row into an object using the column headings as ***keys***. You can see this structure in the Firebug console if you add a `console.log()` statement. Because `d3.csv()` is used in an asynchronous manner, the anonymous `function(csvData)` included as the second parameter is a ***callback function***. This means that any script outside of the `d3.csv()` callback (or a `d3.json()` callback) will execute before the script inside of the callback. This can lead to much confusion if you are expecting the callback to add or change a variable that is declared outside of it; in this situation, the variable ends up remaining empty when used. It takes some thinking about the order in which things happen in the script to solve problems like this. Return to the JavaScript Lynda tutorials for additional information about AJAX programming.

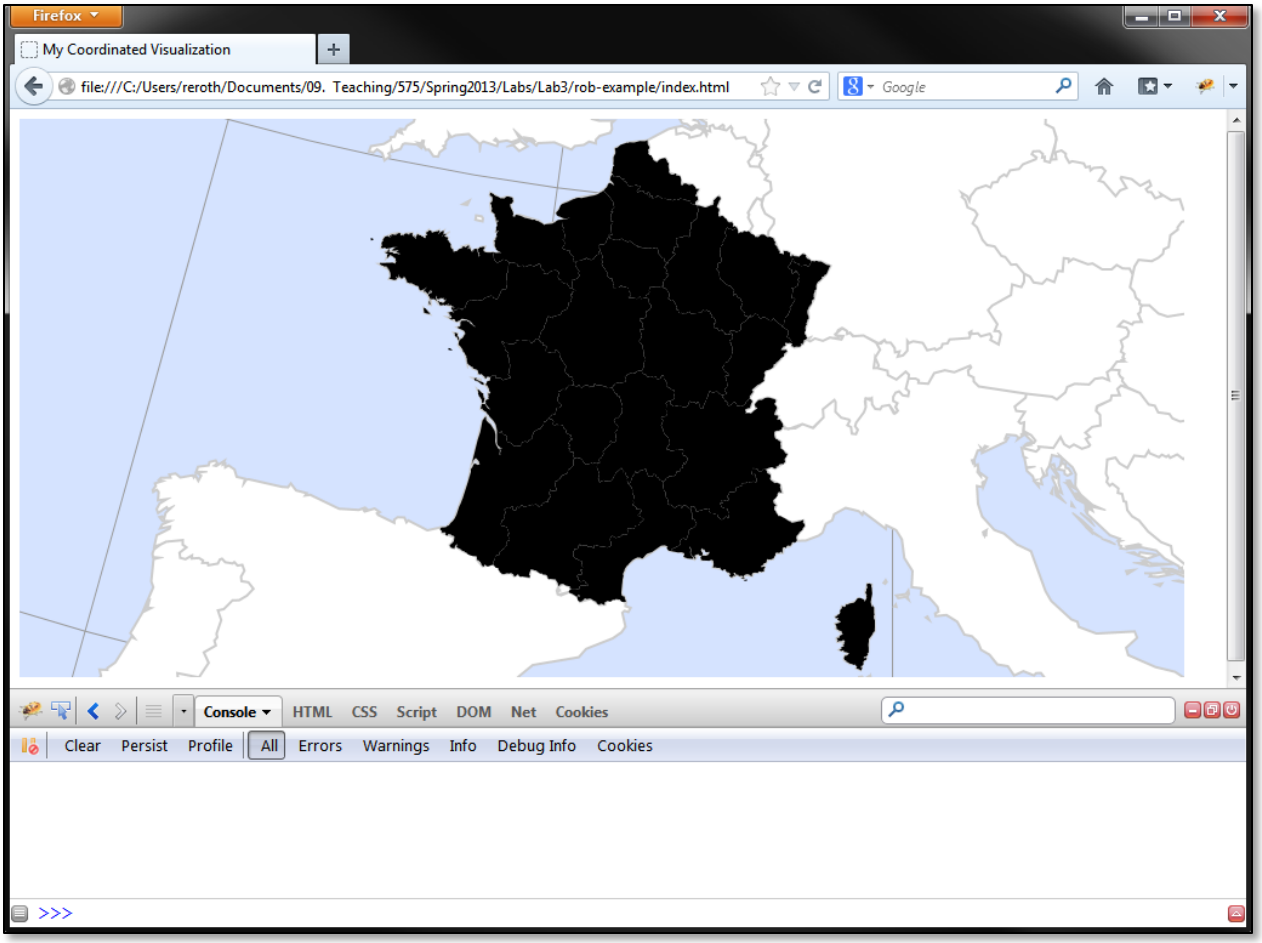

Figure 7. Drawing the Enumeration Units.

The practical upshot here is that in order to attach the multivariate information from the *unitsData.csv* to the geographic information in *europe.json*, you need to nest the two AJAX functions—`d3.json()` and `d3.csv()`—one inside the other. Which AJAX function contains which in the nested hierarchy matters less than making sure that any variables accessing external files are inside of the callback that loads the given file. The `provinces` block from **Code Bank 7** (**10-17**) must be within both callbacks (i.e., the innermost layer) in order for the choropleth to draw (**CB8: 35-49**). It also is within both callbacks, and before the `provinces` block, where you must attach the multivariate information to the `provinces` element through a series of nested loops (**CB8: 11-33**).

Like the `ProcessCSV` prototype function in Labs #1 and #2, loading and parsing external information often is the most challenging and confusing aspect of web map development. Before moving on, add logs to the console line-by-line to inspect how the *.csv* and *.json* contents are being manipulated and combined through the nested AJAX callbacks and the nested for loops. While you need not customize this parser for Lab #3 (aside from file names and instance names), it is likely that you will need to extend or revise this parser, or one you find online, for your final project.

```
1       //retrieve data in csv data file for coloring choropleth
2       d3.csv("data/unitsData.csv", function(csvData){ //callback #1
3
4               //retrieve and process europe json file
5               d3.json("data/europe.json", function(error,europe){ //callback #2
6
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
19                                      //where adm1 codes match, attach csv to json object
20                                      if (jsonProvs[a].properties.adm1_code == csvAdm1){
21
22                                              // assign all five key/value pairs
23                                              for (var b=0; b<keyArray.length; b++){
24                                                      var key = keyArray[b];
25                                                      var val = parseFloat(csvProvince[key]);
26                                                      jsonProvs[a].properties[key] = val;
27                                              };
28
29                                      jsonProvs[a].properties.name = csvProvince.name; //set prop
30                                      break; //stop looking through the json provinces
31                                      };
32                              };
33                      };
34
35                      //add Europe countries geometry to map
36                      var countries = map.append("path") //create SVG path element
37                              .datum(topojson.object(europe,europe.objects.EuropeCountries))
38                              .attr("class", "countries") //assign class for styling countries
39                              .attr("d", path); //project data as geometry in svg
40
41                      //add provinces to map as enumeration units colored by data
42                      var provinces = map.selectAll(".provinces")
43                              .data(topojson.object(europe,
                                                europe.objects.FranceProvinces).geometries)
44                              .enter() //create elements
45                              .append("path") //append elements to svg
46                              .attr("class", "provinces") //assign class for additional styling
47                              .attr("id", function(d) { return d.properties.adm1_code })
48                              .attr("d", path) //project data as geometry in svg
49              });
50      });
```

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

g. Styling Your Choropleth Map

Now that the multivariate information in the *.csv* file is attached to the `provinces` geometry element, 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.

You need to declare two global variables to indicate which variable to represent in the choropleth map (**Code Bank 9**): (1) `keyArray`, which lists the complete set of headers in the *.csv* file, or keys in the `provinces` object, and (2) `expressed`, which indicates the keys currently in use for coloring the choropleth map. These variables must be global (i.e., outside of any function) so that they can be leveraged by both the choropleth map and the PCP. The `keyArray` variable already is declared within the nested AJAX functions (**CB8: 8**); since the keys contained by `keyArray` are strings, they do not need to be inside of the AJAX callback. Be sure that you 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. Then, add the second global variable named `expressed` to indicate the currently selected key for use in the choropleth map. Set the default index to `0`, or the first attribute in the *.csv* file; you later will support multivariate *sequencing* through these keys when implementing the PCP.

```
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*).

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 function are external to the `setMap()` function. The `colorScale()` function (**CB10: 1-20**) 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 the same number of items. Quantile classification is supported natively by D3 through `d3.scale.quantile()`. Importantly, the `colorScale()` function takes the `csvData` object from the `d3.csv()` callback function as a parameter (**CB10: 1**), demonstrating the value of the AJAX solution. Because of this, the call to `colorScale()` must be added as the first line within the `d3.csv()` function definition, before the nested `d3.json()` function definition (**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()` function (**CB10: 3-11**); a five-class, purple color scheme from ColorBrewer is used here. The `domain()` function is added to the `colorScale()` generator so that it is called after determining the minimum and maximum value of the currently expressed key within the `csvData` object (**CB10: 13-17**). The `color` generator is returned to `setMap()` and stored locally in `recolorMap` (**CB10: 19** -> **CB11: 2**).

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 nested AJAX calls in `setMap()` (**CB11: 16-18**). The choropleth function takes two parameters: (1) a `datum` from the *europe.json* `FrancePovinces` object associated with a given province (identified through combined use of `selectAll()` and `enter()`) and (2) the `color` generator, stored locally in the `recolorMap` variable. (**CB11: 17**). The choropleth function identifies the attribute value of the province under investigation (**CB10: 25**) and then checks if a value is valid (**CB10: 27**). If the value exists, class color associated with that value's quantile is returned (**CB10: 28**); if it does not, a default grey is returned (**CB10: 30**).

```
function colorScale(csvData){

	//create quantile classes with color scale
	var color = d3.scale.quantile() //designate quantile scale generator
		.range([
			"#D4B9DA",
			"#C994C7",
			"#DF65B0",
			"#DD1C77",
			"#980043"
		]);

	//set min and max data values as domain
	color.domain([
		d3.min(csvData, function(d) { return Number(d[expressed]); }),
		d3.max(csvData, function(d) { return Number(d[expressed]); })
	]);

	return color; //return the color scale generator
}

function choropleth(d, recolorMap){

	//get data value
	var value = d.properties[expressed];
	//if value exists, assign it a color; otherwise assign gray
	if (value) {
		return recolorMap(value);
	} else {
		return "#ccc";
	};
};
```

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

```
d3.csv("data/unitsData.csv", function(csvData){
	var recolorMap = colorScale(csvData);

	//retrieve and process europe json file
	d3.json("data/europe.json", function(error,europe){

		//CODE BANK 8 SUPPRESSED FOR SPACE

		var provinces = map.selectAll(".provinces")
			.data(topojson.object(europe,
					europe.objects.FranceProvinces).geometries)
			.enter() //create elements
			.append("path") //append elements to svg
			.attr("class", "provinces") //assign class for styling
			.attr("id", function(d) { return d.properties.adm1_code })
			.attr("d", path) //project data as geometry in svg
			.style("fill", function(d) { //color enumeration units
				return choropleth(d, recolorMap);
			});
	});
});
```

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

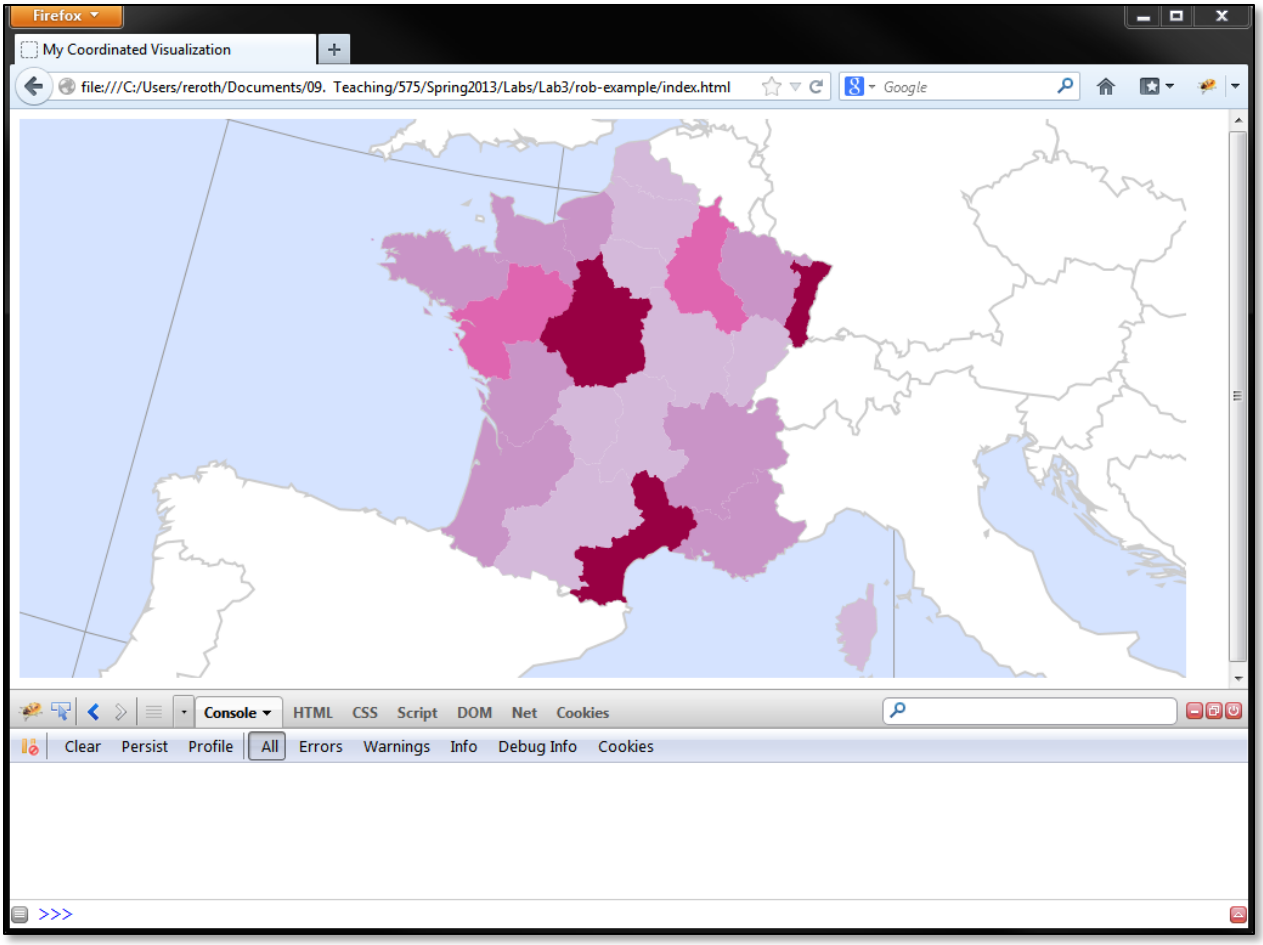

Figure 8. Styling the Choropleth Map.

Refresh *index.html* in Firefox. Bingo! You now have a choropleth map (**Figure 8**)!

h. Implementing *Retrieve* in the Choropleth Map

With the map drawing properly, you now can implement the *retrieve* operator to acquire attribute values from the enumeration units. Like Leaflet, D3 uses `.on()` as the primary method for adding event listeners. Implementing the *retrieve* operator includes two components: (1) providing visual feedback about which enumeration unit was probed, described in lecture as ***highlighting***, and (2) a dynamic label that activates to provide details about the probed enumeration unit. The *retrieve* and associated highlighting solution will be coordinated between the map & PCP in subsequent steps.

You need three functions to implement the *retrieve* operator: (1) `highlight()`, which restyles the probed enumeration unit and populates the content for the dynamic label on `mouseover` (**CB12: 13**), (2) `dehighlight()`, which reverts the enumeration unit back to its original color and deactivates the dynamic label on `mouseout` (**CB12: 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` (**CB12: 15**). The event listeners should be added at the end of the `provinces` block in `setMap()` (**CB11: 9-18**) to make each enumeration unit in the choropleth interactive, making sure you update the position of the semi-colon 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 12** appends the original color as a text string in a `desc` SVG element (**CB12: 16-19**). 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()`.

```
1       //CODE BANK 12 SUPPRESSED FOR SPACE
2
3       var provinces = map.selectAll(".provinces")
4               .data(topojson.object(europe,
                        europe.objects.FranceProvinces).geometries)
5               .enter() //create elements
6               .append("path") //append elements to svg
7               .attr("class", "provinces") //assign class for styling
8               .attr("id", function(d) { return d.properties.adm1_code })
9               .attr("d", path) //project data as geometry in svg
10              .style("fill", function(d) { //color enumeration units
11                      return choropleth(d, recolorMap);
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, recolorMap);
19              });
```

Code Bank 12: Adding Event Listeners to `provinces` in `setMap()` (in: *main.js*).

Once adding the event listeners to the `provinces` block of `setMap()`, you then must define the event handler functions. First, add a new function named `highlight()`, which should be defined outside of `setMap()` (**CB13: 1-19**). The `highlight()` function receives the `data` object associated with highlighted enumeration unit as the parameter. A second function named `datatest()` allows for the use of two different formats of the `data` parameter, which will be necessary to apply `highlight()` and `dehighlight()` to the linked visualization you build later (**CB13: 21-27**). Once formatted, the `data` object is stored in a local variable named `props` (**CB13: 3**). The `highlight()` function then calls `d3.select()` to find the SVG path for the province, using the `adm1_code`, and changes its `fill` style to black (**CB13: 5-6**). You are encouraged to consider the other highlighting solutions discussed in class and to adjust your code accordingly.

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`) (**CB13: 8-9**). 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 (**CB13: 11-19**). A child `div` named `labelname` is added to `infolabel` to position the province name within the label.

```
1       function highlight(data){
2
3               var props = datatest(data);  //standardize json or csv data
4
5               d3.select("#"+props.adm1_code) //select the current province in the DOM
6                       .style("fill", "#000"); //set the enumeration unit fill to black
7
8               var labelAttribute = "<h1>"+props[expressed]+
                                        "</h1><br><b>"+expressed+"</b>"; //label content
9               var labelName = props.name; //html string for name to go in child div
10
11              //create info label div
12              var infolabel = d3.select("body").append("div")
13                      .attr("class", "infolabel") //for styling label
14                      .attr("id", props.adm1_code+"label") //for label div
15                      .html(labelAttribute) //add text
16                      .append("div") //add child div for feature name
17                      .attr("class", "labelname") //for styling name
18                      .html(labelName); //add feature name to label
19      };
20
21      function datatest(data){
22              if (data.properties){ //if json data
23                      return data.properties;
24              } else { //if csv data
25                      return data;
26              };
27      };
```

Code Bank 13: Highlighting the Choropleth Map (in: *main.js*).

Add style rules to the dynamic label in *style.css*, using the `infolabel` and `labelname` class identifiers. **Code Bank 14** 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 (**CB14: 17-26**), as these are used in the `labelAttribute` html string (**CB13: 8**); you are not limited to these tags in the design of your dynamic label. Again, you are encouraged to modify these styles to match the overall design of your map.

Reload your *index.html* page in Firefox and inspect your work. At this point, mousing over an individual enumeration unit should result in highlighting the unit and *retrieving* the associated attribute value (**Figure 9a**). There are two problems with this implementation, however. First, the enumeration units do not return to their original color on `mouseout`, as you have yet to implement the `dehighlight()` function (note the Firefox console error). Use what you have learned up to this point to define the `dehighlight()` function on your own. The `dehiglight()` 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()` function in D3 to learn how to remove an element (the dynamic label) from the DOM.

Second, the dynamic label, while updating on `mouseover`, is positioned away from the map itself. To make the dynamic label follow the user's mouse cursor, first change the positioning of all `svg` elements to absolute in *style.css* (**CB14: 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 15**). The `moveLabel()` function uses the `d3.event()` function 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 (**CB15: 3-4**).

```
svg {
	position: absolute;
}

/*CODEBANK 4 & 6 SUPPRESSED FOR SPACE*/

.infolabel {
	position: absolute;
	width: 200px;
	height: 50px;
	color: #fff;
	background-color: #000;
	border: solid thin #fff;
	padding: 5px;
}

.infolabel h1 {
	margin: 0;
	padding: 0;
	display: inline-block;
	line-height: 1em;
}

.infolabel b {
	float: left;
}

.labelname {
	display: inline-block;
	float: right;
	margin: -25px 0px 0px 40px;
	font-size: 1em;
	font-weight: bold;
	position: absolute;
}
```

Code Bank 14: Styling the Dynamic Label (in: *style.css*).

```
function moveLabel() {

	var x = d3.event.clientX+10; //horizontal label coordinate
	var y = d3.event.clientY-75; //vertical label coordinate

	d3.select(".infolabel") //select the label div for moving
		.style("margin-left", x+"px") //reposition label horizontal
		.style("margin-top", y+"px"); //reposition label vertical
};
```

Code Bank 15: Styling the Dynamic Label (in: *main.js*).

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**)!

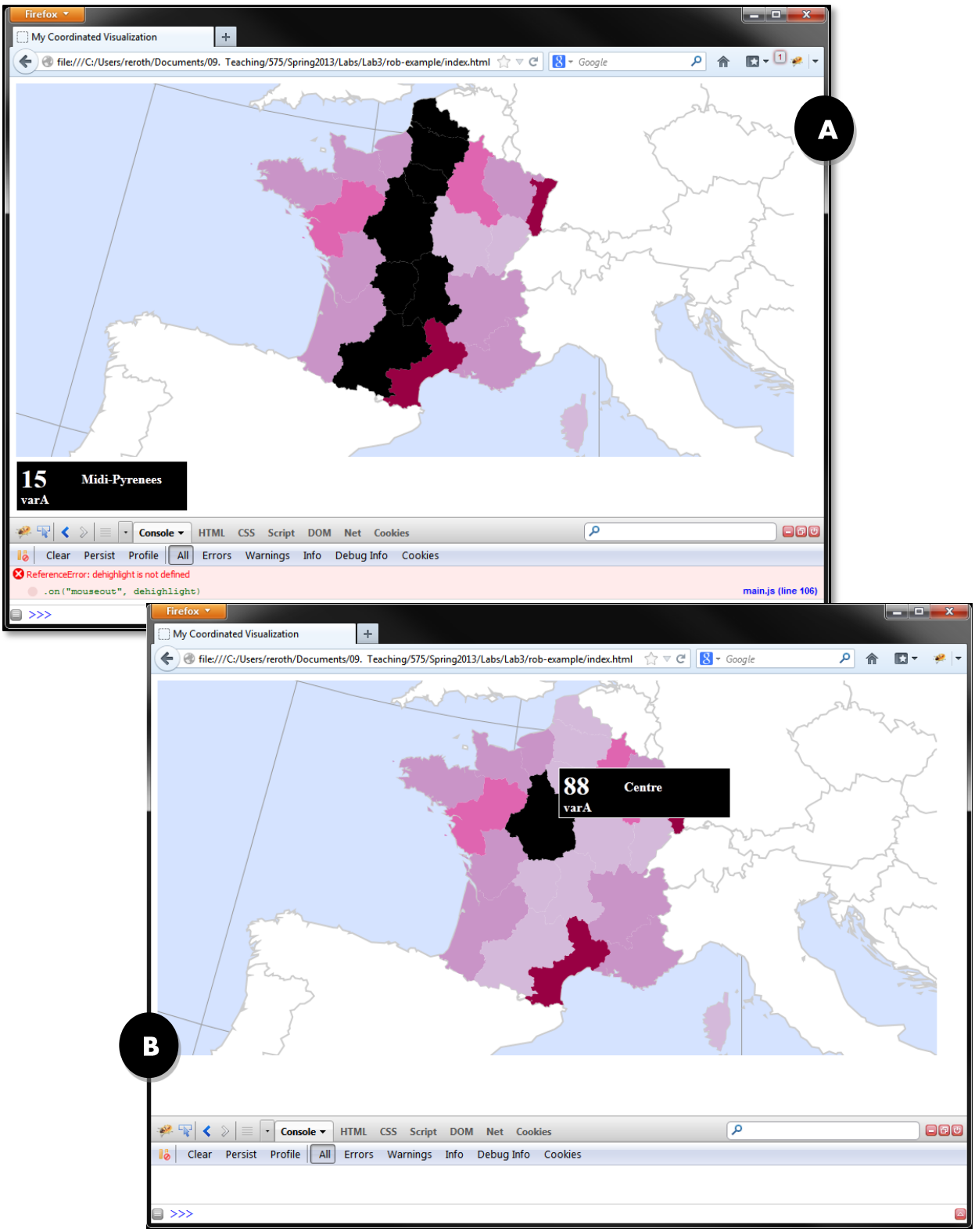

Figure 9. Implementing *Retrieve* in the Choropleth Map.

3. The Parallel Coordinate Plot View

a. Coordinated Visualization with a PCP

After implementing the choropleth view, it is time to turn your attention to the parallel coordinate plot view, and coordination of interaction between the views. As introduced in lecture, a ***parallel coordinate plot***, or ***PCP***, supports visualization of multivariate information. Unlike a scatterplot, which juxtaposes a pair of coordinates orthogonally to one another, a PCP places three or more attribute axes parallel to one another. A unique line then is "threaded" through the axes for each item in the information set, producing an attribute signature of the information item that can be linked with the spatial signature shown in a map view. The following PCP implementation draws from the PCP example posted by Jason Davies.

While you must implement the PCP plot for Lab #3, you will receive +5 bonus points for each additional view you implement, if the interactions are coordinated with the map and PCP views. Example code for implementing a variety of alternative visualizations is provided in the D3 Gallery.

b. Drawing the PCP

As with the `setMap()` function for the choropleth view, create a new function named `drawPCP()` for the PCP view (**Code Bank 16**). The `drawPCP()` function is complex, including multiple steps to create the `svg` elements containing the axes and lines, to scale the axes and place the lines according to the multivariate information in the *.csv* file, and to coordinate interactions with the map view. The broader `drawPCP()` function is divided into a series of code banks, with each conceptual step treated individually; return to the choropleth map description for details about D3 terminology and functionality, as it is not re-introduced here. The `drawPCP()` function should be called within the `d3.csv()` callback in `setMap()` to ensure the *.csv* file has loaded first.

The PCP view needs a second `svg` element different from the choropleth map, so begin by designating its `width` and `height` in local variables. The `width` does not need to match that of the choropleth map, but does in the **Code Bank 16a** example.

```
1       function drawPcp(csvData){
2
3               //pcp dimensions
4               var width = 960;
5               var height = 200;
```

Code Bank 16a: Setting the PCP `svg` Dimensions (in: *main.js*).

Next, prepare the creation of the vertical axes representing each attribute or "coordinate" (**Code Bank 16b**). You need to create an array of coordinate names (name `attributes`) from the key names (stored temporarily in `keys`) within the `csvData` object (**CB16b: 5-16**). Depending on how many non-variable columns are included in your *.csv* file, you may need to adjust the initial index position used to convert `keys` into `attributes` (**CB16b: 14**); the index position of `i=3` is used in **Code Bank 16b** so that the first three columns in **Figure 1** are not included as coordinates in the PCP.

```
5             //create attribute names array for pcp axes
6             var keys = [], attributes = [];
7
8             //fill keys array with all property names
9             for (var key in csvData[0]){
10                  keys.push(key);
11            };
12
13            //fill attributes array with only the attribute names
14            for (var i=3; i<keys.length; i++){
15                  attributes.push(keys[i]);
16            };
```

Code Bank 16b: Preparing an Array of Coordinates for the PCP (in: *main.js*).

Then, set the positioning of the axes using the `d3.scale.ordinal()` generator function (**Code Bank 16c**). This function is an instance of D3's `scale` prototype, somewhat like `d3.scale.quantile()` scale that you used to create your choropleth color scheme. However, unlike the continuous numerical domain of the `quantile()` scale, the `ordinal()` scale deals with a discrete set of items, in this case your attribute categories. The `domain()` function horizontally spaces each axis in the `attributes` string evenly (**CB16c: 19**) and the `rangePoints()` function splits each attribute evenly along a continuum between a numerical start value and end value (**CB16c: 20**); use of the width variable as the end value stretches the axes along the width of the containing `div`. An alternative method provided by D3, `range()`, associates each domain value to a specific range value given in an associated array. Compare the documentation for the `ordinal()` scale to that provided for the `quantile()` color scale, as understanding the difference is useful to understanding how D3 handles data.

```
17            //create horizontal pcp coordinate generator
18            var coordinates = d3.scale.ordinal() //create an ordinal axis scale
19                  .domain(attributes) //horizontally space each axis evenly
20                  .rangePoints([0, width]); //set the horizontal width to svg
```

Code Bank 16c: Creating the `coordinates` Generator(in: *main.js*).

Next, create two generators named `axis` and `line` to draw the PCP axes and lines. Make use of the `d3.svg.axis()` generator function to create a generator (`axis`) for the coordinate axes (**CB16d: 21-22**). Calling `orient("left")` on the generator function sets each axis to a vertical orientation. Yet a third kind of D3 scale, `d3.scale.linear()`, creates a generator to make a visible linear scale (i.e. 'ruler') for each axis (**CB16d: 24-32**). The `d3.scale.linear()` creates the `scales` generator based on the minimum and maximum attribute values of all enumeration units (**CB16d: 28-30**). The scale `range()` is set as the `height` of the `svg` container. The result of this block is an object named `scales` that holds five unique scale generators--one for each coordinate axis. Finally, the `d3.svg.line()` generator function creates a simple generator named `line` for adding the attribute signatures of the enumeration units atop the scaled coordinates.

```
21          var axis = d3.svg.axis() //create axis generator
22                .orient("left"); //orient generated axes vertically
23
24          //create vertical pcp scale
25          scales = {}; //object to hold scale generators
26          attributes.forEach(function(att){ //for each attribute
27                scales[att] = d3.scale.linear() //create a linear scale generator
28                      .domain(d3.extent(csvData, function(data){
29                            return +data[att]; //create array of extents
30                      }))
31                      .range([height, 0]); //set the axis height to SVG height
32          });
33
34          var line = d3.svg.line(); //create line generator
```

Code Bank 16d: Creating `axis` and `line` Generators(in: *main.js*).

With the generators in place, you now can create new `svg` elements for the PCP (**Code Bank 16e**). The `pcplot` block creates an `svg` container for the PCP based on the dimensions previously designated (**CB16e: 38-39**) and gives it a class named `pcplot` for style adjustments *in style.css* (**CB16e: 40**). It also appends a child container element (`g`) to hold all of the line geometry (**CB16e: 41**), and mathematically transforms the child container using `d3.transform()` to make it smaller than the outer boundaries of the `svg` container (**CB16e: 42-44**). More information about this mathematical transformation is available at: http://www.w3.org/TR/SVG/coords.html. The `pcpBackground` block appends a `rect` (rectangle) to `pcplot` for styling the background of the PCP (**CB16e: 46-53**). Make sure to style the rectangle using the class name `pcpBackground` in *styles.css* (**CB17: 1-10**). For more on the `rect` element, see https://developer.mozilla.org/en-US/docs/SVG/Element/rect.

```
35          //create a new svg element with the above dimensions
36          var pcplot = d3.select("body")
37                .append("svg")
38                .attr("width", width)
39                .attr("height", height)
40                .attr("class", "pcplot") //for styling
41                .append("g") //append container element
42                .attr("transform", d3.transform( //change the container size/shape
43                      "scale(0.8, 0.6),"+ //shrink
44                      "translate(96, 50)")); //move
45
46          var pcpBackground = pcplot.append("rect") //background for the pcp
47                .attr("x", "-30")
48                .attr("y", "-35")
49                .attr("width", "1020")
50                .attr("height", "270")
51                .attr("rx", "15")
52                .attr("ry", "15")
53                .attr("class", "pcpBackground");
```

Code Bank 16e: Adding the PCP Container and Drawing the Background (in: *main.js*).

Once the `pcpBackground` rectangle is drawn, you then can draw the lines within the `pcpLines` element. Use your understanding of JavaScript and D3 to interpret the logic in **Code Bank 16f**. Compare the drawing of PCP lines in **Code Bank 16f** with the drawing of map polygons in **Code Bank 12** to see how `svg` elements are used similarly to draw both visualizations. The primary difference is use of the `csvData` object (rather than the TopoJSON object) and the `line` generator, which creates a new line for each `datum` in the `csvData` object. The `map()` method of D3 (used here in a non-cartography way) iterates over each attribute in your `attributes` array and generates an object consisting of (x,y) coordinate arrays of length=2, with each array representing the position of a single PCP line in a single PCP axis (**CB16f: 65-67**).

```
54            //add lines
55            var pcpLines = pcplot.append("g") //append a container element
56                  .attr("class", "pcpLines") //class for styling lines
57                  .selectAll("path") //prepare for new path elements
58                  .data(csvData) //bind data
59                  .enter() //create new path for each line
60                  .append("path") //append each line path to the container element
61                  .attr("id", function(d){
62                        return d.adm1_code; //id each line by admin code
63                  })
64                  .attr("d", function(d){
65                        return line(attributes.map(function(att){ //get coordinates
66                              return [coordinates(att), scales[att](d[att])];
67                        }));
68                  });
```

Code Bank 16f: Drawing the Lines (in: *main.js*).

Finally, add the PCP coordinates on top of the PCP lines (**Code Bank 16g**). The `axes` are drawn atop `pcpLines` to improve the ability to see and interact with them; again, remember that the top-bottom order of your code represents the bottom-up drawing of your map. The first five lines of the `axes` block create a container element for each coordinate axis, setting the class name of each axis to the name of the attribute associated with the axis (**CB16g: 70-74**). The `transform` attribute positions each axis container evenly along the coordinate scale using the `scale` generator (**CB16g: 75-77**); `coordinates(d)` refers to the horizontal pixel position of each axis. The `each()` method invokes a function on each axis element (**CB16g: 78-89**). Within this function, the `axis` generator function is called to create the visible path of the coordinate (**CB16g: 80-82**). The `scale()` method invokes the appropriate `scale` generator from the `scales` object to append the `axis` scale, giving the axis the correct height (**CB16g: 80**). D3's `axis.scale()` method automatically creates a scale with tick marks and numbers, so to avoid this you have to specify both the number of ticks and the tick size as `0` (**CB16g: 81-82**). You also want to give each axis container an `id` so you can change the width of each axis individually when that axis is selected (**CB16g: 84**). Set an initial width for all of the axes (e.g., `5px`) (**CB16g: 85**). The click listener calls a function called `sequence()` to update the variable portrayed in the choropleth map (**CB16g: 86-88**); this function currently is commented out, as the *sequence* operator is treated a subsequent step.

The final block sets a width (e.g., `10px`) for the axis that corresponds to the `expressed` attribute (**CB16g: 91-92**). Before moving onto the next step, add appropriate styles for your lines and axes in *style.css* (**Code Bank 17: 12-23**). Reload your *index.html* page; you should see a PCP (**Figure 10**)!

```
69          //add axes
70          var axes = pcplot.selectAll(".attribute") //prepare for new elements
71                .data(attributes) //bind data (attribute array)
72                .enter() //create new elements
73                .append("g") //append elements as containers
74                .attr("class", "axes") //class for styling
75                .attr("transform", function(d){
76                      return "translate("+coordinates(d)+")"; //position axes
77                })
78                .each(function(d){ //invoke the function for each axis
79                      d3.select(this) //select the current axis container element
80                            .call(axis.scale(scales[d]) //generate the scale
81                                  .ticks(0) //no ticks
82                                  .tickSize(0) //no ticks, I mean it!
83                            )
84                      .attr("id", d) //assign the attribute name as the axis id
85                      .style("stroke-width", "5px") //style each axis
86                      .on("click", function(){ //click listener
87                            // sequence(this, csvData);
88                      });
89                });
90
91          pcplot.select("#"+expressed) //select the expressed attribute's axis
92                .style("stroke-width", "10px");
93    };
```

Code Bank 16g: Drawing the Axes (in: *main.js*).

```
1     .pcplot {
2           margin-top: 470px;
3     }
4
5     .pcpBackground {
6           fill: #ccc;
7           opacity: 0.5;
8           stroke: #000;
9           stroke-width: 4px;
10    }
11
12    .pcpLines path {
13          fill: none;
14          stroke-width: 4px;
15          stroke: #1e90ff;
16    }
17
18    .axes path {
19          fill: none;
20          stroke: #000;
21          stroke-opacity: 0.5;
22          stroke-linecap: round;
23    }
```

Code Bank 17: Styling the PCP View (in: *style.css*).

With the choropleth and PCP views drawn, you now should have a good general impression on how to implement views in D3:

(1) Designate your input parameters (such as dimensions and arrays of data values);
(2) Designate generator functions you will need to draw your `svg` elements;
(3) Select the DOM elements you intend to fill or create (even if they do not exist yet);
(4) Bind `data` to `svg` elements, `enter()` (if multiple elements), and `append()` those elements;
(5) Set the attributes and styles of the `svg` elements, using the generator functions you previously designated or in *style.css*.

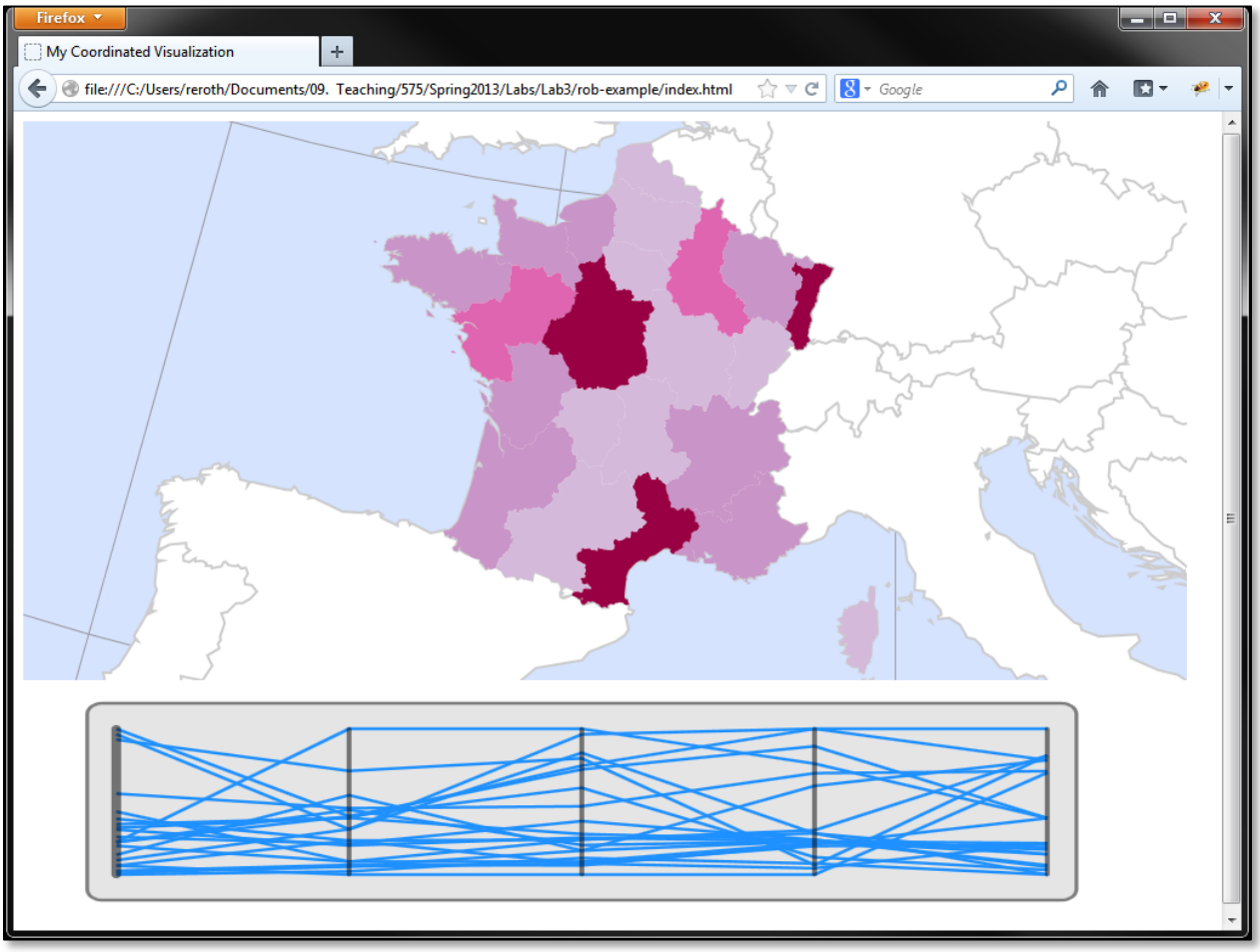

Figure 10: Drawing the PCP View

c. Coordinating *Retrieve* between the Choropleth & PCP

After drawing your PCP, you now can implement the *retrieve* operator for the PCP and coordinate this operator between the map and PCP. 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.

Coordination of the *retrieve* operator across views is completed in two steps. First, add event listeners onto the `pcpLines` element for detecting `mouseover`, `mouseout`, and `mousemove` events atop any of the lines in the PCP (**CB18: 16-18**). The event handers for these three functions are the same as those used to handle the *retrieve* operator : `highlight()`, `dehighlight()`, and `moveLabel()`, respectively (see **CB12: 13-15**). It is through use of the same event handler functions that operators are coordinated: regardless of the view receiving the interaction, the same function updating both views is called. Importantly, because you are chaining additional methods to the end of the `pcpLines` element, you need to change the position of the semi-colon (e.g., from **CB18: 15** to **CB18: 18**).

```
1            //add lines
2            var pcpLines = pcplot.append("g") //append a container element
3                   .attr("class", "pcpLines") //class for styling lines
4                   .selectAll("path") //prepare for new path elements
5                   .data(csvData) //bind data
6                   .enter() //create new path for each line
7                   .append("path") //append each line path to the container element
8                   .attr("id", function(d){
9                         return d.adm1_code; //id each line by admin code
10                  })
11                  .attr("d", function(d){
12                        return line(attributes.map(function(att){ //get coordinates
13                              return [coordinates(att), scales[att](d[att])];
14                        }));
15                  })
16                  .on("mouseover", highlight)
17                  .on("mouseout", dehighlight)
18                  .on("mousemove", moveLabel);
19
```

Code Bank 18: Implementing *Retrieve* on the PCP (in: *main.js*).

Refresh your *index.html* in Firefox. You'll notice that brushing a PCP line now evokes the *retrieve* operator, placing the dynamic label atop the PCP and highlighting the associated enumeration unit in the choropleth map. What you should notice, however, is that the PCP lines themselves do not change their styling and that brushing of the choropleth map does not highlight the associated PCP line. The second step in coordinating the *retrieve* interaction is modifying the `highlight()` and `dehighlight()` event handler functions so that they restyle both the enumeration unit in the map and the line in the PCP. You do not need to update `moveLabel()`, as it repositions the dynamic label to any probed location in the `body` element (i.e., across both the choropleth map and PCP views).

Logic for highlighting the PCP lines is provided in **Code Bank 19**. Notice that `highlight()` function still modifies the styling of the proper enumeration unit using its `adm1_code` (**CB 19: 3-7**) and still populates the `infolabel` element, again using the `adm1_code` (**CB 19: 13-23**). However, now an intermediate block has been added to adjust the styling of the proper PCP line, once again making use of the `adm1_code` (**CB 19: 8-11**). Notice that a different highlighting solution is applied for the map (**CB 19: 6**) versus the PCP (**CB 19: 11**). Return to your lecture notes to consider a better solution for highlighting that maintains highlighting consistency.

```
1       function highlight(data){
2
3               var props = datatest(data);  //standardize json or csv data
4
5               d3.select("#"+props.adm1_code) //select the current province in the DOM
6                       .style("fill", "#000"); //set the enumeration unit fill to black
7
8               //highlight corresponding pcp line
9               d3.selectAll(".pcpLines") //select the pcp lines
10                      .select("#"+props.adm1_code) //select the right pcp line
11                      .style("stroke","#ffd700"); //restyle the line
12
13              var labelAttribute = "<h1>"+props[expressed]+
                                     "</h1><br><b>"+expressed+"</b>"; //label content
14              var labelName = props.name; //html string for name to go in child div
15
16              //create info label div
17              var infolabel = d3.select("body").append("div")
18                      .attr("class", "infolabel") //for styling label
19                      .attr("id", props.adm1_code+"label") //for label div
20                      .html(labelAttribute) //add text
21                      .append("div") //add child div for feature name
22                      .attr("class", "labelname") //for styling name
23                      .html(labelName); //add feature name to label
24      };
```

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

Finally, modify your custom `dehighlight()` function to change the PCP lines back to their original color on `mouseout`. Refresh your *index.html* page and preview in Firefox. You now have made a coordinated visualization!

d. Implement *Sequence* across Attributes

The final step in implementing the PCP view is adding support of the *sequence* operator. In Lab #1 and #2, the user was able to *sequence* by time, with controls supporting animation through the timestamps or jumping to a single timestamp. For Lab #3, the user still needs to *sequence*, but instead through the set of attributes in the multivariate information set (i.e., to adjust which attribute is represented in the choropleth map). For Lab #3, the axes in the PCP view double as a direct manipulation widget supporting the *sequence* operator. When the user clicks on an axis, the choropleth map will update showing the spatial distribution of values for the associated variable (whereas mousing over a line retrieves the associated information item). Thus, *sequence* is coordinated, but can be evoked only from the PCP.

Implement the *sequence* operator by first uncommenting the call to the `sequence()` method that is called on a `click` event (**CB16g: 87-89**). Then, define the `sequence()` function at the bottom of *main.js* (**Code Bank 20**). The `sequence()` function is passed the `axis` element clicked by the user, along with the complete `csvData` object (**CB20: 1**). When the user clicks an axis, the function changes the `width` of all axes to the default `5px`, then increases the width of the `axis` passed as a parameter (**CB20: 3-7**). The global variable `expressed` then is updated to reflect the newly

selected attribute in the PCP (CB20: 3-7). Updating the `expressed` variable ensures that the dynamic label will draw from the selected attribute upon *retrieve* (CB19: 13). The final block restyles the map according to the newly selected attribute and changes each enumeration unit's `desc` element to make the `highlight()` and `dehighlight()` functions update the choropleth styling correctly (CB20: 11-19).

```
1       function sequence(axis, csvData){
2
3               //restyle the axis
4               d3.selectAll(".axes") //select every axis
5                       .style("stroke-width", "5px"); //make them all thin
6
7               axis.style.strokeWidth = "10px"; //change selected axis thickness
8
9               expressed = axis.id; //change the class-level attribute variable
10
11              //recolor the map
12              d3.selectAll(".provinces") //select every province
13                      .style("fill", function(d) { //color enumeration units
14                              return choropleth(d, colorScale(csvData)); //->
15                      })
16                      .select("desc") //replace the text in each province's desc element
17                      .text(function(d) {
18                              return choropleth(d, colorScale(csvData)); //->
19                      });
20      };
```

Code Bank 20: Implementing the *Sequence* Operator (in: *main.js*).

Reload your *index.html* file in Firefox and see the fruits of your labor (Figure 11). You now can change the variable shown in the map interactively by click on a PCP axis! Take a minute to click through all axes and information items to make sure that all information is being displayed properly.

Note that no affordance is provided to indicate that the axes themselves are interactive. On your own, add two additional components to the PCP: (1) additional event listeners atop the axes to restyle the axis on mouseover and mouseout and (2) add text labels for the axes to alert to the user the names of the variables.

e. Transition What You've Learned: Be Awesome

You Lab #3 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 #3 requirements.

You are encouraged to continue to add functionality to your Lab #3 visualization; you will receive +5 bonus points for each additional representation and each additional interaction operator, and +2 for implementing a legend for the choropleth map.

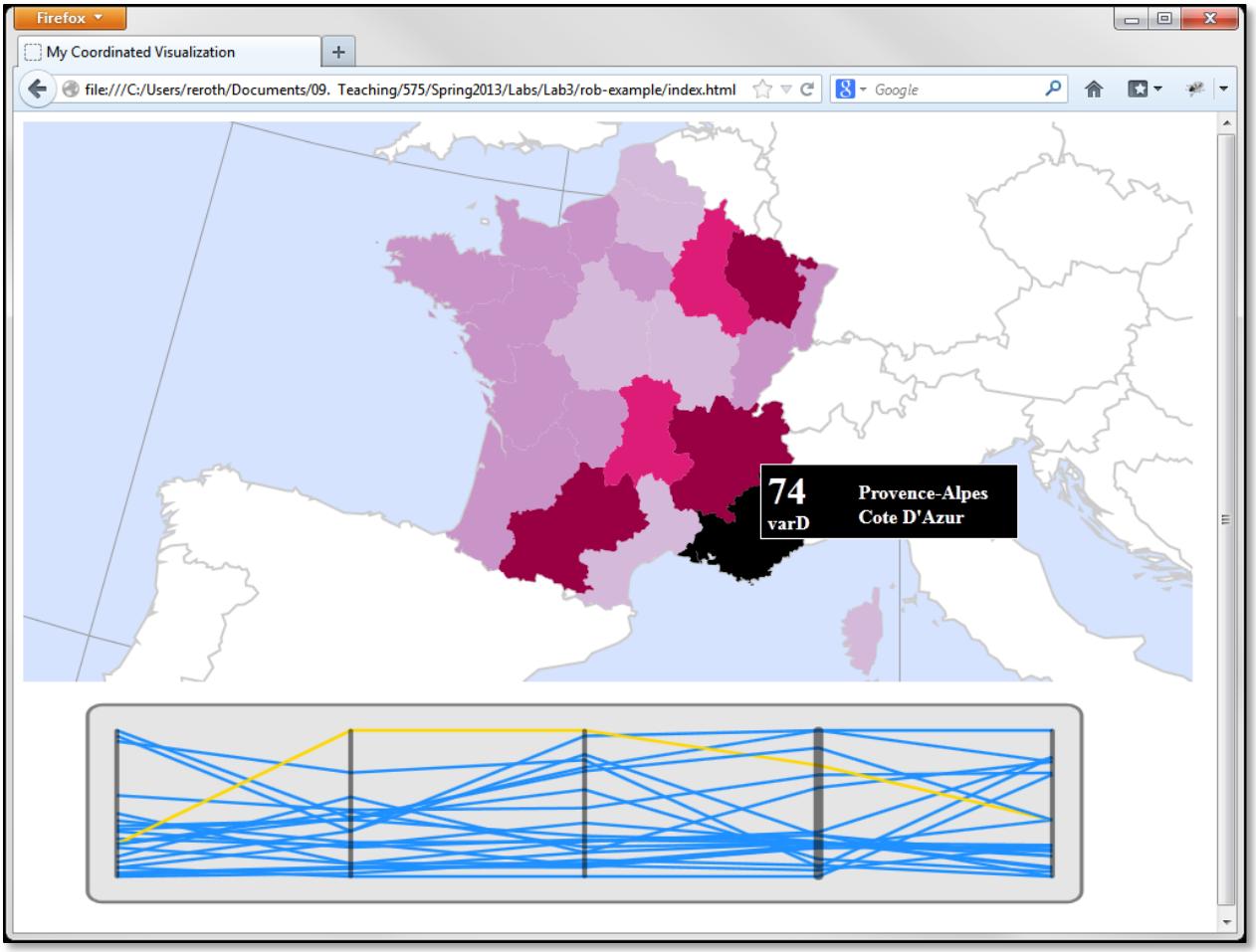

Figure 11: Implementing the *Sequence* Operator

Evaluation Rubric: Interaction Challenge (40pts)

Delivery: You are required to publish a version of your map to your webspace AND upload a *.zip* of your entire directory to the Learn@UW Lab #2 Dropbox at least one hour before your lab on **April 18th, 2013**. While you may receive bonus points by implementing additional views, you cannot exceed 40 points overall on this assignment.

Representation: Choropleth Map View (10)

(7) Basemap: Projection, Generalization, Visual Hierarchy, Etc.
(1) Choropleth Normalization
(1) Choropleth Color Scheme
(1) Choropleth Classification

Representation: PCP View (10)

(6) Coordinate Drawing, Position, and Scaling
(2) PCP Line Legibility
(2) PCP Coordinate Labels

Coordinated Interaction: *Sequence* (7)

(5) Working Coordination
(2) Affordances/Feedback

Coordinated Interaction: *Retrieve* (9)

(5) Working Coordination
(2) Affordances/Feedback
(2) Information Window Design & Content

Design for Scenario (4)

(2) Design Clarity and Style
(2) Overall Consideration of Scenario

(+5) Each additional coordinated view
(+5) Each additional operator coordinated across views
(+2) Choropleth Legend