DOI: 10.14714/CP78.1278

Interactive and Multivariate Choropleth Maps with D3

Carl M. Sack
University of Wisconsin–Madison
cmsack@wisc.edu

Richard G. Donohue
University of Kentucky
rgdonohue@uky.edu

Robert E. Roth
University of Wisconsin–Madison
reroth@wisc.edu

ASSUMED SKILLS AND LEARNING OUTCOMES

The following tutorial describes how to make an interactive choropleth map using the D3 (Data-Driven Documents) web mapping library (**d3js.org**). This tutorial is based on a laboratory assignment created in the fall of 2014 for an advanced class titled Interactive Cartography and Geovisualization at the University of Wisconsin–Madison. This is the second of two *On the Horizon* tutorials on web mapping and extends a previous tutorial that used the Leaflet JavaScript library (see Donohue et al. 2013; dx.doi.org/10.14714/CP76.1248). Fully commented source code for both tutorials is available on GitHub (**github.com/uwcart/cartographic-perspectives**). All code is distributed under a Creative Commons 3.0 license and available for unconditional use, with the exception of the files in the `lib` directory, for which certain license conditions are required as described in the file *LICENSE.txt*.

This tutorial assumes literacy in JavaScript, HTML, and CSS programming for the web. In particular, you should be comfortable with the manipulation of JavaScript arrays and objects. Free tutorials and reference documentation for these languages are available at **www.w3schools.com**. Additionally, D3 makes heavy use of jQuery-style DOM element selection and dot syntax (**jquery.com**). It is further assumed that you are familiar with in-browser development tools such as those provided by Google Chrome (**developers.google.com/chrome-developer-tools**), Mozilla Firefox (**developer.mozilla.org/en-US/docs/Tools**), and Firebug (**getfirebug.com**). An important limitation of D3 is its incompatibility with Microsoft's Internet Explorer browser prior to version 9; use of Internet Explorer below version 10 is not recommended. Finally, the tutorial assumes you have set up a development server running either remotely or as a localhost. MAMP for Mac (**www.mamp.info/en**) and WAMP for Windows (**www.wampserver.com/en**) are useful for this.

After completing the tutorial, you should be able to:

- work with the TopoJSON data format;
- use the D3 library to publish a multivariate choropleth map to the web; and
- implement interactivity, including attribute selection, mouseover highlighting, and dynamic labels.

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 Web Mercator, which is highly distorted at higher latitudes and inappropriate for many forms of data visualization at small map scales. The D3 alternative utilizes vector graphics rendered in the browser for the basemap, 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* (**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 (**trac.osgeo.org/proj4js**).

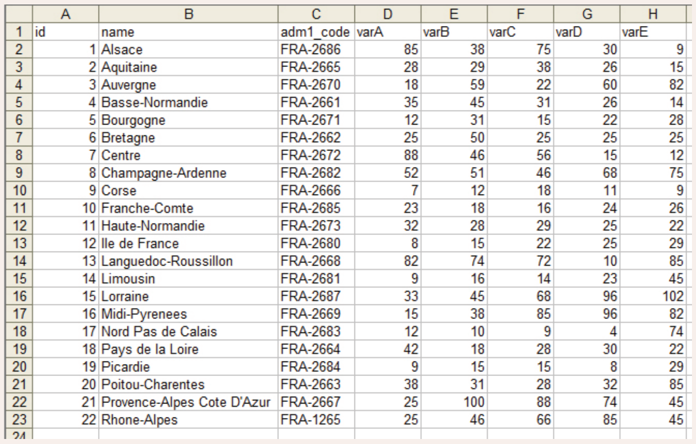
The goal of this tutorial is to provide you with a broad introduction to using D3 for web cartography. The following tutorial extends two excellent online learning resources: (1) Mike Bostock's "Let's Make a Map" tutorial (**bost.ocks.org/mike/map**), and (2) developer Scott Murray's e-book *Interactive Data Visualization for the Web* (**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 appropriate ***multivariate*** (i.e., multiple attributes enumerated over the same set of spaces) information to portray in a choropleth map. Your data should be numerical, with attributes (or fields) organized as separate table columns and each enumeration unit (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 regions 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.

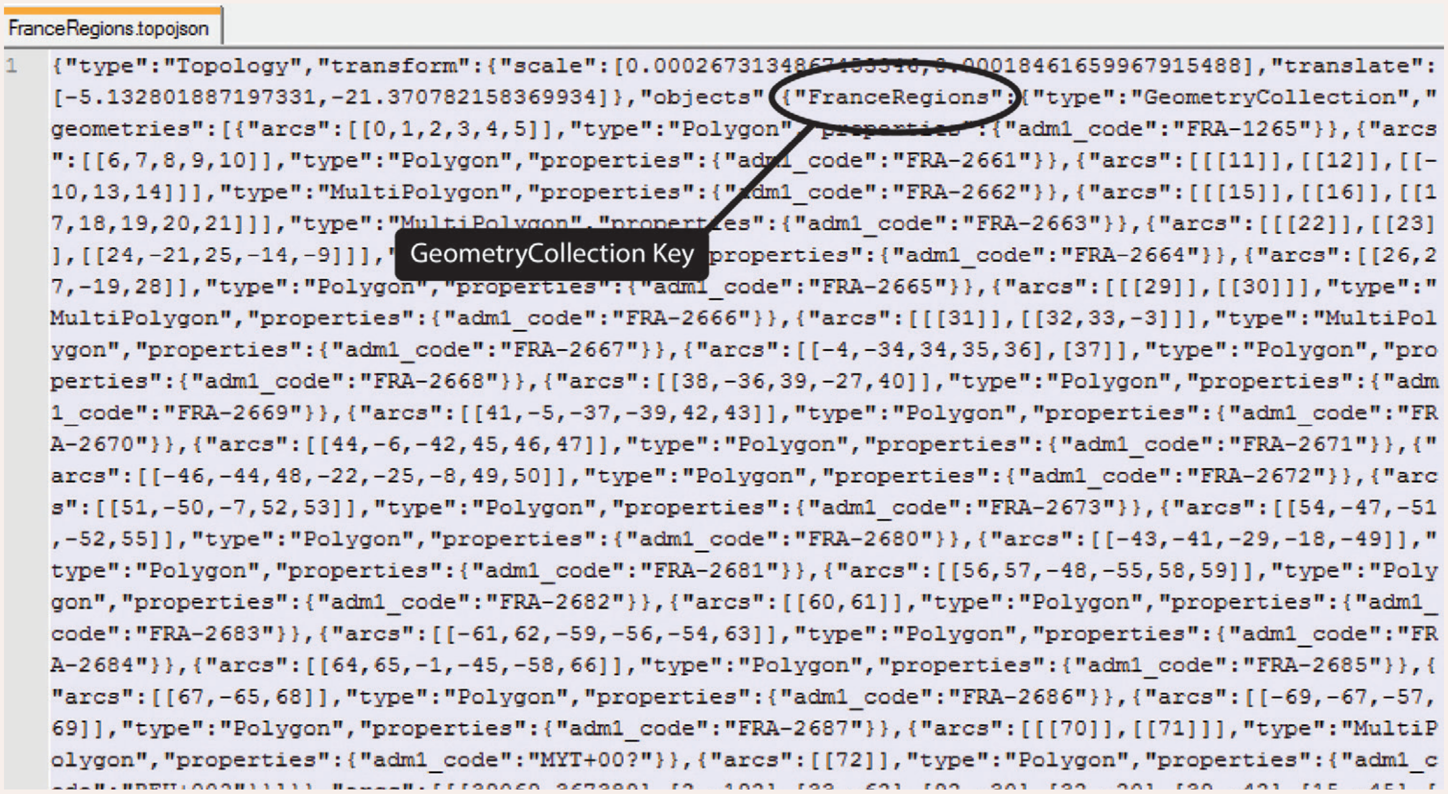


	A	B	C	D	E	F	G	H
1	id	name	adm1_code	varA	varB	varC	varD	varE
2	1	Alsace	FRA-2686	85	38	75	30	9
3	2	Aquitaine	FRA-2665	28	29	38	26	15
4	3	Auvergne	FRA-2670	18	59	22	60	82
5	4	Basse-Normandie	FRA-2661	35	45	31	26	14
6	5	Bourgogne	FRA-2671	12	31	15	22	28
7	6	Bretagne	FRA-2662	25	50	25	25	25
8	7	Centre	FRA-2672	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 (**github.com/mbostock/topojson/wiki**). TopoJSON is similar to GeoJSON, introduced in the Leaflet tutorial, but it can have significantly reduced file sizes and also stores ***topology***: 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, and a transform equation that situates the nodes in the coordinate reference system. 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 **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 datasets used in this tutorial (*EuropeCountries.topojson* and *FranceRegions.topojson*, included in the tutorial source code) were prepared using two shapefiles downloaded from the Natural Earth website (**www.naturalearthdata.com**): one with the nations of Europe as a whole, and one with the regions 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),



```
{"type":"Topology","transform":{"scale":[0.0002673134[illegible],0.0001846165996791548 8],"translate":
[-5.132801887197331,-21.370782158369934]},"objects":{"FranceRegions":{"type":"GeometryCollection","
geometries":[{"arcs":[[0,1,2,3,4,5]],"type":"Polygon","properties":{"adm1_code":"FRA-1265"}},{"arcs
":[[6,7,8,9,10]],"type":"Polygon","properties":{"adm1_code":"FRA-2661"}},{"arcs":[[[11]],[[12]],[[-
10,13,14]]],"type":"MultiPolygon","properties":{"adm1_code":"FRA-2662"}},{"arcs":[[[15]],[[16]],[[1
7,18,19,20,21]]],"type":"MultiPolygon","properties":{"adm1_code":"FRA-2663"}},{"arcs":[[[22]],[[23]
],[[24,-21,25,-14,-9]]],"[illegible]properties":{"adm1_code":"FRA-2664"}},{"arcs":[[26,2
7,-19,28]],"type":"Polygon","properties":{"adm1_code":"FRA-2665"}},{"arcs":[[[29]],[[30]]],"type":"
MultiPolygon","properties":{"adm1_code":"FRA-2666"}},{"arcs":[[[31]],[[32,33,-3]]],"type":"MultiPol
ygon","properties":{"adm1_code":"FRA-2667"}},{"arcs":[[-4,-34,34,35,36],[37]],"type":"Polygon","pro
perties":{"adm1_code":"FRA-2668"}},{"arcs":[[38,-36,39,-27,40]],"type":"Polygon","properties":{"adm
1_code":"FRA-2669"}},{"arcs":[[41,-5,-37,-39,42,43]],"type":"Polygon","properties":{"adm1_code":"FR
A-2670"}},{"arcs":[[44,-6,-42,45,46,47]],"type":"Polygon","properties":{"adm1_code":"FRA-2671"}},{"
arcs":[[-46,-44,48,-22,-25,-8,49,50]],"type":"Polygon","properties":{"adm1_code":"FRA-2672"}},{"arc
s":[[51,-50,-7,52,53]],"type":"Polygon","properties":{"adm1_code":"FRA-2673"}},{"arcs":[[54,-47,-51
,-52,55]],"type":"Polygon","properties":{"adm1_code":"FRA-2680"}},{"arcs":[[-43,-41,-29,-18,-49]],"
type":"Polygon","properties":{"adm1_code":"FRA-2681"}},{"arcs":[[56,57,-48,-55,58,59]],"type":"Poly
gon","properties":{"adm1_code":"FRA-2682"}},{"arcs":[[60,61]],"type":"Polygon","properties":{"adm1_
code":"FRA-2683"}},{"arcs":[[-61,62,-59,-56,-54,63]],"type":"Polygon","properties":{"adm1_code":"FR
A-2684"}},{"arcs":[[64,65,-1,-45,-58,66]],"type":"Polygon","properties":{"adm1_code":"FRA-2685"}},{
"arcs":[[67,-65,68]],"type":"Polygon","properties":{"adm1_code":"FRA-2686"}},{"arcs":[[-69,-67,-57,
69]],"type":"Polygon","properties":{"adm1_code":"FRA-2687"}},{"arcs":[[[70]],[[71]]],"type":"MultiP
olygon","properties":{"adm1_code":"MYT+00?"}},{"arcs":[[72]],"type":"Polygon","properties":{"adm1_c
```

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

MapShaper (mapshaper.com), and GeoJSON.io (geojson.io). To reduce file size, all attribute fields were removed from the original shapefiles except *adm1_code* in the regions shapefile and *name_long* in the countries shapefile. The shapefiles were converted into TopoJSON files by first using QGIS to save them as GeoJSON format, then using MapShaper to convert the GeoJSON to TopoJSON files (MapShaper will not preserve attributes if used to directly convert a shapefile to TopoJSON). Note that the original shapefiles must use an unprojected WGS84 coordinate reference system (EPSG:4326) for the resulting TopoJSON files to work with D3.

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 `GeometryCollection` object (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**.

2. PREPARING YOUR DIRECTORY STRUCTURE AND BOILERPLATE HTML

With your *.csv* and TopoJSON (*.topojson*) 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 use a development server or live preview function of your development software to view your website in the browser. 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 *.topojson* files into the *data* folder. Add the boilerplate HTML code provided in **Example 1** to the *index.html file*.

After configuring your directory, acquire three *.js* files from Bostock's GitHub account: (1) *d3.v3.js* (github.com/mbostock/d3), containing the D3 visualization library, (2) *topojson.v1.min.js* for parsing your TopoJSON file (github.com/mbostock/topojson), and (3) *queue.js* (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* (**EX1: 11–13**).

```
<!DOCTYPE HTML>
<html>
      <head>
            <meta charset=”utf-8”>
            <title>My Coordinated Visualization</title>

            <!--main stylesheet-->
            <link rel=”stylesheet” href=”css/style.css” />
      </head>
      <body>
            <!--libraries-->
            <script src=”lib/d3.v3.js”></script>
            <script src=”lib/topojson.v1.min.js”></script>
            <script src=”lib/queue.js”></script>

            <!--link to main JavaScript file-->
            <script src=”js/main.js”></script>
      </body>
</html>
```

Example 1. *Basic HTML5 boiler plate (in:* index.html*).*

3. LOADING YOUR .TOPOJSON FILES INTO THE BROWSER

THE NEXT STEP is loading the geographic information assembled in your .topojson files. 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 are loaded ***asynchronously***—separately from the rest of the script, so the browser does not have to wait for the 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 are 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.

Example 2 provides the logic needed to initialize the webpage and print the two .topojson files used in this example to the console. The `queue()` method (**EX3: 12–16**), 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**).

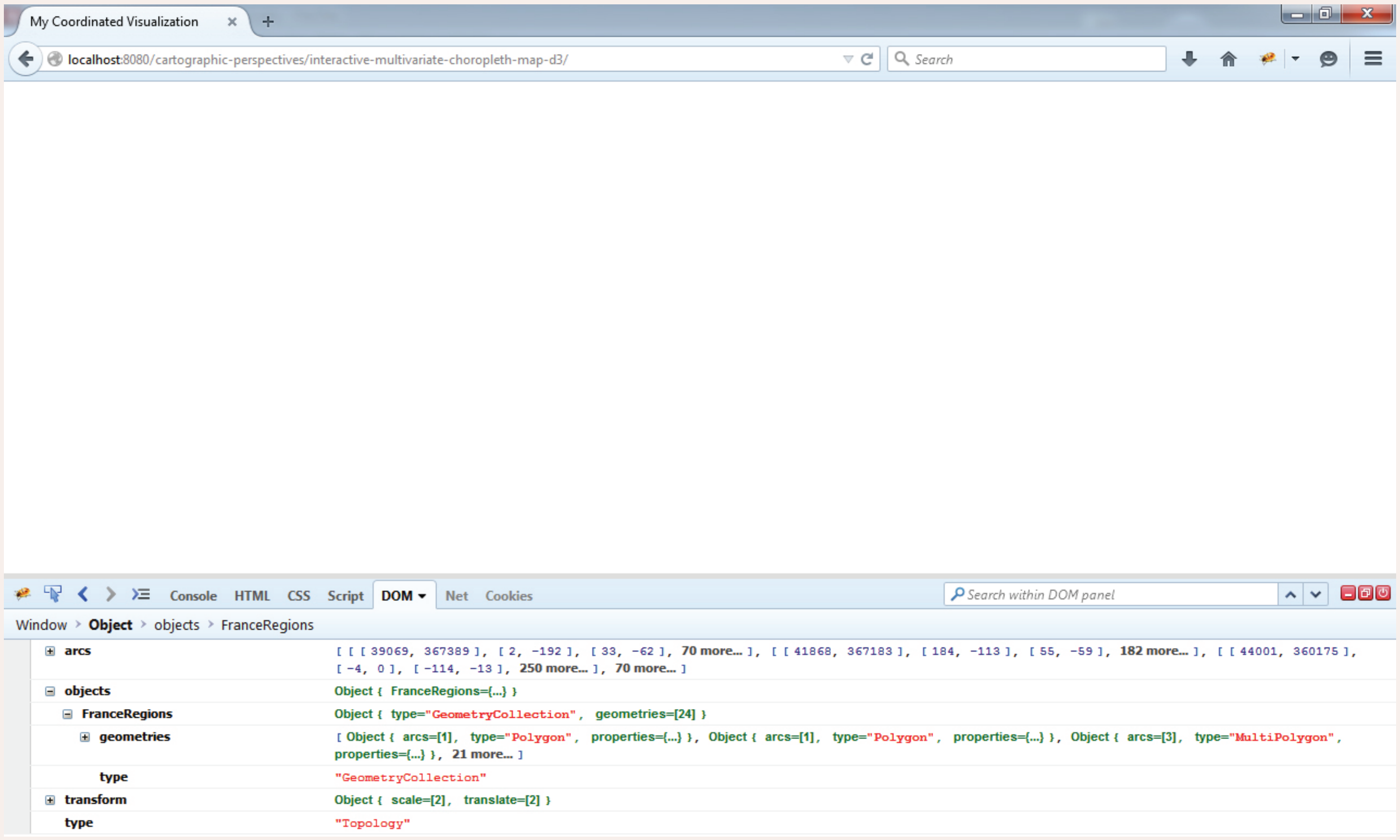
4. DRAWING YOUR BASEMAP

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. It will be easier to interpret the following instructions in this subsection if you first review the SVG specification at 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 first TopoJSON. All

```
//begin script when window loads
window.onload = initialize();

//the first function called once the html is loaded
function initialize(){
        setMap();
};

//set choropleth map parameters
function setMap(){
        //use queue.js to parallelize asynchronous data loading
        queue()
                .defer(d3.csv, “data/unitsData.csv”) //load attributes from csv
                .defer(d3.json, “data/EuropeCountries.topojson”) //load
                .defer(d3.json, “data/FranceRegions.topojson”) //load geometry
                .await(callback); //trigger callback function once data is loaded

        function callback(error, csvData, europeData, franceData){
                console.log();
        };
}
```

Example 2. *Loading data files and printing FranceRegions.topojson data to the console (in:* main.js*).*


My Coordinated Visualization
localhost:8080/cartographic-perspectives/interactive-multivariate-choropleth-map-d3/
Search
Console
HTML
CSS
Script
DOM
Net
Cookies
Search within DOM panel
Window > Object > objects > FranceRegions

```
arcs              [ [ [ 39069, 367389 ], [ 2, -192 ], [ 33, -62 ], 70 more... ], [ [ 41868, 367183 ], [ 184, -113 ], [ 55, -59 ], 182 more... ], [ [ 44001, 360175 ],
                  [ -4, 0 ], [ -114, -13 ], 250 more... ], 70 more... ]
objects           Object { FranceRegions={...} }
  FranceRegions   Object { type="GeometryCollection", geometries=[24] }
    geometries    [ Object { arcs=[1], type="Polygon", properties={...} }, Object { arcs=[1], type="Polygon", properties={...} }, Object { arcs=[3], type="MultiPolygon",
                  properties={...} }, 21 more... ]
    type          "GeometryCollection"
transform         Object { scale=[2], translate=[2] }
type              "Topology"
```

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

graphics associated with the choropleth view are drawn within the `setMap()` function, created in **Example 2** and extended in **Example 3**. First, set the size of the map view in pixels (**EX3: 3–5**). Note the absence of `px` after each number, as used in stylesheets; dimensions given without units to SVG elements will automatically be translated to pixels.

Next, create an `<svg>` element to contain the choropleth map using the `d3.select()` function (**EX3: 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

```
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/EuropeCountries.topojson") //load geometry
29            .defer(d3.json, "data/FranceRegions.topojson") //load geometry
30            .await(callback); //trigger callback function once data is loaded
31
32        function callback(error, csvData, europeData, franceData){
33            //add Europe countries geometry to map
34            var countries = map.append("path") //create SVG path element
35                .datum(topojson.feature(
                      europeData,europeData.objects.EuropeCountries))
36                .attr("class", "countries") //class name for styling
37                .attr("d", path); //project data as geometry in svg
38        };
39    }
```

Example 3. *Extending* `setMap()` *to draw the basemap.*

$("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 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(), 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: 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(), centered on France (**EX3: 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:

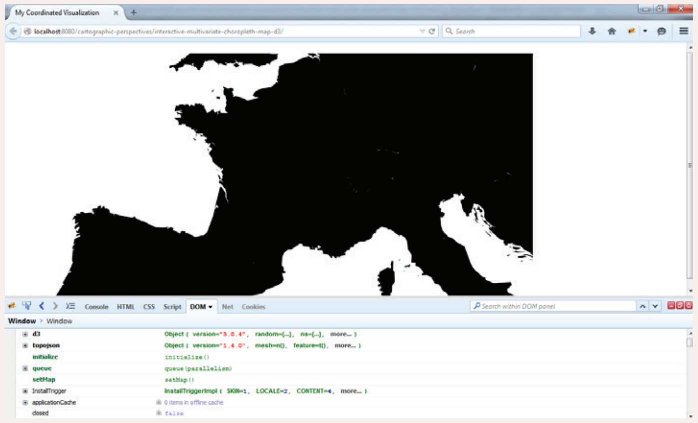
- .center recenters the map at a given [lon, lat] coordinate;
- .rotate rotates the globe counter-clockwise (from the North Pole) given angles of [lon, lat, roll] away from the geographic center;
- .parallels sets the standard parallels of the projection, given as [lat1, lat2];
- .scale is the scale of the map, set using an arbitrary scale factor; and
- .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 geometry according to these projection parameters. D3 uses a "geo path" SVG element to render the geometry included in a GeoJSON object as SVG. The d3.geo.path() function creates a new path generator with a default projection of Albers, centered on the 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() is a D3 ***generator function*** that creates a new function (the ***generator***) based on the parameters you send it. You then can store this generator as a variable, and access the variable like you would call a function, passing it parameters 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 the tutorial.

First, make use of the d3.geo.path() generator function to define a path generator that creates projected SVG paths from the geometry based on your map projection (**EX3: 21–23**). Then, make use of this path generator through the append() function to add an SVG <path> element containing the geometry derived from your TopoJSON, projected according to the generator definition (**EX3: 33–37**); 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 (**EX3: 34**). The second line specifies the datum() that will be attached to the countries selection (**EX3: 35**). 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). D3's .append() method returns the new element, so the countries selection will reference the appended <path> element and its associated datum.

At this point, it is acceptable to treat the polygons in the EuropeCountries 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(), data(), and enter() functions (described below). The datum() function expects a JSON or GeoJSON; to use the newer TopoJSON format, access the topojson.feature() method from *topojson.js*, indicating the object (EuropeCountries) 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* (**EX3: 36**). In the fourth line, the `d` attribute contains a string of information that describes the `<path>` (see: developer.mozilla.org/en-US/docs/SVG/Attribute/d) (**EX3: 37**). It is for this purpose that the `path` generator is so useful: it projects the `EuropeCountries` geometry and translates it into an SVG `<path>` description string.

Altogether, the revisions in **Example 3** result in four D3 ***blocks*** of chained methods connected by dot syntax to minimize the file size (**8–11**; **14–19**; **22–23**; **33–37**). 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


My Coordinated Visualization
localhost:8080/cartographic-perspectives/interactive-multivariate-choropleth-map-d3/
Search
Console
HTML
CSS
Script
DOM
Net
Cookies
Search within DOM panel
Window > Window

```
d3                Object { version="3.0.4", random={...}, ns={...}, more... }
topojson          Object { version="1.4.0", mesh=n(), feature=t(), more... }
initialize        initialize()
queue             queue(parallelism)
setMap            setMap()
InstallTrigger    InstallTriggerImpl { SKIN=1, LOCALE=2, CONTENT=4, more... }
applicationCache  0 items in offline cache
closed            false
```

Figure 4. *Drawing the basemap.*

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 (**EX3: 36**). Return to *style.css* and add the basic style rules provided in **Example 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.

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

To add the graticule to your basemap, begin by creating a generator called `graticule` (**EX5: 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.

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

Example 4. *Basic styles for the countries class (in style.css).*

The `graticule` generator should be placed **after** creating the `path` generator, but **before** loading and processing the TopoJSON files 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()` function and configure its attributes (**EX5: 5–9**). Then, add `gratLines` to the `map` using `selectAll()`, `data()`, and `enter()` and configure its attributes (**EX5: 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 **Example 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***, or a blank array into which will be placed one element for each graticule line (**EX5: 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) (**EX5: 13**). The `enter()` function adds

```
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
```

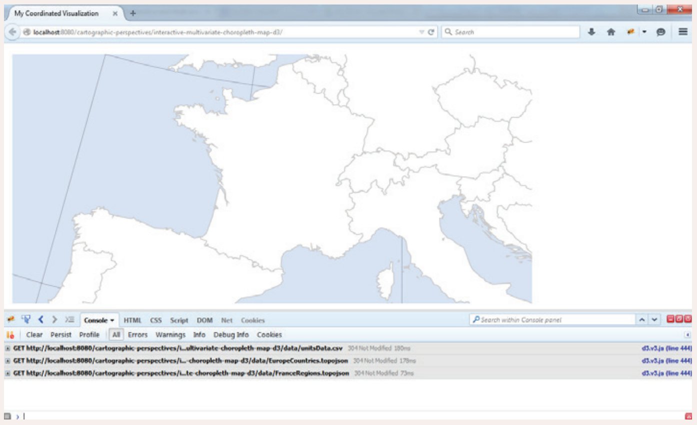
Example 5. *Adding a graticule to* `setMap()` *(in: main.js).*

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 (**EX5: 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()` function adds a new `<path>` element for each object in the selection, binding the datum to that element (**EX5: 15**). The first `attr()` call assigns each `<path>` element a class for styling purposes (**EX5: 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 each datum through the `path` generator into the `d` attribute, just as `gratBackground` and `countries` are projected.

Finally, style the `gratBackground` and `gratLines` in *style.css* using the class names "`gratBackground`" and "`gratLines`" (**Example 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**).

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

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

Example 6. *Styling the graticule (in style.css).*


My Coordinated Visualization
localhost:8080/cartographic-perspectives/interactive-multivariate-choropleth-map-d3/
Search
Console
HTML
CSS
Script
DOM
Net
Cookies
Search within Console panel
Clear
Persist
Profile
All
Errors
Warnings
Info
Debug Info
Cookies

```
GET http://localhost:8080/cartographic-perspectives/i...ultivariate-choropleth-map-d3/data/unitsData.csv 304 Not Modified 180ms    d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...-choropleth-map-d3/data/EuropeCountries.topojson 304 Not Modified 178ms    d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...te-choropleth-map-d3/data/FranceRegions.topojson 304 Not Modified 73ms    d3.v3.js (line 444)
```

Figure 5. *Styling the basemap.*

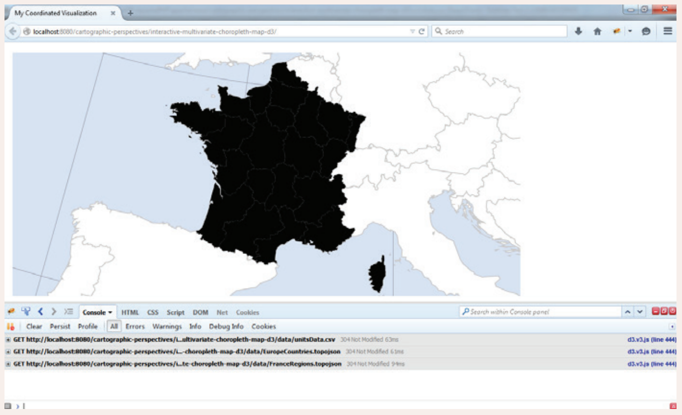
6. DRAWING YOUR CHOROPLETH MAP

WITH THE BASEMAP CONTEXT in place, you are ready to draw your choropleth map. Again, the choropleth map uses the geometry included in *FranceRegions*.

```
1      //retrieve and process data
2      function callback(error, csvData, europeData, franceData){
3
4             //add Europe countries geometry to map
5             var countries = map.append(“path”) //create SVG path element
6                    .datum(topojson.feature(europeData,
                                  europeData.objects.EuropeCountries))
7                           .attr(“class”, “countries”) //class name for styling
8                           .attr(“d”, path); //project data as geometry in svg
9
10            //add regions to map as enumeration units colored by data
11            var regions = map.selectAll(“.regions”)
12                   .data(topojson.feature(franceData,
                                        franceData.objects.FranceRegions).features)
13                   .enter() //create elements
14                   .append(“path”) //append elements to svg
15                   .attr(“class”, “regions”) //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     };
```

Example 7. *Drawing the choropleth map in* `setMap()` *(in:* main.js*).*

shp and converted to the `FranceRegions` object in your second TopoJSON. The `FranceRegions` object could be added through the `path` generator using `datum()`, as with `EuropeCountries` above. Unlike the basemap, however, each region is unique in both representation and interaction. You need to add each region separately in order to set different properties and attach different event listeners to each individual region. Therefore, you need to use the `selectAll()`, `data()`, and `enter()` methods—much like you did to draw each graticule line—rather than the simpler `append()` and `datum()` methods—like you did for drawing the basemap countries and graticule background (**Example 7**). A new selection named `regions` is created using the `selectAll()` method and each region is added to the map by the `path` generator (**EX7: 10–17**). It is important to note that **Example 7** must be added within the `callback` function, as the TopoJSON must first be processed before adding the `regions` element.


My Coordinated Visualization
localhost:8080/cartographic-perspectives/interactive-multivariate-choropleth-map-d3/
Search
Console
HTML
CSS
Script
DOM
Net
Cookies
Search within Console panel
Clear
Persist
Profile
All
Errors
Warnings
Info
Debug Info
Cookies

```
GET http://localhost:8080/cartographic-perspectives/i...ultivariate-choropleth-map-d3/data/unitsData.csv   304 Not Modified 63ms   d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...-choropleth-map-d3/data/EuropeCountries.topojson   304 Not Modified 61ms   d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...te-choropleth-map-d3/data/FranceRegions.topojson   304 Not Modified 94ms   d3.v3.js (line 444)
```

Figure 6. *Drawing the enumeration units.*

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**).

7. RELATING YOUR .CSV AND .TOPOJSON 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. **Example 8** makes use of a file named *unitsData.csv*. Again note that this file includes a column with the `adm1_code` header for each region (**Figure 1**), which can be used to

```
queue()
	.defer(d3.csv, “data/unitsData.csv”) //load attributes data from csv
	.defer(d3.json, “data/EuropeCountries.topojson”) //load geometry
	.defer(d3.json, “data/FranceRegions.topojson”) //load geometry
	.await(callback);

function callback(error, csvData, europeData, franceData){
	//variables for csv to json data transfer
	var keyArray = [“varA”,”varB”,”varC”,”varD”,”varE”];
	var jsonRegions = franceData.objects.FranceRegions.geometries;

	//loop through csv to assign each csv values to json region
	for (var i=0; i<csvData.length; i++) {
		var csvRegion = csvData[i]; //the current region
		var csvAdm1 = csvRegion.adm1_code; //adm1 code

		//loop through json regions to find right region
		for (var a=0; a<jsonRegions.length; a++){

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

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

				jsonRegions[a].properties.name = csvRegion.name; //set prop
				break; //stop looking through the json regions
			};
		};
	};

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

	//add regions to map as enumeration units colored by data
	var regions = map.selectAll(“.regions”)
		.data(topojson.feature(franceData,
			franceData.objects.FranceRegions).features)
		.enter() //create elements
		.append(“path”) //append elements to svg
		.attr(“class”, “regions”) //assign class for additional styling
		.attr(“id”, function(d) { return d.properties.adm1_code })
		.attr(“d”, path) //project data as geometry in svg
};
```

Example 8. *Relating your .csv and .topojson information within* `setMap()` *(in:* main.js*).*

join each region's multivariate information in the *.csv* file to its geographic information in the *.topojson*.

Example 8 includes part of **Example 2**, which shows the use of the `d3.csv` function with `queue()` to load and parse *unitsData.csv*. The `d3.csv` function parses each row into an object using the column headings as ***keys***, while `queue().await(callback)` passes the object to the callback function (**EX8: 4**). In order to attach the multivariate information from the *unitsData.csv* to the geographic information in *FranceRegions.topojson*, 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 **Example 8**, a set of nested loops is used to attach the multivariate information contained in `csvData` to the `FranceRegions` 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 (**EX8: 9–10**). An outer loop then loops through each of the region objects in the `csvData` array, assigning each object to the variable `csvRegion` and assigning the region's `adm1_code` to the variable `csvAdm1` (**EX8: 12–15**). An inner loop then cycles through each topojson `geometry` object, testing whether that object's `adm1_code` matches the `adm1_code` from the `csvData` region (**EX8: 17–21**). If the region codes match, a final loop runs through each `key` in the `keyArray` and assigns the corresponding key/value pair from the `csvData` region object to the `properties` object of the topojson `geometry` (**EX8: 23–28**). Also within the `if` statement, the topojson region is assigned the name of the `csvRegion` (**EX8: 30**). Once the right match has been found and attribute values transferred, the `jsonRegions` loop can be broken to save on processing time (**EX8: 31**). If this loop structure remains unclear, it is recommended that you add `console.log` statements line-by-line to inspect how the *.csv* and *.topojson* contents are being manipulated and combined through the nested `for` loops.

8. STYLING YOUR CHOROPLETH MAP

Now that the multivariate information in the *unitsData.csv* file is attached to the `FranceRegions` topojson object, you can color each region 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 **Example 8** to the top of main.js to make it a global variable (**EX9: 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.

Next, you need to add two new functions providing the logic for styling the choropleth map (**Example 10**): `colorScale()` and `choropleth()`. These functions are **external** to the `setMap()` function. The `colorScale()` function (**EX10: 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` object from the callback function as a parameter (**EX10: 1**),

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

Example 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    };
```

Example 10. *Functions for styling the choropleth map (in:* main.js*).*

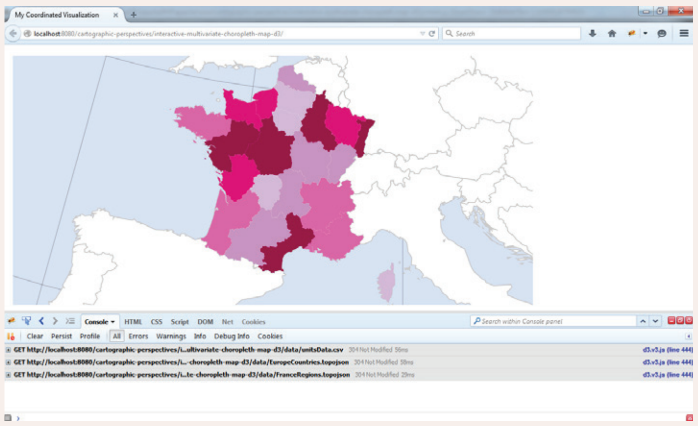
demonstrating the value of the AJAX solution. Because of this, the call to `colorScale()` must be added within the callback function, but before the `regions` block where the resulting scale will be used (**EX11: 2**). The `colorScale()` function first creates a `d3.scale.quantile()` generator named `color` and indicates the color scheme for the choropleth using the `range()` method, which specifies the scale's output (**EX10: 3–11**); a five-class, purple color scheme from ColorBrewer (colorbrewer.org) 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` is used to push the expressed attribute value for each enumeration unit into a single array, which is then passed to the `domain()` method (**EX10: 13–20**). The `color` generator is returned to `setMap()`and stored locally in `recolorMap` (**EX10: 19–EX11: 2**).

```
1      function callback(error, csvData, europeData, franceData){
2             var recolorMap = colorScale(csvData);
3
4             //EXAMPLE 8 SUPPRESSED FOR SPACE
5
6             var regions = map.selectAll(".regions")
7                    .data(topojson.feature(franceData,
                     franceData.objects.FranceRegions).features)
8                    .enter() //create elements
9                    .append("path") //append elements to svg
10                   .attr("class", "regions") //assign class for styling
11                   .attr("id", function(d) { return d.properties.adm1_code })
12                   .attr("d", path) //project data as geometry in svg
13                   .style("fill", function(d) { //color enumeration units
14                          return choropleth(d, recolorMap);
15                   });
16     };
```

Example 11. *Styling the choropleth map in* `setMap()` *(in:* main.js*).*

The `choropleth()` function then colors the enumeration units according to this quantile classification. The `choropleth()` function is called on the `style()` method in the `regions` block, within the callback function in `setMap()` (**EX11: 13–15**). The choropleth function takes two parameters: (1) a `datum` from the `FranceRegions` object associated with a given region (passed through the selection) and (2) the `color` generator, stored locally in the `recolorMap` variable. (**EX11: 14**). The choropleth function identifies the attribute value of the region under investigation (**EX10: 25**) and then checks if a value is valid (**EX10: 27**). If the value exists, the class color associated with that value's quantile is returned (**EX10: 28**); if it does not, a default grey is returned (**EX10: 30**).

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


My Coordinated Visualization
localhost:8080/cartographic-perspectives/interactive-multivariate-choropleth-map-d3/
Search
Console
HTML
CSS
Script
DOM
Net
Cookies
Search within Console panel
Clear
Persist
Profile
All
Errors
Warnings
Info
Debug Info
Cookies

```
GET http://localhost:8080/cartographic-perspectives/i...ultivariate-choropleth-map-d3/data/unitsData.csv   304 Not Modified 56ms   d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...-choropleth-map-d3/data/EuropeCountries.topojson   304 Not Modified 58ms   d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...te-choropleth-map-d3/data/FranceRegions.topojson   304 Not Modified 29ms   d3.v3.js (line 444)
```

Figure 7. *Styling the choropleth map.*

9. IMPLEMENTING DYNAMIC ATTRIBUTE SELECTION

With the map drawing properly, you are now ready to make it interactive. Interactivity changes a computer-generated map from static to dynamic, increasing both its utility and its aesthetic attraction for the user. D3 allows for an indefinite variety of map interactions, although implementing interaction behavior is not automatic and requires some creativity on the part of the developer. This tutorial will cover 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

```
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    };
```

Example 12. *Adding a dropdown menu in* `setMap()` *(in:* main.js*).*

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.`

The first block (**EX12: 2–7**) selects the HTML `<body>` element and appends a new `<div>` 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 `<div>`. The final line of the block appends a new `<select>` element (the menu itself). The second block (**EX12: 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 (**EX12: 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.

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 (**EX13: 8–10**). The `changeAttribute()` function contains the code that restyles the map according to the selected attribute (**EX13: 14–23**). First, the `expressed` variable is reassigned with the attribute option selected by the user (**EX13: 16**). Then, the `path` elements of all of the existing regions on the map are selected and restyled by the `choropleth()` function using a new `color` generator. Recall that the `colorScale()` function sets the scale range using the *.csv* data values of the expressed attribute, as shown in **Example 10**.

```
function createDropdown(csvData){
    //add a select element for the dropdown menu
    var dropdown = d3.select(“body”)
        .append(“div”)
        .attr(“class”,”dropdown”) //for positioning menu with css
        .html(“<h3>Select Variable:</h3>”)
        .append(“select”)
        .on(“change”, function(){
            changeAttribute(this.value, csvData);
        });

//REMAINDER OF EXAMPLE 12 AND EXAMPLE 10 SUPPRESSED FOR SPACE

function changeAttribute(attribute, csvData){
    //change the expressed attribute
    expressed = attribute;

    //recolor the map
    d3.selectAll(“.regions”) //select every region
        .style(“fill”, function(d) { //color enumeration units
            return choropleth(d, colorScale(csvData)); //->
        });
};
```

Example 13. *Adding an event listener and callback function (in:* main.js*).*

10. IMPLEMENTING HIGHLIGHTING AND 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` (**EX14: 11**), (2) `dehighlight()`, which reverts the enumeration unit back to its original color and deactivates the dynamic label on `mouseout` (**EX14: 12**), and (3) `moveLabel()`, which updates the position of the dynamic label according to changes in the x/y coordinates of the mouse on `mousemove` (**EX14: 13**). The event listeners should be added at the end of the `regions` block in `setMap()` (**EX11: 6–15**) 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 **Example 14** appends the original color as a text string in a `<desc>` SVG element (**EX14: 14–17**). 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 **Example 15**.

Once adding the event listeners to the `regions` 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()` (**EX16: 1–19**). The `highlight()` function receives the `data` object associated with highlighted enumeration unit as the parameter. The `data` object’s properties are stored in a local variable named `props` (**EX16: 3**). The `highlight()`

```
var regions = map.selectAll(“.regions”)
        .data(topojson.feature(franceData,
                franceData.objects.FranceRegions).features)
        .enter() //create elements
        .append(“path”) //append elements to svg
        .attr(“class”, “regions”) //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);
        })
        .on(“mouseover”, highlight)
        .on(“mouseout”, dehighlight)
        .on(“mousemove”, moveLabel)
        .append(“desc”) //append the current color
                .text(function(d) {
                        return choropleth(d, recolorMap);
        });
```

Example 14: Adding event listeners to `regions` *in* `setMap()` *(in: main.js).*

```
function changeAttribute(attribute, csvData){
        //change the expressed attribute
        expressed = attribute;

        //recolor the map
        d3.selectAll(“.regions”) //select every region
                .style(“fill”, function(d) { //color enumeration units
                        return choropleth(d, colorScale(csvData)); //->
                })
                .select(“desc”) //replace the color text in each desc element
                        .text(function(d) {
                                return choropleth(d, colorScale(csvData)); //->
                        });
};
```

Example 15. *Resetting the* `<desc>` *element in* `changeAttribute()` *(in: main.js).*

function then calls `d3.select()` to find the SVG path for the region, using the `adm1_code`, and changes its `fill` style to black (**EX16: 5–6**). There are many other possible highlighting solutions; this one was chosen for simplicity.

The `highlight()` function then designates two HTML strings used in the dynamic label, one with the attribute data (`labelAttribute`) and one with the region name (`labelName`) (**EX16: 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 (**EX16: 11–19**). A child `<div>` named `labelname` is added to `infolabel` to position the region name within the label.

```
1      function highlight(data){
2
3             var props = data.properties; //json properties
4
5             d3.select(“#”+props.adm1_code) //select the current region 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 dehighlight(data){
22
23            //json or csv properties
24            var props = data.properties; //json properties
25            var region = d3.select(“#”+props.adm1_code); //select the current region
26            var fillcolor = region.select(“desc”).text(); //access original color from desc
27            region.style(“fill”, fillcolor); //reset enumeration unit to orginal color
28
29            d3.select(“#”+props.adm1_code+”label”).remove(); //remove info label
30     };
```

Example 16. *Highlighting and de-highlighting the choropleth map (in: main.js).*

Add style rules to the dynamic label in *style.css*, using the `infolabel` and `labelname` class identifiers. **Example 17** provides basic style rules to create a 200×50px dynamic label with white text and a black background. Note that the `<h1>` and `<b>` tags are styled for the `infolabel` text (**EX17: 17–26**), as these are used in the `labelAttribute` HTML string (**EX16: 8**); you are not limited to these tags in the design of your dynamic label.

Reload your *index.html* page in your browser 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 in a label (**Figure 8a**). The `dehighlight()` function must be implemented to revert the enumeration unit to its original color and deactivate the dynamic label (**EX16: 21–30**). First, the `properties` of the `data` object for the selected enumeration unit are stored in the `props` variable (**EX16: 24**). Then the enumeration unit itself is selected and the text within its `<desc>` element is retrieved and assigned to the variable `fillcolor` (**EX16: 21–30**). The selected region is assigned this color as its fill (**EX16: 27**). Finally, the dynamic label is selected using its id and removed (**EX16: 29**).

Although the enumeration units should now correctly highlight and dehighlight, the dynamic label is positioned away from the map itself rather than near the highlighted region. To make the dynamic label follow the user's mouse cursor, first change the positioning of the `svg` map container to `absolute` in *style.css* (**EX17: 1–3**); this should be defined early in the stylesheet so it may be overridden

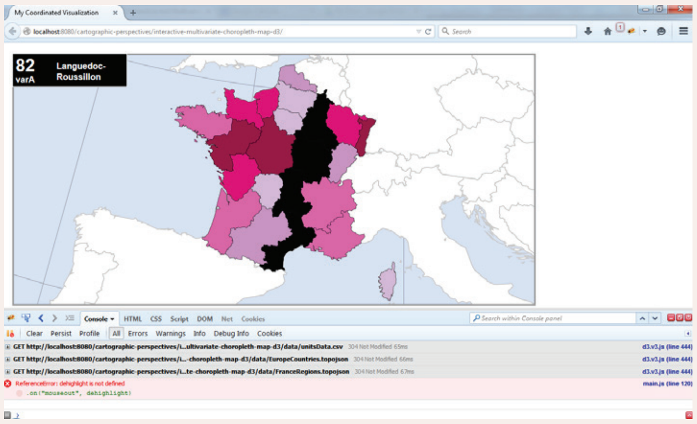
```css
svg {
      position: absolute;
}

//EXAMPLES 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;
}
```

Example 17. *Styling the dynamic label (in: style.css).*

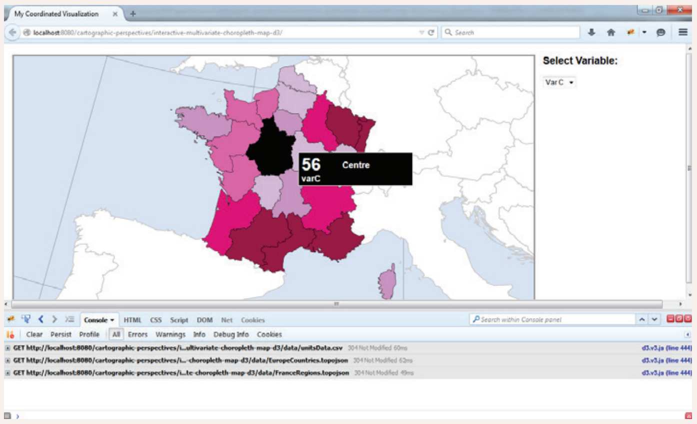
by individual class rules. Then, define the `moveLabel()` function in *main.js* that is called on `mousemove` atop an enumeration unit (**Example 18**). The `moveLabel()` function uses the `d3.event()` method to access the current mouse event (`mousemove`), which includes mouse coordinate properties (`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 (**EX18: 3–4**). Reload the *index.html* page; you now should be able to mouse over each enumeration unit, resulting in highlighting of the enumeration unit and activation of a dynamic label that follows the mouse (**Figure 8b**).

```
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
};
```

Example 18. *Styling the dynamic label (in:* main.js*).*



```
GET http://localhost:8080/cartographic-perspectives/i...ultivariate-choropleth-map-d3/data/unitsData.csv 304 Not Modified 65ms d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...-choropleth-map-d3/data/EuropeCountries.topojson 304 Not Modified 66ms d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...te-choropleth-map-d3/data/FranceRegions.topojson 304 Not Modified 67ms d3.v3.js (line 444)
ReferenceError: dehighlight is not defined main.js (line 120)
    .on("mouseout", dehighlight)
```



```
GET http://localhost:8080/cartographic-perspectives/i...ultivariate-choropleth-map-d3/data/unitsData.csv 304 Not Modified 60ms d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...-choropleth-map-d3/data/EuropeCountries.topojson 304 Not Modified 62ms d3.v3.js (line 444)
GET http://localhost:8080/cartographic-perspectives/i...te-choropleth-map-d3/data/FranceRegions.topojson 304 Not Modified 49ms d3.v3.js (line 444)
```

Figures 8a (left) and 8b (below)*. Implementing highlighting and tooltips in the choropleth map.*

DOING MORE WITH D3

At this point, you should be comfortable with the essential workings of D3's mapping functionality. This tutorial has covered finding and formatting data, converting geographic data into GeoJSON and TopoJSON formats, asynchronously loading data into the browser, using D3 Projections and a D3 Geo Path generator to draw a graticule and basemap, styling the basemap based on one of multiple attributes, creating a dropdown menu to allow the user to select between attributes, and implementing some interactions for retrieving information from the choropleth map.

Much more can be done with the example data provided than has been covered here. D3 provides many different types of visualizations that allow for dynamic data journalism (for examples: github.com/mbostock/d3/wiki/Gallery). These may be used independently or have their interactions linked through the magic of JavaScript. The authors hope that this toehold into the world of D3 lowers the bar for readers seeking to experiment with new web-mapping technologies, and that you will contribute to the growing canon of interactive multivariate web maps that follow sound cartographic principles.

REFERENCE

Donohue R. G., C. M. Sack, and R. E. Roth. 2013. "Time series proportional symbol maps with Leaflet and jQuery." *Cartographic Perspectives* 76: 43–66. doi: 10.14714/CP76.1248.