Geography 575

Lab #1: Spatiotemporal Visualization Challenge

Lab Objectives:

- Introduce you to JavaScript and the Leaflet.js mapping library
- Introduce you to NotePad++ and Firebug development tool
- Dynamically load, map, and animate a spatiotemporal information set

Evaluation:

This lab is worth **20 points** toward the Lab Assignments evaluation item, which is worth 25% of your overall course grade. A grading rubric is provided at the end of the lab to inform your work.

Schedule of Deliverables:

• **January 31st:** Lab #1 Assigned	//client contract begins
• **February 7th:** Information Check-In	//initial research completed
• **February 14th:** No Lab; schedule w/ DoIT	//input & feedback from client
• **February 21st:** Lab #1 Due	//contract deadline

Challenge Description

You have been invited by Professor Kris Olds to contribute dynamic content to the *World Regions in Global Context* course (Geography 340), an introductory course on regional development. The G340 course leverages web-based technologies to bring students and experts together from around the globe to discuss the geographic phenomena and processes (i.e., space over time) constituting the cultural and natural worlds. Professor Olds has requested that you contribute to the course material by developing an example spatiotemporal visualization that "makes visible" some regional geographic phenomenon or process for structuring in-class discussion. Professor Olds has recommended that you map spatiotemporal information collected about cities—given his expertise in Urban Geography—making proportional symbols the appropriate thematic map approach; the specific geographic phenomena/processes portrayed by the spatiotemporal visualization remains your choice. The final webpage should prompt hypotheses about the underlying drivers of the spatiotemporal pattern/process to promote discussion in the course.

Editor's Notes from the World Regions *Class*

Your spatiotemporal visualization must include *at least 15* point locations, with each point location exhibiting variation across *at least 7* timestamps (e.g., days, months, years, decades). The point locations can be either within a single country or across a larger region. While you are required to design a proportional symbol map (i.e., not a choropleth map, dot density map, etc.), you may choose to map a spatiotemporal information set that is aggregated to units other than cities with permission from the course instructor.

1. Spatiotemporal Visualization on the Web

a. Overview of Leaflet

In this lab, you will use a simple parsing script to dynamically load information from a spreadsheet file and use Leaflet to draw and animate proportional symbols representing that information atop a slippy web map. ***Leaflet*** is one of many available JavaScript libraries or APIs now available for publishing tile-based web maps. Leaflet is quickly growing in popularity within the web development community because it is both lightweight (it is only 28kb of code) and open source (meaning you both can view how it functions and extend it to fit your needs). Maps produced using Leaflet can draw from a variety of basemap tile services and are viewable on mobile devices; because of its use of ***scalable vector graphics*** (*.svg*) for drawing vector overlays (described below), maps produced using Leaflet are not compatible with older versions of Internet Explorer (before IE9) without a considerable work-around. Additional information about Leaflet is available at: http://leafletjs.com/

In the following lab tutorial, all code samples are shown in Notepad++, a simple, free, and powerful open-source text editor. Notepad++ is available on all Science Hall lab computers and can be downloaded for use on your own machine at: http://notepad-plus-plus.org/. You may use other text editors or web-scripting software, such as Aptana (another popular open-source package), TextWrangler (for Mac), and Dreamweaver (a proprietary product from Adobe). Also, the following lab tutorial uses the Mozilla Firefox browser (http://www.mozilla.org/en-US/firefox/fx/#desktop) and the Firebug development tool for Firefox (https://getfirebug.com/) for previewing and debugging your spatiotemporal visualization. Developer tools in Chrome also work well.

The following tutorial assumes that you have a basic understanding of HTML5, CSS, and JavaScript; please use the following resources to review these languages, if needed:

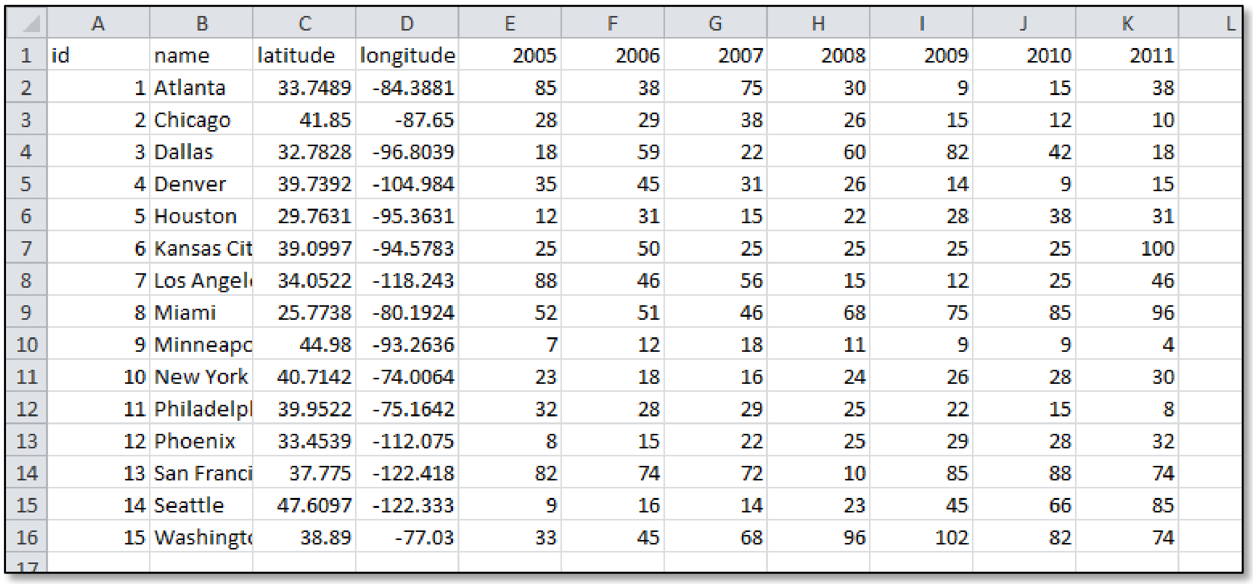
- Lynda Tutorials (free when logging in as a UW student)
- Codecademy (free when logging in as a UW student)
- DoIT STS Training
- Mozilla Developer Network
- w3schools

b. Finding and Formatting Spatiotemporal Information

The first step towards completing the challenge is the assembly of appropriate ***spatiotemporal*** (space+time) information regarding an urban pattern or process. In G370, several sources of geographic information were introduced, including the ArcGIS data bank, Natural Earth, GeoCommons, and Wikipedia; the latter two are particularly helpful for finding statistical information that has a temporal component (i.e., the same geographic information, but captured multiple times). Because proportional symbol maps leverage the visual variable size, you only should map information that is at the ordinal or, preferably, numerical level of measurement (i.e., do not collect categorical information). Be sure that the information you acquire tells a compelling story and/or reveals new insight into the geographic pattern/process.

In order to leverage the following code banks, you need to format your spatiotemporal information set with the unique ***map features*** (e.g., cities, regions) included as rows and the unique

timestamps (generically describing either a single moment in time or a time interval) included as columns (**Figure 1**). Include an additional pair of columns at that start of your file for a unique ID and name field; because Leaflet natively understands the geographic coordinate system, you also must include a pair of columns for the latitude and longitude of the proportional symbol anchor (e.g., the city center, the centroid of the region). Be sure to use logical header names (e.g., "name", "latitude", "2005" etc.), as these terms are used as attribute keys for referencing the spatiotemporal information using JavaScript. While you may use any spreadsheet software (e.g., Excel, Google docs, Notepad++) to assemble your information set, you need to save your table to the ***.csv*** (comma separated values); the following code banks use the filename "csvData.csv".



	A	B	C	D	E	F	G	H	I	J	K	L
1	id	name	latitude	longitude	2005	2006	2007	2008	2009	2010	2011	
2	1	Atlanta	33.7489	-84.3881	85	38	75	30	9	15	38	
3	2	Chicago	41.85	-87.65	28	29	38	26	15	12	10	
4	3	Dallas	32.7828	-96.8039	18	59	22	60	82	42	18	
5	4	Denver	39.7392	-104.984	35	45	31	26	14	9	15	
6	5	Houston	29.7631	-95.3631	12	31	15	22	28	38	31	
7	6	Kansas Cit	39.0997	-94.5783	25	50	25	25	25	25	100	
8	7	Los Angel	34.0522	-118.243	88	46	56	15	12	25	46	
9	8	Miami	25.7738	-80.1924	52	51	46	68	75	85	96	
10	9	Minneapo	44.98	-93.2636	7	12	18	11	9	9	4	
11	10	New York	40.7142	-74.0064	23	18	16	24	26	28	30	
12	11	Philadelpl	39.9522	-75.1642	32	28	29	25	22	15	8	
13	12	Phoenix	33.4539	-112.075	8	15	22	25	29	28	32	
14	13	San Franci	37.775	-122.418	82	74	72	10	85	88	74	
15	14	Seattle	47.6097	-122.333	9	16	14	23	45	66	85	
16	15	Washingt	38.89	-77.03	33	45	68	96	102	82	74	
17												

Figure 1: An Example Spatiotemporal Information Set. In the table, each map feature should be included as a row while the lat/long coordinates and timestamps should be included as columns. Note: The above information is meaningless and does not describe any real spatiotemporal phenomenon or process.

2. Getting Started with Leaflet

a. Preparing Your Basic Webpage Structure

The first step in creating your spatiotemporal visualization is to prepare your directory. First create a project directory (folder) for your website, giving it a logical name (e.g., "g575-lab1"), and then create four folders within the project directory named "css", "data", "img", and "js" to store the different types of files related to your website. Place the *.csv* file you assembled in Section 1b in the *data* folder.

Open Notepad++ and create a new file (*File->New*). Save the file in the parent directory (i.e., not with in a subfolder) using the name "index.html" (*File->Save as…*); the "index.html" file serves as the landing page for your spatiotemporal visualization. Once you have created the file, add the boilerplate text provided in **Code Bank 1** needed for all webpages; change the content of the `<title>` element to something logical for your spatiotemporal visualization.

```
<!DOCTYPE HTML>
<html>
      <head>
            <meta charset="utf-8">
            <title>My Spatiotemporal Visualization</title>
      </head>
      <body>
      </body>
</html>
```

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

The first html element to add to your boilerplate is a `<div>` element that contains the map served by Leaflet. The `<div>` element should be added to the `<body>` of *index.html* and have an `id` attribute of "`map`" for referencing by stylesheets and scripts (**Code Bank 2: Lines 2-3**)

```
1     <body>
2           <!--div for the map-->
3           <div id="map"></div>
4     </body>
```

Code Bank 2: Preparing the <body> Element (in: *index.html*).

Next, create a second file in Notepad++, this time saving it as a stylesheet (*.css*) with the name "style.css"; your stylesheet should be saved in the *css* folder. Once created, add the `<link>` element to *.css* file in the `<head>` element of *index.html*, pointing to the directory location of *style.css* (**Code Bank 3: Line 5-6**).

```
1     <head>
2           <meta charset="utf-8">
3           <title>Cities Map</title>
4
5           <!--main stylesheet-->
6           <link rel="stylesheet" href="css/style.css" />
7     </head>
```

Code Bank 3: Referencing the *style.css* Stylesheet (in: *index.html*).

The *style.css* stylesheet contains all rules for positioning and styling your webpage, including both the central spatiotemporal visualization and the surrounding HTML5 elements. **Code Bank 4** provides the basic rules needed to view the `<div>` element containing your map; you are encouraged to adjust the position and size of your map based on your design vision for the scenario. Although the emphasis in Lab #1 is on the map itself, and the JavaScript needed to implement it, <u>be sure to add additional elements to *index.html* that are needed for a professional-looking webpage and to style them using the *style.css* stylesheet</u> (this is worth a portion of your grade).

```css
body {
	width: 100%;
	height: 100%;
	margin: 5px;
	font-family: "Helvetica Neue", Arial, Helvetica, sans-serif;
}

#map {
	height: 400px;
	width: 800px;
}
```

Code Bank 4: Styling the `<div>` Element Containing the Map (in: *style.css*).

Finally, create a third file in Notepad++, this time saving it as a JavaScript (*.js*) file with the name "main.js"; save it to the *js* folder. The *main.js* file contains the logic (code) needed to execute your spatiotemporal visualization. While you could include this code within a `<script>` element in the *index.html* file, it is better practice to keep lengthier blocks of code organized in separate files; such practice also promotes reusability of code. Once created, reference *main.js* in the `<body>` of the *index.html* file using the `<script>` element, following the existing `<div>` element for the map (**Code Bank 5: Line 5-6**).

```html
1	<body>
2		<!--div for the map-->
3		<div id="map"></div>
4
5		<!--link to main javascript file-->
6		<script src="js/main.js"></script>
7	</body>
```

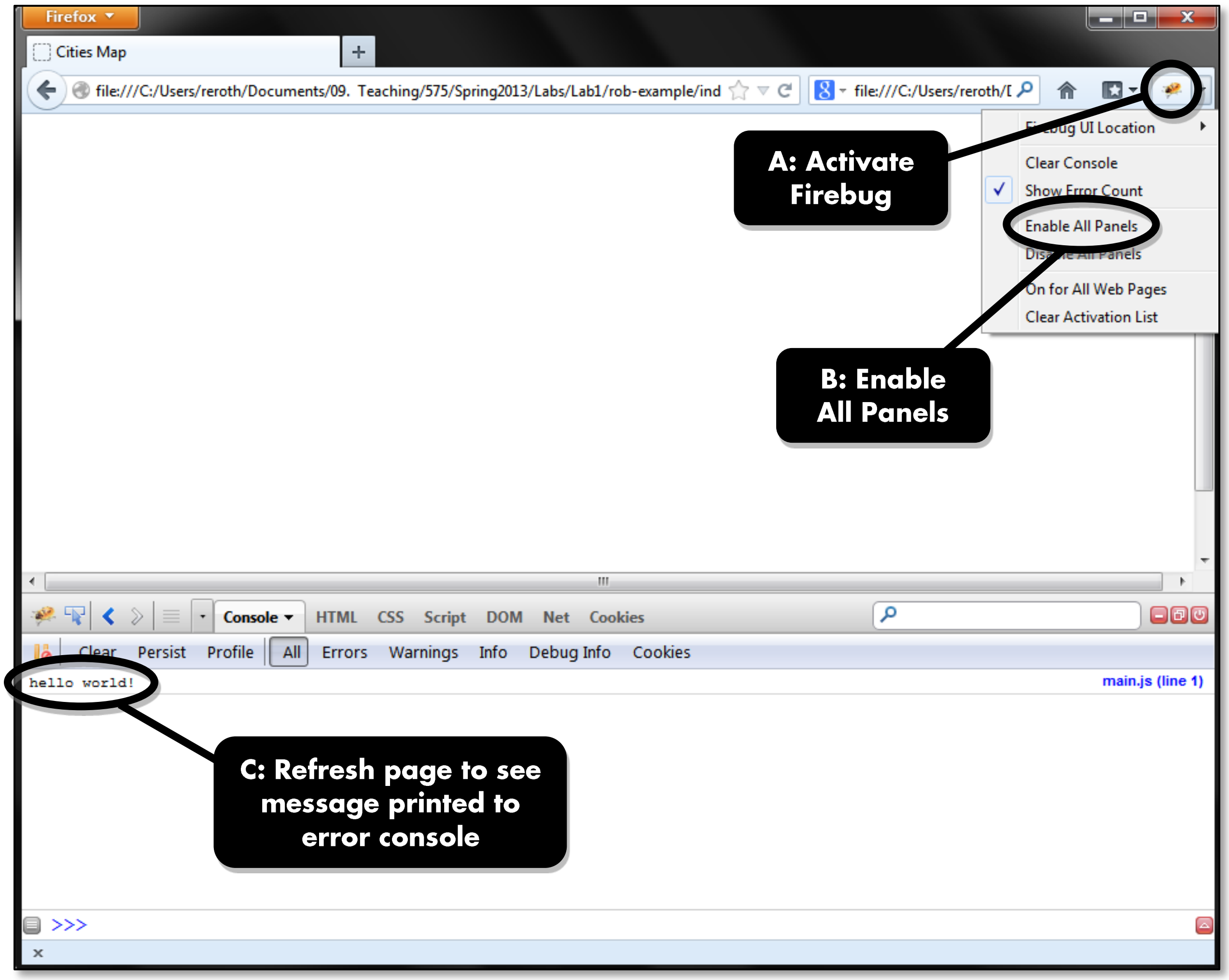
Code Bank 5: Referencing the *main.js* JavaScript file (in: *index.html*).

Before moving onto the next step, check to see if your file structure and webpage files are properly configured. The primary method for debugging scripts is by printing a message to the ***error console*** using the `console.log()` method in JavaScript. To demonstrate its utility, and confirm that your webpage is properly configured, add a script to print to the console in the *main.js* file (**Code Bank 6**).

```javascript
console.log("hello world!");
```

Code Bank 6: Debugging Scripts with the Console (in: *main.js*).

Once added, open *index.html* in Firefox; at this point, it should be a blank webpage (**Figure 2**). Activate Firebug by clicking the Firebug icon; you may need to *Enable All Panels* in the Firebug dropdown option, if not already enabled. Once activated, click the console tab and reload the page.


Firefox
Cities Map
file:///C:/Users/reroth/Documents/09. Teaching/575/Spring2013/Labs/Lab1/rob-example/ind
A: Activate Firebug
Firebug UI Location
Clear Console
Show Error Count
Enable All Panels
Disable All Panels
On for All Web Pages
Clear Activation List
B: Enable All Panels
Console HTML CSS Script DOM Net Cookies
Clear Persist Profile All Errors Warnings Info Debug Info Cookies
C: Refresh page to see message printed to error console
>>>

```
hello world!                                                    main.js (line 1)
```

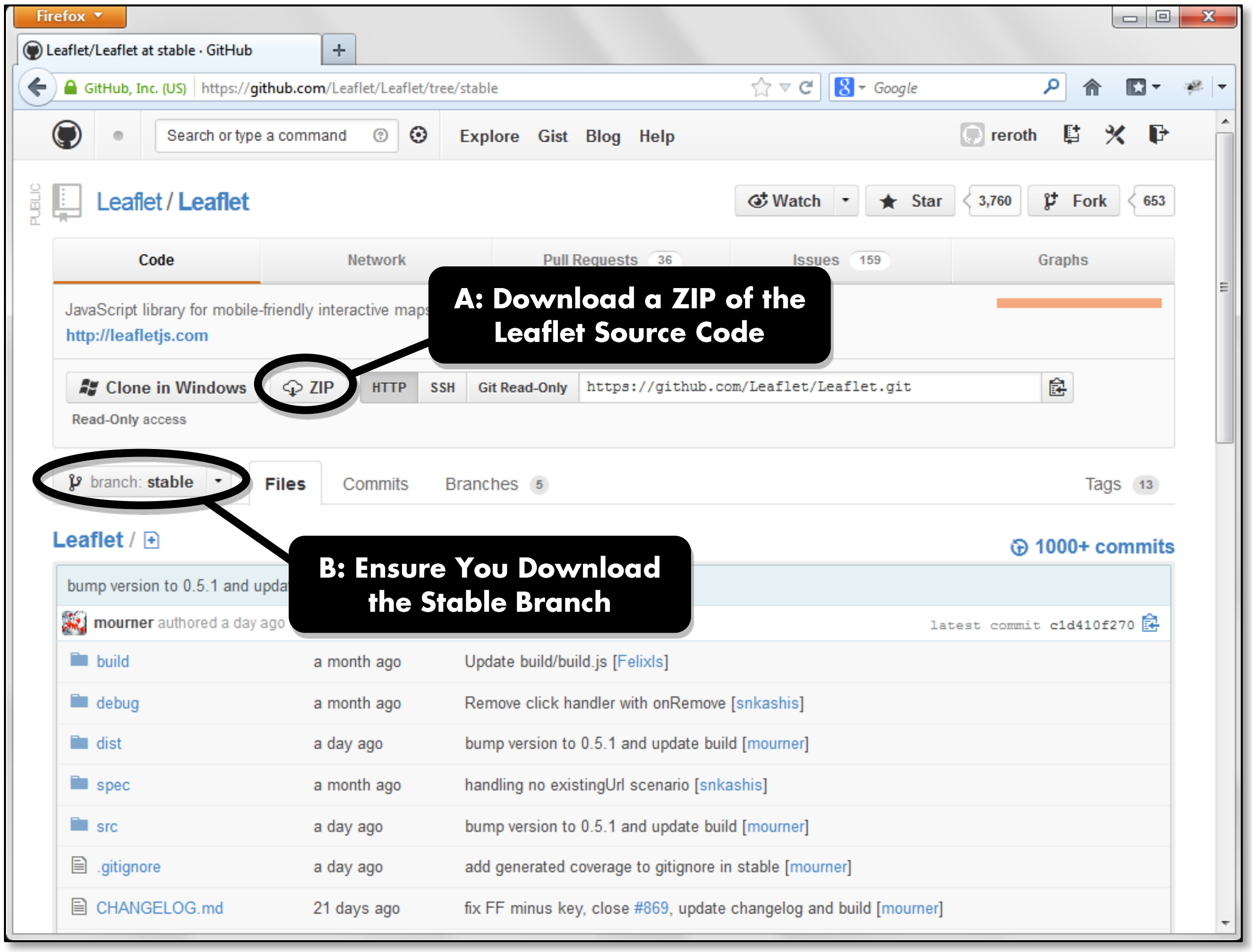
Figure 2: Debugging using the Error Console in Firebug.

You should see your "hello world" as a line of text, with its line number in the file it originates from shown on the far right.

b. Preparing Your Project Directory to Use Leaflet

Now that you have a basic webpage configured—and understand how to debug this webpage using Firebug—you can add the Leaflet source code. As introduced above, information about Leaflet is available at: http://leafletjs.com/. Throughout G575, it is important that you get comfortable reviewing the Leaflet API Reference, available at: http://leafletjs.com/reference.html. The Leaflet API Reference provides an overview of all classes included in the Leaflet source code, with a description and code example of each class's associated properties, methods, and events; the G575 labs introduce only a portion of the total Leaflet functionality available for your final project.

You need to download the Leaflet JavaScript source code before you can leverage it in your spatiotemporal visualization. To download the library, navigate to the Leaflet Github page (https://github.com/Leaflet/Leaflet) and click on the "ZIP" icon (Figure 3). Make sure that the branch you are acquiring is the "stable" version (*branch->stable*).


Firefox
Leaflet/Leaflet at stable · GitHub
GitHub, Inc. (US) https://github.com/Leaflet/Leaflet/tree/stable
Search or type a command
Explore Gist Blog Help
reroth
Leaflet / Leaflet
Watch
Star 3,760
Fork 653
Code Network Pull Requests 36 Issues 159 Graphs
A: Download a ZIP of the Leaflet Source Code
Clone in Windows ZIP HTTP SSH Git Read-Only https://github.com/Leaflet/Leaflet.git
Read-Only access
branch: stable
Files Commits Branches 5 Tags 13
B: Ensure You Download the Stable Branch
1000+ commits

JavaScript library for mobile-friendly interactive maps
http://leafletjs.com

Leaflet /

bump version to 0.5.1 and upda
mourner authored a day ago — latest commit c1d410f270

build	a month ago	Update build/build.js [Felixls]
debug	a month ago	Remove click handler with onRemove [snkashis]
dist	a day ago	bump version to 0.5.1 and update build [mourner]
spec	a month ago	handling no existingUrl scenario [snkashis]
src	a day ago	bump version to 0.5.1 and update build [mourner]
.gitignore	a day ago	add generated coverage to gitignore in stable [mourner]
CHANGELOG.md	21 days ago	fix FF minus key, close #869, update changelog and build [mourner]

Figure 3: Acquiring the Leaflet Library.

Save the *.zip* file to your desktop and extract it to a folder. You only need to copy a portion of the downloaded library into your project directory, as Leaflet provides two version of its source code in the *dist* folder: (1) a human-readable version (*dist/leaflet-src.js*) and (2) a condensed version (*dist/leaflet.js*) that runs slightly faster, but only in the range of milliseconds. It is recommended that you make use of the human-readable version in the G575 labs to learn how the *leaflet.js* functionality is working, and thus how to modify and extend this functionality for your final project. Copy the following files/folders to your project directory:

- Copy the *dist/leaflet-src.js* file and the *dist/images* folder into your *js* project folder; do not copy the *dist/images* folder into your *img* project folder, as the latter is for additional images added to your webpage.

- Copy the *dist/leaflet.css* and *dist/leaflet.ie.css* files into your *css* project folder.

One copied to your project directory, you need to reference the Leaflet source files in *index.html*. The reference to the stylesheets should be included in the `<head>` element (**Code Bank 7**); the pair of stylesheets is needed to layout and style the map differently if viewed in Internet Explorer (**Code Bank 7: Lines 10-12**). The reference to *leaflet-src.js* should be included in the `<body>` element, before linking to *main.js* (as the code in *main.js* makes use of code in *leaflet-src.js*; **Code Bank 8**).

```
<head>
      <meta charset="utf-8">
      <title>Cities Map</title>

      <!--main stylesheet-->
      <link rel="stylesheet" href="css/style.css" />

      <!--leaflet stylesheet-->
      <link rel="stylesheet" href="css/leaflet.css" />
      <!--[if lte IE 8]>
            <link rel="stylesheet" href="css/leaflet.ie.css" />
      <![endif]-->
</head>
```

Code Bank 7: Linking to the Leaflet Stylesheets (in: *index.html*).

```
<body>
      <!--div for the map-->
      <div id="map"></div>

      <!--libraries-->
      <script src="js/leaflet-src.js"></script>

      <!--link to main javascript file-->
      <script src="js/main.js"></script>
</body>
```

Code Bank 8: Linking to the Leaflet Script (in: *index.html*).

c. Loading a Basemap Using Leaflet

You are now ready to load basemap tiles into your webpage using Leaflet! Leaflet allows you to load tiles from a variety of sources, including those that use OpenStreetMap information such as the CloudMade Map Styles (http://maps.cloudmade.com/editor) and those available through commercial services such as ArcGIS Online, Bing!, and Google. An overview of public tile services is available on the GIS Collective blog: http://giscollective.org/tutorials/web-mapping/wmsthree/. For Leaflet to use a public tile service, you need to reference the URL using the following syntax:

- ***http://{s}.acetate.geoiq.com/tiles/acetate/{z}/{x}/{y}.png***

Every tile in a slippy map is a separate 256 x 256 pixel image—a *.png* file in the above example syntax. The {s} indicates possible server instances from which the map can draw tiles. For each loaded tile, {z} indicates its zoom level, {x} indicates its horizontal coordinate, and {y} indicates its vertical coordinate. Near all public tile services use this z/x/y directory format, which was pioneered by Google. The example syntax above loads the minimalist ***Acetate*** tile service (http://developer.geoiq.com/tools/acetate/) from GeoIQ (now Esri), the same tile service used for GeoCommons.com; a minimalist tile design is recommended when adding thematic content atop the basemap tiles.

Once you have selected a tile service, you need to make four additions to your *main.js* file in order to load the selected tile service: (1) Create a global variable to store your map object for subsequent reference; use the instance name `map` by convention (**Code Bank 9: Lines 1-2**); (2) Add the `window.onload` event handler to call the custom `initialize()` function when the page loads (**Code Bank 9: Lines 4-5**); (3) complete the `initialize()` function definition by having it re-route the execution to the custom `setMap()` function (**Code Bank 9: Lines 7-11**); this function can be used to initialize multiple, linked views (e.g., both a map and linked information graphic, as with Lab #3); and (4) complete the `setMap()` function definition (**Code Bank 9: Lines 13-26**). It is the `setMap()` function that leverages the *leaflet-src.js* code, setting the initial centering and zoom level of the map (**Code Bank 9: Lines 17-18**) as well as setting the Acetate tile service for the basemap, providing proper attribution (**Code Bank 9: Lines 20-25**).

Comments have been added to **Code Bank 9** to explain the purpose of each line of code, including comment arrows (`//<-` and `//->`) that make the flow of execution explicit in *main.js*. The code provided in **Code Bank 9** is more complicated than the Leaflet Quick Start Guide (http://leafletjs.com/examples/quick-start.html), but provides necessary structure for future steps in Lab #1. After adding the code in **Code Bank 9** to the *main.js* file; return to Firefox and refresh the *index.html* page; the map `<div>` now should be populated with the Acetate tileset, including basic slippy map interactivity (panning + zooming).

```
1       //global variables
2       var map; //map object
3
4       //begin script when window loads
5       window.onload = initialize(); //->
6
7       //the first function called once the html is loaded
8       function initialize(){
9               //<-window.onload
10              setMap(); //->
11      };
12
13      //set basemap parameters
14      function setMap() {
15              //<-initialize()
16
17              //create the map and set its initial view
18              map = L.map('map').setView([38, -94], 4);
19
20              //add the tile layer to the map
21              var layer = L.tileLayer(
22                      'http://{s}.acetate.geoiq.com/tiles/acetate/{z}/{x}/{y}.png',
23                      {
24                              attribution: 'Acetate tileset from GeoIQ'
25                      }).addTo(map);
26      };
```

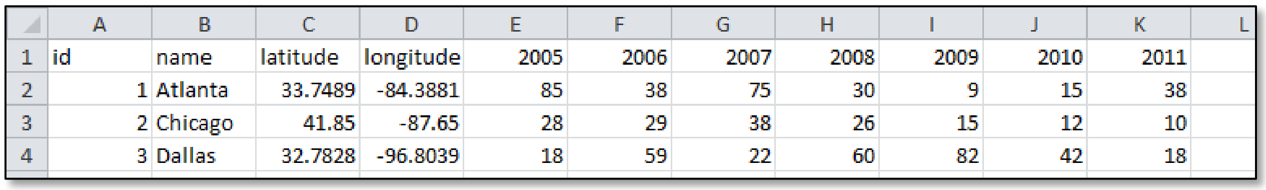
Code Bank 9: Loading a Basemap using Leaflet (in: *main.js*).

3. Loading & Mapping Your Spatiotemporal Information

a. Dynamically Loading Your CSV File

Once you have successfully loaded a tileset into your map `<div>`, the next step is to load the spatiotemporal information you prepared in the *.csv* file (located in the *data* folder) into your webpage; once loaded, this information is used to draw and animate the proportional symbols atop the tile service.

Two *.js* files—drawn from the Open Source community, as with the Leaflet library—are provided that support the dynamically loading and parsing of your spatiotemporal information: *EventTarget.js* and *ProcessCSV.js*. Together they convert comma separated information into an array of JavaScript objects, a data structure enclosed by curly braces that contains comma separated key/value pairs. The *.csv* information processed by *EventTarget.js* and *ProcessCSV.js* is returned to the *main.js* script and made accessible by the DOM. **Figure 4** illustrates how the information in the *.csv* file is converted into JavaScript objects, using the header names of the *.csv* file as the keys and populating the cell contents as the attribute values of the JavaScript objects.



	A	B	C	D	E	F	G	H	I	J	K	L
1	id	name	latitude	longitude	2005	2006	2007	2008	2009	2010	2011	
2	1	Atlanta	33.7489	-84.3881	85	38	75	30	9	15	38	
3	2	Chicago	41.85	-87.65	28	29	38	26	15	12	10	
4	3	Dallas	32.7828	-96.8039	18	59	22	60	82	42	18	

Figure 4a: Original *.csv* File.

```
[{id=”1”, name=”Atlanta”, latitude=”33.7489”, longitude,=”-
84.3381, 2005=”85”, 2006=”38”, 2007=”75”, 2008=”30”, 2009=”9”,
2010=”15”, 2011=”38”},{id=”2”, name=”Chicago”, latitude=”41.85”,
longitude=”-87.65”, ... ]
```

Figure 4b: Converted JavaScript Objects Accessible in the DOM.

The Open Source software movement is about sharing and collaboration, largely rejecting the use of strong copyright. Nonetheless, an important standard practice is to give credit to the original author of any script you incorporate into your final website. You should notice that these files each have a commented header section that provides attribution; the *ProcessCSV.js* includes other comments indicating how the original scripts were modified subsequently by other developers. The *.js* files make use of an MIT License (http://opensource.org/licenses/MIT), an open-source software license that grants the right to freely use and redistribute the software, so long as credit to the original author is maintained. It is strongly recommend that you add such licensing to your own scripts generated from G575, and required that you provide proper attribution for the source of modified scripts.

Note that the filename of each *.js* file begins with a capital letter. It is convention in the programming world to begin functions with a lowercase letter and classes with a capital letter. The JavaScript language does not have true classes, but you can create a pseudo-class using a ***prototype function***, a generic function that defines specific behaviors for manipulating and returning objects sent to the function through the function parameters. Prototype functions are one way to define behaviors in a format that is reusable both within a single application and across multiple applications.

Read through *ProcessCSV.js* (which again uses *EventTarget.js*) to understand what is happening at each stage in the flow of execution; the header box provides an executive summary of the script behaviors. An important trait of a polished web developer the ability to understand and modify existing code (i.e., customize a script); you will build on this skill incrementally over the course of the three G575 lab assignments.

To make use of the pair of prototype functions, copy the *EventTarget.js* and *ProcessCSV.js* files from Learn@UW and place them into your *js* project folder; you now should have four *.js* files in the *js* folder (the pair of prototype functions along with *leaflet-src.js* and *main.js*). You need to link to the pair of prototype functions within the `<body>` element of *index.html*, keeping *main.js* as the last linked *.js* file, again because it makes use of code in the prior three (**Code Bank 10: Lines 12-13**).

```
1       <body>
2               <!--div for the map-->
3               <div id="map"></div>
4
5               <!--libraries-->
6               <script src="js/leaflet-src.js"></script>
7
8               <!--link to CSV prototype functions-->
9               <script src="js/EventTarget.js"></script>
10              <script src="js/ProcessCSV.js"></script>
11
12              <!--link to main javascript file-->
13              <script src="js/main.js"></script>
14      </body>
```

Code Bank 10: Linking to *EventTarget.js* and *ProcessCSV.js* (in: *index.html*).

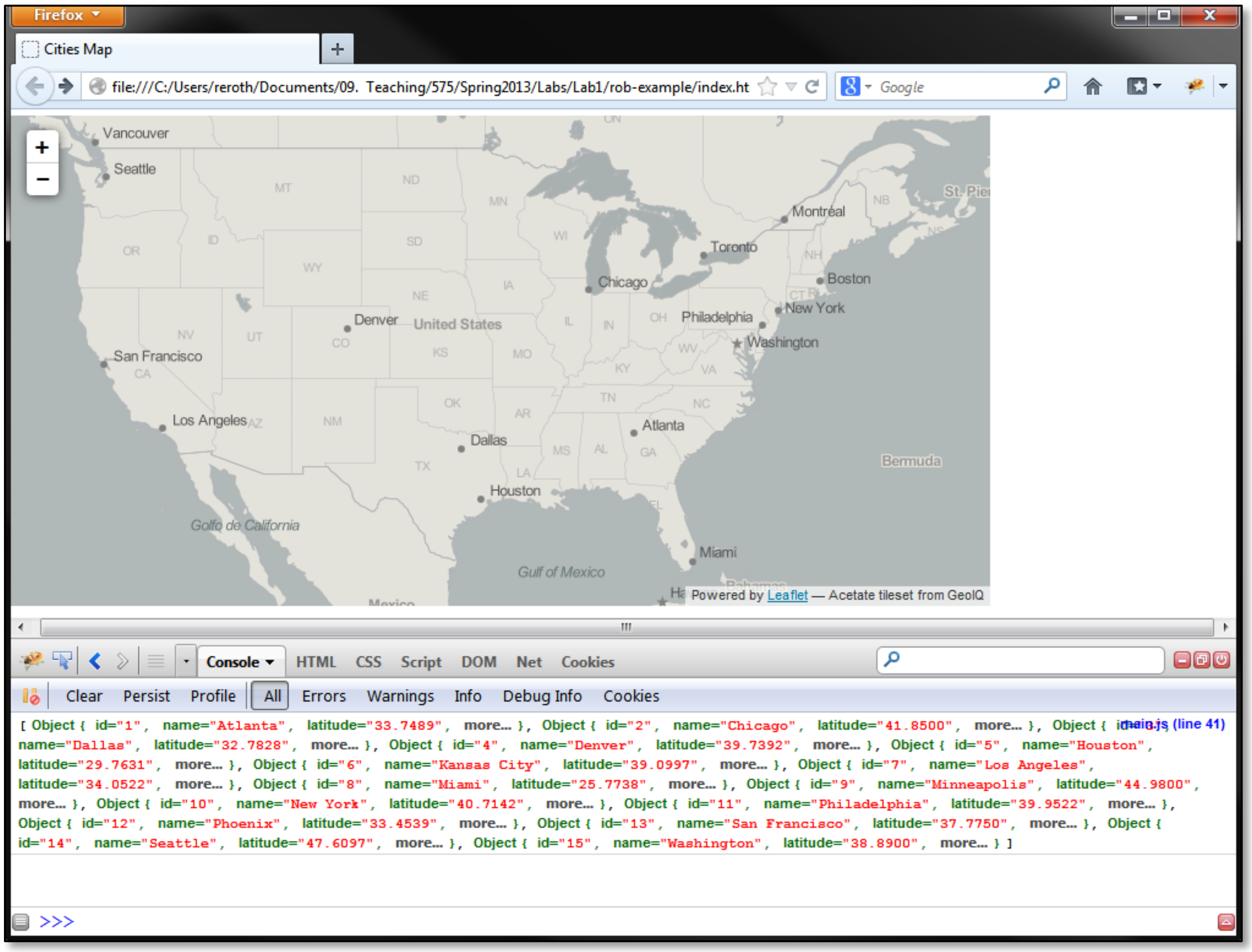
Return to *main.js* and first add a global array variable named `csvData` to store the contents of the *.csv* file as an array of JavaScript objects (**Code Bank 11**); it is recommended that you make this variable global so that you need to load the *.csv* file only once. Then add a new function named `processCSV()` that makes use of the *ProcessCSV.js* prototype function in order to populate the `csvData` array with the spatiotemporal information in your *.csv* file (**Code Bank 12**). The `processCSV()` function must be called at the end of the `setMap()` function (**Code Bank 12: Line 2**) so that the *.csv* file is loaded and drawn only after the map `<div>` has initialized and the basemap tileset has been loaded.

```
//global variables
var map; //map object
var csvData; //array of objects
```

Code Bank 11: Adding a Global Variable to Hold the *.csv* as JavaScript Objects

```
function processCSV() {
        //<-setMap()

        //process the csvData csv file
        var processCSV = new ProcessCSV(); //-> to ProcessCSV.js
        var csv = 'data/csvData.csv'; // set location of csv file

        processCSV.addListener("complete", function(){
                csvData = processCSV.getCSV(); //-> to ProcessCSV.js
                console.log(csvData);
        });

        processCSV.process(csv); //-> to ProcessCSV.js
};
```

Code Bank 12: Loading a Basemap using Leaflet (in: *main.js*).



```
[ Object { id="1", name="Atlanta", latitude="33.7489", more... }, Object { id="2", name="Chicago", latitude="41.8500", more... }, Object { id="3",
name="Dallas", latitude="32.7828", more... }, Object { id="4", name="Denver", latitude="39.7392", more... }, Object { id="5", name="Houston",
latitude="29.7631", more... }, Object { id="6", name="Kansas City", latitude="39.0997", more... }, Object { id="7", name="Los Angeles",
latitude="34.0522", more... }, Object { id="8", name="Miami", latitude="25.7738", more... }, Object { id="9", name="Minneapolis", latitude="44.9800",
more... }, Object { id="10", name="New York", latitude="40.7142", more... }, Object { id="11", name="Philadelphia", latitude="39.9522", more... },
Object { id="12", name="Phoenix", latitude="33.4539", more... }, Object { id="13", name="San Francisco", latitude="37.7750", more... }, Object {
id="14", name="Seattle", latitude="47.6097", more... }, Object { id="15", name="Washington", latitude="38.8900", more... } ]
```

main.js (line 41)

Figure 5: Printing the Loaded CSV Information to the Console.

The `processCSV()` function begins by creating a new variable instance (named `processCSV`) of the *ProcessCSV.js* prototype using the `new` keyword followed by a call to the `ProcessCSV()` constructor function (**Code Bank 12: Line 5**). A second, string variable (named `csv`) is created to store the relative location of the *.csv* file in the project directory (**Code Bank 12: Line 6**). An event listener then is added to the process `processCSV` variable to store the processed *.csv* file in the global `csvData` variable (**Code Bank 12: Lines 8-11**); the *.csv* file should be converted into JavaScript objects only after it has completely loaded into the browser, hence the use of the `complete` event (**Code Bank 12: Line 8**). The `processCSV()` function finishes execution by calling the `process()` function in *ProcessCSV.js* prototype definition, passing the `csv` string holding the directory location as the file parameter (**Code Bank 12: Line 13**).

Note that the event listener includes a statement to print the processed `csvData` array to the error console (**Code Bank 12: Line 10**). Return to Firefox and refresh your webpage. If you have modified your *main.js* code correctly, and configured your directory properly, you should see your complete *.csv* file printed to the console as an array of JavaScripts objects (**Figure 5**); clicking on an individual object navigates to the DOM tab, where you can inspect its properties.

b. Dynamically Drawing Markers onto the Tiles

Now that you have your spatiotemporal information loading dynamically into the browser, the next step is to add symbols atop the loaded tiles using the latitude and longitude coordinates within the *.csv* file. Leaflet supports the overlay of map symbols, or ***markers***, using either pre-rendered iconic point symbols (e.g., in *.png* format) or dynamically drawn ***scalable vector graphics*** (*.svg*); because you want to dynamically resize the markers to create proportional symbols, the latter option is more appropriate for Lab #1 (and thematic mapping generally). The *.svg* format is now common to the web, but, as stated in Section 1a, is not supported by older versions of Internet Explorer (before IE9) without a considerable work-around.

Begin by declaring a global variable of type array named `markersArray` to hold the complete set of markers added to the map (**Code Bank 13**). While each JavaScript object in the `csvData` array receives its own unique marker on the map, it is necessary to group these markers into a single array so that they subsequently can be added or restyled all at once. The `markersArray` variable needs to be global so that the map can be animated after first initialized (i.e., so that the `markersArray` variable can be accessed using multiple functions).

```
//global variables
var map; //map object
var csvData; //array of objects
var markersLayer; //markers layer group object
```

Code Bank 13: Declaring a Global Variable to Hold the Markers (in: *main.js*).

After declaring the `markersArray` variable, you then need to define a new function named `createMarkers()` that draws the set of *.svg* markers onto the map (**Code Bank 14**). The `createMarkers()` function should be called from the `processCSV()` function only after the `csvData` array is populated with JavaScript objects (as indicated in **Code Bank 14: Line 2**); to do this, replace the `console.log()` function in the `processCSV()` function (**Code Bank 12: Line 10**) with a call to `createMarkers()`.

The `createMarkers()` function begins by declaring three local variables needed to draw (the first two variables) or organize (the third variable) the set of markers (**Code Bank 14: Lines 4-14**). First, declare an integer variable named `r` and assign a default value of `10` (**Code Bank 14: Lines 4-5**); this value defines the default radius of each marker that later is scaled in proportion to the attribute value at a given timestamp (see Section 3c). Then, instantiate a new `markerStyle` variable in a manner similar to style rules in CSS (but with commas instead of semicolons), setting the `radius` (the previously defined `r` variable, which then can be updated programmatically) and `fillColor` (a default blue) similarly for all markers (**Code Bank 14: Lines 7-11**). You are encouraged to explore additional style options, as listed under L.Path in the Leaflet API Reference.

```
1       function createMarkers() {
2               //<-processCSV()
3
4               //radius of markers
5               var r = 10;
6
7               //marker style object
8               var markerStyle = {
9                       radius: r,
10                      fillColor: "#39F",
11              };
12
13              //create array to hold markers
14              var markersArray = [];
15
16              //create a circle marker for each object in the csvData array
17              for (var i=0; i<csvData.length; i++) {
18                      var feature = {};
19                      feature.properties = csvData[i];
20                      var lat = Number(feature.properties.latitude);
21                      var lng = Number(feature.properties.longitude);
22                      var marker = L.circleMarker([lat,lng], markerStyle);
23                      marker.feature = feature;
24                      markersArray.push(marker);
25              };
26
27              //create a markers layer with all of the circle markers
28              markersLayer = L.featureGroup(markersArray);
29
30              //add the markers layer to the map
31              markersLayer.addTo(map);
32      }
```

Code Bank 14: Adding Markers to the Map for the Proportional Symbols (in: *main.js*).

With the trio of variables declared, next create a `loop` to access individually each JavaScript object in the `csvData` array and create an associated marker for placement on the map (**Code Bank 14: Lines 16-25**). It does not matter how many objects are included in your `csvData` array, as the `length` of the array (**Code Bank 14: Line 17**) is used to determine how many times the conditional block of code is executing before breaking out of the loop (**Code Bank 14: Lines 18-**

24). For each JavaScript object in the `csvData` array, a local variable named `feature` is declared and assigned the values in the associated row (`i` standing for the current loop index) in the `csvData` array as its properties (**Code Bank 14: Lines 18-19**). The latitude and longitude then are extracted from the `feature` properties and stored in local variables `lat` and `long` (**Code Bank 14: Lines 20-21**); the keys "latitude" and "longitude" **must** use the same spelling and capitalization as used for the *.csv* file header (**Figure 1**). A new Leaflet `circleMarker` variable—named `marker`—then is created, passing the lat/long coordinates and the previously created `markerStyle` as the parameters (**Code Bank 14: Line 22**); the `circleMarker` object is used given the goal of producing a circular proportional symbol map, but is only one of many *.svg* vector layers that can be drawn atop the map. The Leaflet `marker` object is now created, but only contains the lat/long information. To add the timestamp information, set the local `feature` variable previously created as the `feature` property of the `marker` (**Code Bank 14: Line 23**). The conditional block of code completes by pushing the created `marker` object into the `markersArray` variable (**Code Bank 14: Line 24**).

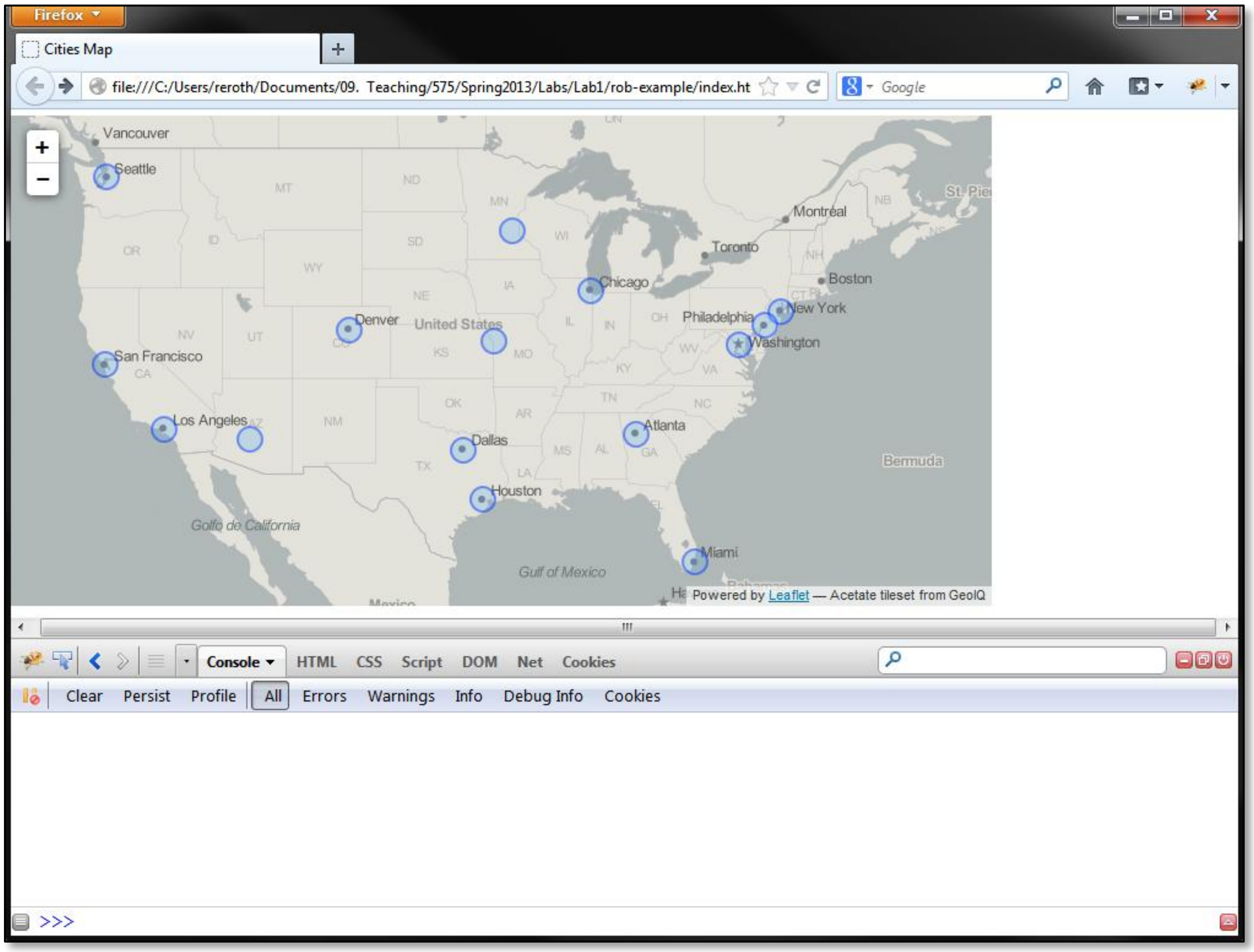


Figure 6: Drawing SVG Markers onto the Basemap.

With the `markersArray` array populated, you then can create the global `markersLayer` object, using the populated `markersArray` variable as the constructor parameter. `FeatureGroup` is a prototype function provided by Leaflet for organizing map layers and can be instantiated in long form (`new L.FeatureGroup(markersArray)`) or using the alias shortcut (**Code Bank 14: Lines 27-28**); refer to L.FeatureGroup in the Leaflet API Reference for details. Finally, use the `addTo()` function of `FeatureGroup` to add the `markersLayer` variable (and its contents) to the `map` object (**Code Bank 14: Lines 30-31**).

Return to Firefox and refresh your webpage. You know should see *.svg* vectors drawing atop the loaded map tiles, positioned according to the latitude and longitude of your mapped features (**Figure 6**)!

c. Dynamically Scaling the Markers

After drawing the *.svg* markers to the map, you now need to add the functionality to resize each marker according to an attribute value. Such a function needs to be applied uniquely to each marker in the newly created `markersLayer` variable, as each proportional symbol on your map could have a different attribute value (i.e., a differently sized proportional symbol), and these attribute values vary over the included set of timestamps.

The `FeatureGroup` prototype (of which `markersLayer` is an instance) extends the `LayerGroup` prototype in the Leaflet source code, meaning that it includes all properties and methods of the `LayerGroup` definition, as well as several unique properties and functions within its own definition (refer back to L.LayerGroup in the Leaflet API Reference for details). The `LayerGroup` prototype definition includes a useful method named `eachLayer()`, which applies a custom method of your own creation to every feature in a `FeatureGroup` instance (i.e., to every marker in `markersLayer`).

To make use of the `eachLayer()` method to resize your markers, first add a pair of global variables at the top of *main.js*: (1) an integer named `timestamp` that holds the current timestamp shown on the map (**Code Bank 15: Line 5**); the `timestamp` variable should be assigned the header value of the first timestamp in your *.csv* file (e.g., the year `2005` from **Figure 1**); and (2) an integer named `scaleFactor` that determines the mathematical scaling ratio of your proportional symbols (**Code Bank 15: Line 5**); experiment with different values to identify an optimal scaling ratio at the smallest cartographic scale (think back to discussion about proportional symbol scaling in G370). This pair of variables should be global, as you are likely to manipulate them in Lab #2 when implementing cartographic interaction (e.g., enable the user to change the `timestamp` or the `scaleFactor` interactively).

```
1       //global variables
2       var map; //map object
3       var csvData; //array of objects
4       var markersLayer; //markers layer group object
5       var timestamp = 2005; //initial timestamp
6       var scaleFactor = 25; //scale factor for marker area
```

Code Bank 15: Adding Global Variables for Proportional Symbols (in: *main.js*).

After adding the pair of global variables, insert a call to the `eachLayer()` method at the end of the `createMarkers()` function that you defined in Section 3b (**Code Bank 14**). This function call should come after the call to `markersLayer.addTo(map)` (**Code Bank 14: Lines 30-31**). The `eachlayer()` method defines a new function as its parameter, further passing the `layer` as the parameter of this new function (**Code Bank 16: Lines 5-8**). The solution in **Code Bank 16** calls yet a third function named `onEachFeature()`, also passing it the `layer` as the parameter. The `onEachFeature()` function is included so that the radius of each marker can be updated during the animation as well as during the initial drawing of the map (see Section 3d); in other words, having the resizing functionality with the `onEachFeature()` function allows this code to be reused multiple times within the script.

Why is this method named `eachLayer()` and not `eachFeature()`? Because Leaflet considers each `circleMarker` its own layer, the `markersLayer` is a distinct, composite object holding this set of layers. The 'feature' is the geographic object embedded in the `layer` variable, while the `layer` contains the properties, methods, and events Leaflet gives it to make it visible and interactive. If you're confused, inspecting the layer group with the `console.log()` function may make it clearer.

```
1       function createMarkers() {
2
3               // code from Code Bank 13 removed here for brevity
4
5               //call the function to size each marker and add its popup
6               markersLayer.eachLayer(function(layer) {
7                       onEachFeature(layer);//->
8               })
9       }
```

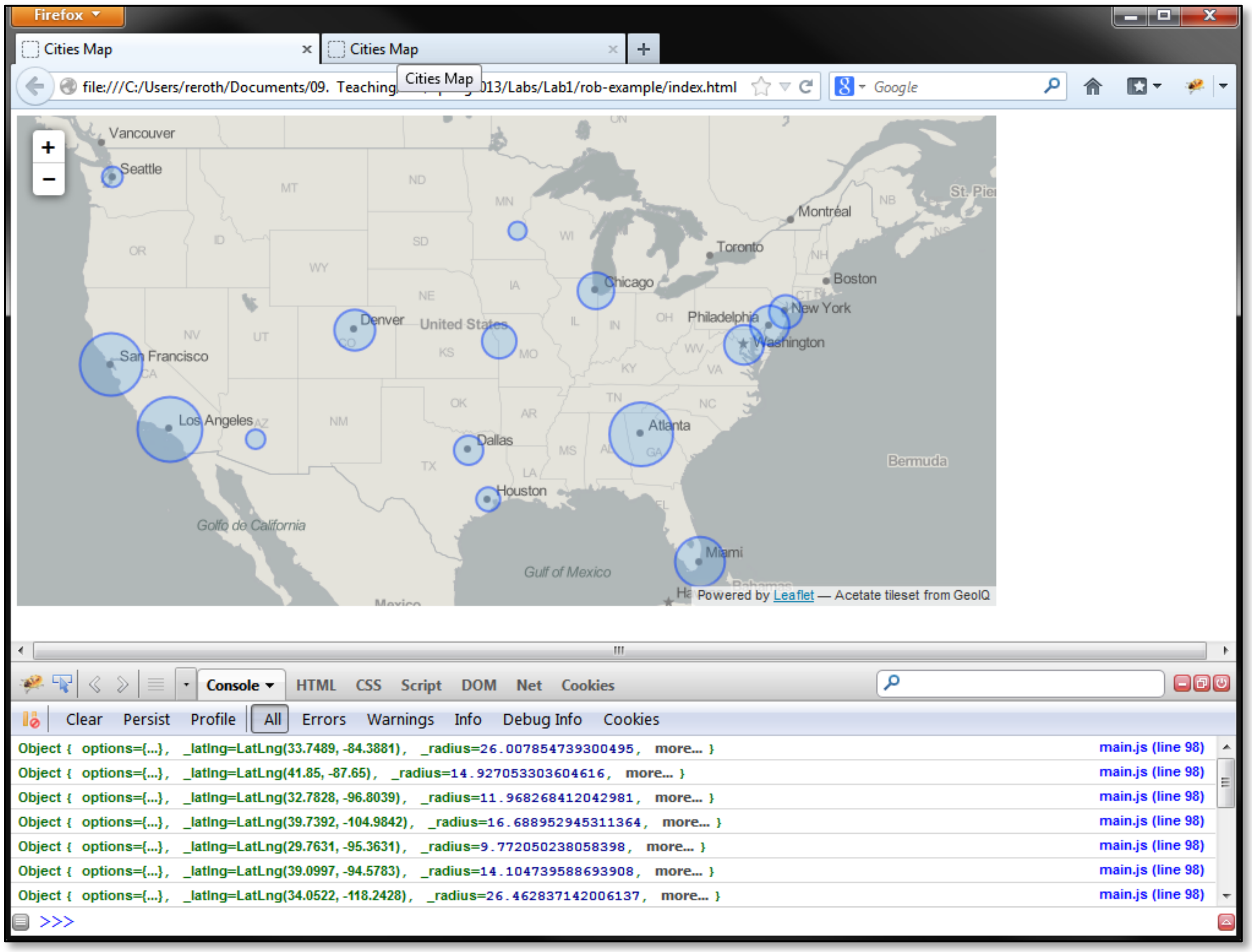
Code Bank 16: Adding a Call to the `eachLayer()` function (in: *main.js*).

The final step in resizing the proportional symbols is to define the `onEachFeature()` function, which performs the geometric calculations needed to calculate the radius of the scaled proportional symbol from its attribute value (**Code Bank 17**). The function first calculates the `area` of the symbol by multiplying the attribute value of the `layer` at the current `timestamp`—which is located in the DOM under `layer.feature.properties`—against the `scaleFactor` (**Code Bank 17: Lines 4-5**). Basic geometry then is applied to convert the `area` into a `radius` value, which is needed for drawing the marker in Leaflet (**Code Bank 17: Lines 4-5**). The marker then is redrawn with the newly calculated radius using the `setRadius()` method of `circleMarker` objects in Leaflet (**Code Bank 17: Lines 4-5**).

Again, `onEachFeature()` will be called individually for every `marker` stored in the `markersLayer` variable (thus, at least 15 times). A call to `console.log()` is added at the end of the `onEachFeature()` function to print the layer's properties to the error console. Once you have added the code from **Code Banks 15-17**, return to Firefox and refresh the webpage; you should see the *.svg* markers scaling proportionately to their attribute values in the first timestamp and see the complete `layer` properties printing to the error console (**Figure 7**).

```
function onEachFeature(layer) {
        //<-createMarkers()

        //calculate the area based on the data for that timestamp
        var area = layer.feature.properties[timestamp] * scaleFactor;

        //calculate the radius
        var radius = Math.sqrt(area/Math.PI);

        //set the symbol radius
        layer.setRadius(radius);

        console.log(layer);
}
```

Code Bank 16: Calculating the Radius for Each Proportional Symbol (in: *main.js*).



```
Object { options={...}, _latlng=LatLng(33.7489, -84.3881), _radius=26.007854739300495, more... }   main.js (line 98)
Object { options={...}, _latlng=LatLng(41.85, -87.65), _radius=14.927053303604616, more... }   main.js (line 98)
Object { options={...}, _latlng=LatLng(32.7828, -96.8039), _radius=11.968268412042981, more... }   main.js (line 98)
Object { options={...}, _latlng=LatLng(39.7392, -104.9842), _radius=16.688952945311364, more... }   main.js (line 98)
Object { options={...}, _latlng=LatLng(29.7631, -95.3631), _radius=9.772050238058398, more... }   main.js (line 98)
Object { options={...}, _latlng=LatLng(39.0997, -94.5783), _radius=14.104739588693908, more... }   main.js (line 98)
Object { options={...}, _latlng=LatLng(34.0522, -118.2428), _radius=26.462837142006137, more... }   main.js (line 98)
>>>
```

Figure 7: Dynamically Scaling the Markers.

d. Animating the Proportional Symbols

The final required step of Lab #1 is animating the proportional symbols. As stated in lecture, while animation and interaction together are considered a part of "dynamic" cartography, they are fundamentally different, as animation is a change to the display evoked by the system while interaction is a change to the display evoked by the user. Lab #2 will build upon the spatiotemporal visualization developed in Lab #1 by adding interaction.

To animate your proportional symbols, first add two global variables (**Code Bank 18: Lines 6-7**): (1) a `timer` variable that provides the system control for updating the display and (2) a `timerInterval` integer that sets the speed (in milliseconds) of the animation. This pair of variables needs to be global so that animation can be manipulated using interaction (e.g., VCR controls, a temporal slider bar) in Lab #2.

```
1       //global variables
2       var map; //map object
3       var csvData; //array of objects
4       var timestamp = 2005; //initial timestamp
5       var scaleFactor = 25; //scale factor for marker area
6       var timer; //timer object for animation
7       var timerInterval = 1000; //initial animation speed in milliseconds
```

Code Bank 18: Adding a Global Variable for the Animation (in: *main.js*).

The animation behavior requires a pair of functions. First, define a new function called `animateMap()` that is called at the end of the `setMap()` function definition (**Code Bank 19**). The `animateMap()` function instantiates the `timer` variable by calling the `setInterval()` function (**Code Bank 19: Lines 4-6)**. The `setInterval()` function in JavaScript takes two parameters: (1) a function to be called at a regular interval (a custom function named `step()`) and (2) the interval itself (using the global `timerInterval` variable). What this means is that the custom `step()` function is called by the system every 1000 milliseconds (or whatever value you assigned to `timerInterval`).

It is the `step()` function that provides the logic to change the proportional symbols based on a new timestamp (**Code Bank 20**). An `if-else` statement is used to check if the current `timestamp` is the last in the animation (**Code Bank 20: Lines 4-9**); **Code Bank 20** "hard codes" this first and last timestamp values (`2005` and `2011` respectively; you need to update these values based on your assembled *.csv* file), but this can be determined programmatically using the `length` property. If the current timestamp is not the last in the animation, then the `timestamp` variable is incremented by one (**Code Bank 20: Line 6**). If the current timestamp is the last in the animation, then the `timestamp` variable is set to the first timestamp, looping the animation (**Code Bank 20: Line 8**). Once the timestamp variable is updated, the `eachLayer()` function of the `markersLayer` variable again is called (**Code Bank 20: Lines 11-14**), effectively removing the proportional symbols from the map and redrawing them using new radius values.

```
function animateMap() {
        //<-setMap();

        timer = setInterval(function(){
                step();//->
        },timerInterval);
}
```

Code Bank 19: Adding a Global Variable for the Animation (in: *main.js*).

```
function step(){
        //<-animateMap()

        //cycle through years
        if (timestamp < 2011){ //update with last timestamp header
                timestamp++;
        } else {
                timestamp = 2005; //update with first timestamp header
        };

        //upon changing the timestamp, call onEachFeature to update the display
        markersLayer.eachLayer(function(layer) {
                onEachFeature(layer);//->
        });
}
```

Code Bank 20: Adding a Global Variable for the Animation (in: *main.js*).

Return to Firefox one last time and refresh *index.html*. You now should have an animated proportional symbol map, completing the spatiotemporal visualization required for the Lab #1 challenge!

4. Bonus: Adding Legends

a. Transitioning What You Have Learned

The instructions and code banks in Sections #1-3 provide you with all of the details needed to prepare your spatiotemporal visualization. You are encouraged to take what you have learned from these steps and add two legends: one for explaining the meaning of the proportional symbols on the map and one for explaining the meaning of the temporal animation. You will be awarded two bonus points for each legend you add on your own, if your solution is fully functional at the time of submitting Lab #1. Refer to the Slocum textbook for examples of a proportional symbol legend (e.g., stacked versus nested) and a temporal animation legend (e.g., a timeline versus a digital clock). You will be provided with solutions for both after submitting Lab #1, as both legends must be included in your Lab #2 design.

Evaluation Rubric: Animation Challenge (20pts)

You are required to upload a *.zip* of your entire directory to the Learn@UW Lab #1 Dropbox at least one hour before your lab on February 21st, 2013.

Representation (15)

(5) Basemap Tiles (tiles loading with correct scale/centering)
(5) Proportional Symbols (markers added to map that are properly scaled)
(5) Animation (looping through timestamps when loading the webpage)

Design for Scenario (5)

(1) Appropriate Spatiotemporal Information Set
(1) Overall Consideration of Scenario
(3) Webpage Design (surrounding HTML5/CSS elements)

Legends (Bonus Points)

(+2) Proportional Symbol Legend
(+2) Temporal Animation Legend