

**Software Development**

Greensboro, NC

2015

Table of Contents

[Research 3](#_Toc414478442)

[Description 4](#_Toc414478443)

[Plan of Work Log 5](#_Toc414478444)

[Documentation 7](#_Toc414478445)

[Project Requirements 7](#_Toc414478446)

[High-Level Software Design 8](#_Toc414478447)

[Testing 9](#_Toc414478448)

[End-User Documentation 73](#_Toc414478449)

[Evaluation 75](#_Toc414478450)

[References 76](#_Toc414478451)

# Research

Before and during the development of EcoCartographer extensive research was required to create the final version. Over the course of the last several months gas prices have risen and dropped and are on the rise again, and with gas prices some of the highest they have been in years the price of gasoline is on the minds of many drivers. In efforts to create a fuel saving and money saving product EcoCartographer was created as a program that helps drivers find the most fuel efficient route to their destination. Research was conducted to find other applications with similar purposes or functionality, and there is only one other, which is EcoSpeed.

In order to get this project underway research had to be done on optimal speed for gas consumption as well as the force required to make cars move along a segment of road. This was helped by a member of our chapter team, Hunter Chen. We learned that rolling friction, drag, and elevation are the factors in determining this, and we looked up equations from the Engineering Toolbox, and then double-checked the code with him. He has taken physics courses.

Additionally, information needed to be gathered from the Google Documentation about how to use the Maps API, and general reference to software development documentation sites like W3Schools and Q&A sites like StackOverflow.

During the preliminary stages of EcoCartographer we received help from Sayari Ghosh, a professional software developer, thanks to the help of chapter advisor Monika Vasili; she gave our team suggestions on good algorithms to use, such as Djikstra’s algorithm for finding the best routes, and supported the original idea of using a grid of geolocated points to build a network of routes.

Vehicles largely vary in weight, size, and torque—all important factors that affect the fuel economy of vehicles. Extensive research went into how to find the fuel efficiency of a vehicle at any given output force. There appears to be no sort of specific fuel-efficiency data for the vast majority of vehicles of high enough quality to use in this application. Linear systems taken from the two city/highway data points that all cars are required to have do not match the curve associated with Brake Specific Fuel Consumption, and would not be very accurate. The amount of fuel used per unit of output is high at low speeds, and decreases down to a minimum around 2000rpm on most vehicles, and then rises again as speeds get very high.

During the coding process, information about syntax, control structures, prototype functions, API usage, etc. was gathered from a variety of sources such as StackOverflow, W3Schools, Google Maps API Reference, and the Overpass API Wiki, among others, and applied directly to the code.

# Description

The problems this project makes an effort in solving are that people need to spend large portions of their incomes to buy fuel, and that vast quantities of gasoline are burnt, producing climate-change-inducing carbon dioxide.

One of the roots of this problem is that people do not choose the most fuel efficient route, generally. We know from experience that this is true. The fuel-efficient routes are usually on relatively slow roads and therefore take longer to go on. Additionally, software to calculate the most fuel-efficient route is not very widespread, and there is an availability issue.

This project adds to the solution to the problem by providing a quick and free means of finding such a route. This is done by using publically available map data from Google Maps and OpenStreetMap, which is an open-source map database, to which the team has contributed information about local roads. This data is then formatted into a network, with edges and nodes, and the optimal path from start to end is found using this network. The interface is simple to use, with an input box featuring intuitive controls, and a console, which allows the user to see the progress of the algorithm. There are also instructions for usage included, as well as a means of connecting with the team to suggest improvements or report issues.

While this software is not a silver bullet, this project is significant in society by providing to people the ability to make their fuel go further. It allows anyone who owns a motor vehicle to save money on gasoline or diesel. In these tough economic times, this can be truly useful, especially to the poorest Americans. It can be applied to any industry in which transportation of goods is important, and allow companies to save money on fuel, should they choose so. It gives guidance to drivers on how to use less fuel, and in turn lessens the emissions of greenhouse gases like carbon dioxide and other pollutants. Finally, it can help to lessen the dependence we all have on the scarce resource of oil, and increase the energy independence of the United States. Through the use of this application in conjunction with other technologies like more efficient cars, a huge impact can be made on the above issues.

# Plan of Work Log

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date** | **Task** | **Time Involved** | **Team Member Responsible** | **Comments** |
| 1. 9/11/14 | Research | 1.5 hours | Daniel, Collin | Today we started research on fuel efficiency and mpg |
| 1. 9/13/14 | Research | 1 hour | Collin | Further research into fuel efficiency for specific car models |
| 1. 9/18/14 | Research | 1.5 hours | Daniel, Collin | Today I looked into the force required for movement with cars |
| 1. 9/25/14 | Research | 1 hour | Collin | Today I looked into fuel pricing and rise in need over time |
| 1. 10/2/14 | Research | 2 hours | Collin | Research is finished website design will start found final information |
| 1. 10/4/14 | Version 1.1 | 4 hours | Daniel | Basic Google Maps query, set up the basics |
| 1. 10/8/14 | Website design | 1 hour | Collin | Today we had a software developer talk to us about programming |
| 1. 10/9/14 | Website design | 1 hour | Collin (Daniel) | Today we started creating the actual webpage. |
| 1. 10/10/14 | Version 1.2 | 6 hours | Daniel | Experimented with grid technique to build up a network of roads |
| 1. 10/16/14 | Version 1.3 | 5 hours | Daniel | Experimented with generating a network of roads using a circular network of directions queries |
| 1. 10/16/14 | Website design | 1 hour | Collin (Daniel) | Today I worked on creating gradient boarders as well as other initial design |
| 1. 10/23/14 | Website design | 2 hours | Collin | Today I experimented with new design techniques as well as a potential logo |
| 1. 10/30/14 | Website design | 1 hour | Collin | Today I figured out how to create a button with an email link to default program |
| 1. 11/6/14 | Website design | 1 hour | Collin | Having issues with implementing coding still trying to find the issue |
| 1. 11/11/14 | Version 1.4 | 8 hours | Daniel | Method for analyzing frequencies of roads by dissecting the written instructions returned by directions service. Measuring point frequency |
| 1. 11/13/14 | Website design | 1.5 hours | Collin | Was able to fix error in coding website design is still coming along well |
| 1. 11/20/14 | Website design | 1 hour | Collin | Worked on creating new title possibly a custom logo |
| 1. 12/04/14 | Website design | 1.5 hours | Collin | Created a new tool bar with links to helpful pages |
| 1. 12/6/14 | Version 1.5 | 9 hours | Daniel | Used method of slicing path to generate intersection network. |
| 1. 2/4/15 | Version 1.6 | 60 hours | Daniel | Adding elevation data, finding data for next and last point in sequence. Cleaned up code. Multiple routes for travel used to build up point network. Changed search algorithm for intersections to ends of paths. Overpass API query for speed limit data, parsing that data. Took much time to figure out how to do that. Calculates fuel expenditures using elevation/speed data. Research done on how to calculate fuel efficiency at output of force, but nothing productive turned up. |
| 1. 2/17/15 | Version 1.7 | 12 hours | Daniel | Branched through intersections to find possible routes. Calculated total force for each route. Made instructions for this route, and a polyline on the map. |
| 1. 2/19/15 | Version 2.1 | 3 hours | Daniel | Tried to get elevation data from an XMLHttpRequest, but that failed. |
| 1. 2/24/15 | Version 2.2 | 3 hours | Daniel | Added sample icons for the output on the map. Modified stop light/stop sign modification algorithm to not have duplicates in the same way. |
| 1. 3/10/15 | Version 2.3 | 11 hours | Daniel | Discovered alternate data in the google maps directions service, and substituted that for the Overpass data. Continued research on integrating engine-specific factors into cost. Integrated acceleration/deceleration into the algorithm. Made an output HTML sheet to display the map and instructions on. Improved number of elevation queries to be made. Replaced XMLHttpRequest with Ajax. |

Advisor Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*All dates for Daniel are dates that the task was completed on. Also, Time Involved is the approximate amount of time needed to make/document the version.

# Documentation

## Project Requirements

This program is known to run correctly on the web browsers Google Chrome (recommended) and Mozilla Firefox. It does not run correctly on Internet Explorer or Safari (Windows).

Additionally, testing was conducted only on the Windows operating system, and while it theoretically works nominally on the Apple and Linux operating systems, this is not confirmed and therefore not recommended for use of this application.

It has been tested on laptops and desktops. The files have not been uploaded to a smart phone nor have they been placed on a server for remote access.

Some policies of the internet service provider of the user or servers in the middle of the exchange of data may cause the program to not run correctly (i.e. blocking some data requests).

## High-Level Software Design

This project requires the inputs of a start location, an end location, and specifications for the vehicle to be used, including frontal surface area, mass, and drag coefficient.

The start location and end location are collected in the GetRouteFromInput function from the HTML document, and parsed through the Google Geocoding Service into a LatLng object. This is passed to the Network function through the two arguments.

Inside the Network function, there are several nested functions, so that they can use a pool of variables local to the Network function, and not need an excessive number of arguments. In the beginning of the Network function, the shared variables are created, and a combination of various Google Maps Directions Service Queries are put together, using a net shape of waypoints in a circular arrangement. The purposes of these are to create a network of roads which can be traveled on. The queries are passed to a linking function called SendDirRequests, which sequences the queries and keeps them in order, since the Service is asynchronous. The SendDirRequests function passes the queries at the appropriate time to the getDirs function, which collects the Directions data, and compiles it into a series of points with different attributes that are collected directly from the Directions Service.

On the last query, the PointReferences and getBounds functions are called, to compile lists of the indexes of duplicate coordinates in the pts array, and get the total Bounds of the network of roads, respectively. overpass\_signals function is run after that, using the bounds generated from getBounds to find all the traffic signals in the bounds in the OpenStreetMap database. This is done using Ajax.

Finally, the elevation function is run, and it collects elevation data for all the unique coordinates in the network (that is the purpose of the PointReference function). It assigns the data to the points, and also generates additional point-to-point data in the pts variable in the SetData sub-function by storing the data in “next” and “last” attributes in each individual point.

After the elevation data is collected and assigned, the compute function is executed. This function controls other functions. It begins by assigning the amount of force needed to get to the next/last point to all the points in the pts variable. After, it executes the add\_signals function, which assigns the traffic signals collected using the Overpass API to intersections; If not intersections, then ordinary points. It also checks that there are not consecutive lights. It then builds a list of intersections, which have a list of connections to any other intersections it connects to, along with a path and force quanities.

These intersections are then used in a tree-branch finding function called Branches. It is recursive on itself, and finds all possible paths through the intersections from start to end, without going to the same intersection twice. A quick cost is generated for each branch. The optimal branch is found, and a list called candidates is the other possible branches that could be competitive with the optimal branch. This list of branches is sent through a more complicated force calculating function called GetDetailedCost, and the optimal candidate is found based off of that data.

Finally, instructions are generated from Google Geocoding API data, and select information is sent via URL to the output window.

## Testing

**OUTPUTS**

Version 1.1

**Summary:**

Set up a Google Map, and generate a route and examine the output.

**Outputs:**

*Output 1.1.1:*

Object {origin: df, destination: df, waypoints: Array[0], travelMode: "DRIVING"}destination: dfB: -80.39394299999998k: 36.062835\_\_proto\_\_: dfconstructor: function df(a,b,c){a-=0;b-=0;c||(a=le(a,-90,90),180!=b&&(b=me(b,-180,180)));this.k=a;this.B=b}equals: function (a){return a?ne(this.lat(),a.lat())&&ne(this.lng(),a.lng()):!1}j: function (a){return a?ne(this.lat(),a.lat())&&ne(this.lng(),a.lng()):!1}lat: function (){return this[a]}lng: function (){return this[a]}toString: function (){return"("+this.lat()+", "+this.lng()+")"}toUrlValue: function (a){a=ue(a)?a:6;return gf(this.lat(),a)+","+gf(this.lng(),a)}\_\_proto\_\_: Objectorigin: dfB: -80.28694000000002k: 36.0941915\_\_proto\_\_: dftravelMode: "DRIVING"waypoints: Array[0]\_\_proto\_\_: Object script.js:22

Object {routes: Array[1], status: "OK", oc: Object}oc: Objectdestination: dforigin: dftravelMode: "DRIVING"waypoints: Array[0]\_\_proto\_\_: Objectroutes: Array[1]0: Objectbounds: LgEa: Jgj: 36.0942k: 36.06255\_\_proto\_\_: Jgva: Eg\_\_proto\_\_: Lgcopyrights: "Map data ©2014 Google"legs: Array[1]0: Objectdistance: Objecttext: "7.6 mi"value: 12265\_\_proto\_\_: Objectduration: Objecttext: "9 mins"value: 543\_\_proto\_\_: Objectend\_address: "6941 Millbridge Road, Clemmons, NC 27012, USA"end\_location: dfB: -80.39387090000002k: 36.0632875\_\_proto\_\_: dfstart\_address: "342-378 Knollwood Street, Winston-Salem, NC 27103, USA"start\_location: dfsteps: Array[6]0: Objectdistance: Objectduration: Objectencoded\_lat\_lngs: "wsh{El\_`iNtEOhBC"end\_location: dfend\_point: dfinstructions: "Head <b>south</b> on <b>Knollwood St</b> toward <b>Hanover Arms Ct</b>"lat\_lngs: Array[3]maneuver: ""path: Array[3]0: dfB: -80.28679000000001k: 36.0942\_\_proto\_\_: df1: df2: dflength: 3\_\_proto\_\_: Array[0]polyline: Objectpoints: "wsh{El\_`iNtEOhBC"\_\_proto\_\_: Objectstart\_location: dfstart\_point: dftravel\_mode: "DRIVING"\_\_proto\_\_: Object1: Objectdistance: Objectduration: Objectencoded\_lat\_lngs: "wih{Ex~\_iNPXDHPVbAxAn@|@NTPVLTHPDHDHDH@DBDBB@@B@HDrBzFh@zAb@pANl@Rf@p@rB^hAJZTx@Jd@Pv@XdBf@nCV`BX~AJl@`@xB`@zB^xB^xBLn@RfAh@lCf@zBn@fCp@jCj@vBVt@X`AN^Vv@BDZ~@dAbCBFh@tAVl@Vl@Zr@\t@Xn@d@~@Vd@~@fB`@x@DFNXXd@f@z@nK|P|@xApCvEfCfEhEdHdAlBlAtBNXvCvF@@P^rCzFRf@~CbHrBtE`@jA~@rC"end\_location: dfend\_point: dfinstructions: "Turn <b>right</b> to merge onto <b>I-40BUS W/US-421 N</b>"lat\_lngs: Array[85]maneuver: ""path: Array[85]polyline: Objectstart\_location: dfstart\_point: dftravel\_mode: "DRIVING"\_\_proto\_\_: Object2: Object3: Object4: Object5: Objectlength: 6\_\_proto\_\_: Array[0]via\_waypoint: Array[0]length: 0\_\_proto\_\_: Array[0]via\_waypoints: Array[0]length: 0\_\_proto\_\_: Array[0]\_\_proto\_\_: Objectlength: 1\_\_proto\_\_: Array[0]overview\_path: Array[91]0: dfB: -80.28679000000001k: 36.0942\_\_proto\_\_: df1: dfB: -80.28669000000001k: 36.092600000000004\_\_proto\_\_: df2: dfB: -80.28687000000001k: 36.09248\_\_proto\_\_: df3: df4: df5: df6: df7: df8: df9: df10: df11: df12: df13: df14: df15: df16: df17: df18: df19: df20: df21: df22: df23: df24: df25: df26: df27: df28: df29: df30: df31: df32: df33: df34: df35: df36: df37: df38: df39: df40: df41: df42: df43: df44: df45: df46: df47: df48: df49: df50: df51: df52: df53: df54: df55: df56: df57: df58: df59: df60: df61: df62: df63: df64: df65: df66: df67: df68: df69: df70: df71: df72: df73: df74: df75: df76: df77: df78: df79: df80: df81: dfB: -80.38947k: 36.06255\_\_proto\_\_: df82: df83: df84: df85: df86: df87: df88: df89: df90: dflength: 91\_\_proto\_\_: Array[0]overview\_polyline: "wsh{El\_`iN~HSVb@tCdEn@hAP^RL|CvIr@~BpB`G`@~Aj@|CdB~JbChN`@vBpAhG`BrGbAlDdA~CnC`HhBdEvC|FpA`CvLxRnEpHpIlNrCbFzDrHfDbHrGxNjCbIhAbEbAtEhAnGfBlKj@lEVbC\hHBjDE|Eo@`^}@he@MxKC~FFvKVbZJjOCtEM`Ei@~IqCv]gGzx@mEfl@mA~OKTKp@g@fCq@dBw@xAwArBe@dAQj@YdBIj@VLbDfBfAd@tDtAxAZvC\jH`@nN`Ar@DnBD`BGzD\_@fQkBtEe@x@GdAC~CD~BBChCEfDGhAOl@Wf@s@rAOj@In@KlF"summary: "I-40BUS W and US-421 N"warnings: Array[0]length: 0\_\_proto\_\_: Array[0]waypoint\_order: Array[0]length: 0\_\_proto\_\_: Array[0]\_\_proto\_\_: Objectlength: 1\_\_proto\_\_: Array[0]concat: function concat() { [native code] }constructor: function Array() { [native code] }every: function every() { [native code] }filter: function filter() { [native code] }forEach: function forEach() { [native code] }indexOf: function indexOf() { [native code] }join: function join() { [native code] }lastIndexOf: function lastIndexOf() { [native code] }length: 0map: function map() { [native code] }pop: function pop() { [native code] }push: function push() { [native code] }reduce: function reduce() { [native code] }reduceRight: function reduceRight() { [native code] }reverse: function reverse() { [native code] }shift: function shift() { [native code] }slice: function slice() { [native code] }some: function some() { [native code] }sort: function sort() { [native code] }splice: function splice() { [native code] }toLocaleString: function toLocaleString() { [native code] }toString: function toString() { [native code] }unshift: function unshift() { [native code] }\_\_proto\_\_: Objectstatus: "OK"\_\_proto\_\_: Object

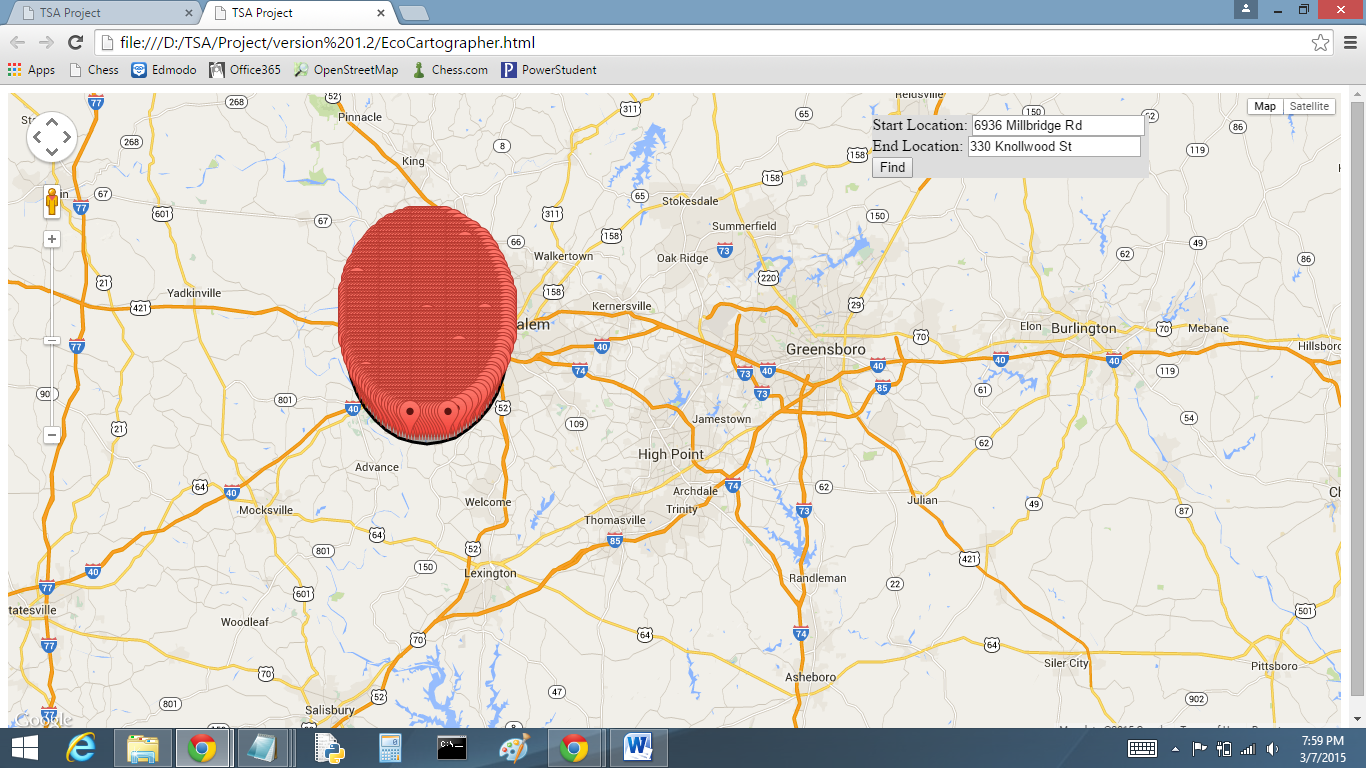
Version 1.2

**Summary:**

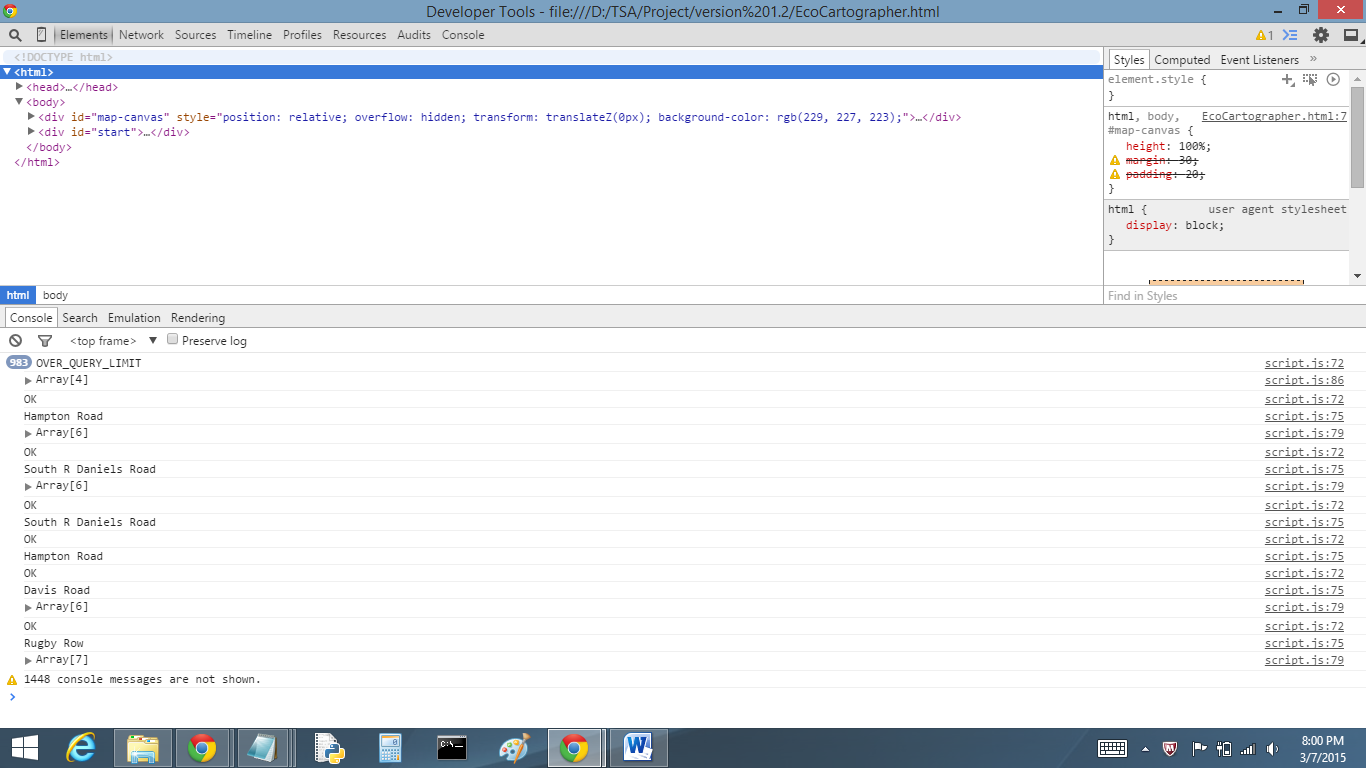
Experiment with using a grid of geocoded points to find a network of roads. A visual demonstration is the circle of markers shown below. This failed, because there were repeated “Over Query Limit” error messages.

**Outputs:**

*Output 1.2.1:*



*Output 1.2.2:*



**New code:**

*Code 2.1.1:*

function Grid(sp,ep){

var dlat = sp.lat()-ep.lat(); //difference lat/lng

var dlng = sp.lng()-ep.lng();

var center = new google.maps.LatLng(sp.lat()-dlat/2, sp.lng()-dlng/2); // center between dest and orig

var cm = new google.maps.Marker({position:center, map:map}); //center marker

var hdist = Math.sqrt(dlat\*dlat + dlng\*dlng); //half dist from orig to dest

//basic measures

//function checks if a point is in the circle with center at <center> and radius of <hdist>

function InCircle(lat,lng){

if(Math.pow(lat-center.lat(),2)+Math.pow(lng-center.lng(),2) < Math.pow(hdist,2)){

return true;

}else{

return false;

}

}

var lpts = [];

var Roads = [];

for(var i=0; i<Math.PI\*2; i+=Math.PI/36){

lpts.push(new google.maps.LatLng(center.lat()+hdist\*Math.sin(i), center.lng()+hdist\*Math.cos(i)));

}

var Poly = new google.maps.Polygon({path:lpts});

Poly.setMap(map);

//draws circle

for(var i=center.lat()-hdist; i<center.lat()+hdist; i+=0.004){

for(var j=center.lng()-hdist; j<center.lng()+hdist; j+=0.004){

//iterates through grid with 0.004 deg intervals

if(InCircle(i,j)){ // checks if coord is in circle

var latlng = new google.maps.LatLng(i,j);

var Marker = new google.maps.Marker({position:latlng, map:map});

//finds address

geocoder.geocode({'latLng':latlng}, function(results, status){

console.log(status);

if(status==google.maps.GeocoderStatus.OK){

var street = results[0].address\_components[1].long\_name;

console.log(street);

if(Roads.indexOf(street) === -1){

//adds road name to <Roads> if it is not there already

Roads.push(street);

console.log(results);

}

}

});

}

}

}

console.log(Roads);

//Output <Roads> list

}

Version 1.3

**Summary:**

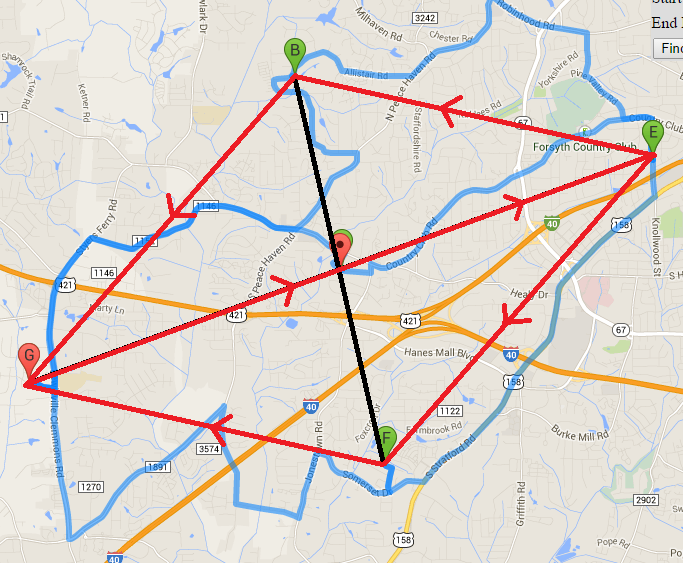
Tried different variations of methods to generate roads and intersections.

1. (Version 1.2) Make a grid, geocode the coordinates, and make a list of unique roads. List of intersection objects with information about paths to other intersections is made. Failed because of geocoding limits to 10 requests per second after creation of grid in a circle around two test places. There were approximately 2400 points to be geocoded. Since the shortest route is relatively short (~7.7 miles, 9 minutes), it would be unfeasible for even short routes. Sayari Ghosh originally supported this idea, or some variant on in.
2. Make a circle, and generate a route in a star shape around the circle. Use the data from road names in the routes to make a list of unique road names by geocoding the endpoints of divisions in the route called “steps” (Geocoding results in the third image). Make a list of intersections like the first idea. The intersections are the red markers in the second image. The problem with this approach is that the roads are not very diverse to begin with, and also have a large number of useless roads, since the via points in the generation tend to not be on non-side roads.
3. Variation on second idea, with making an “X” shape on the circumference of the circle. Attempted to make road options more relevant to a direct route.
4. Variation on second idea, with via points perpendicular to the line between origin on destination. Had similar results to third idea. (Image below with editing)

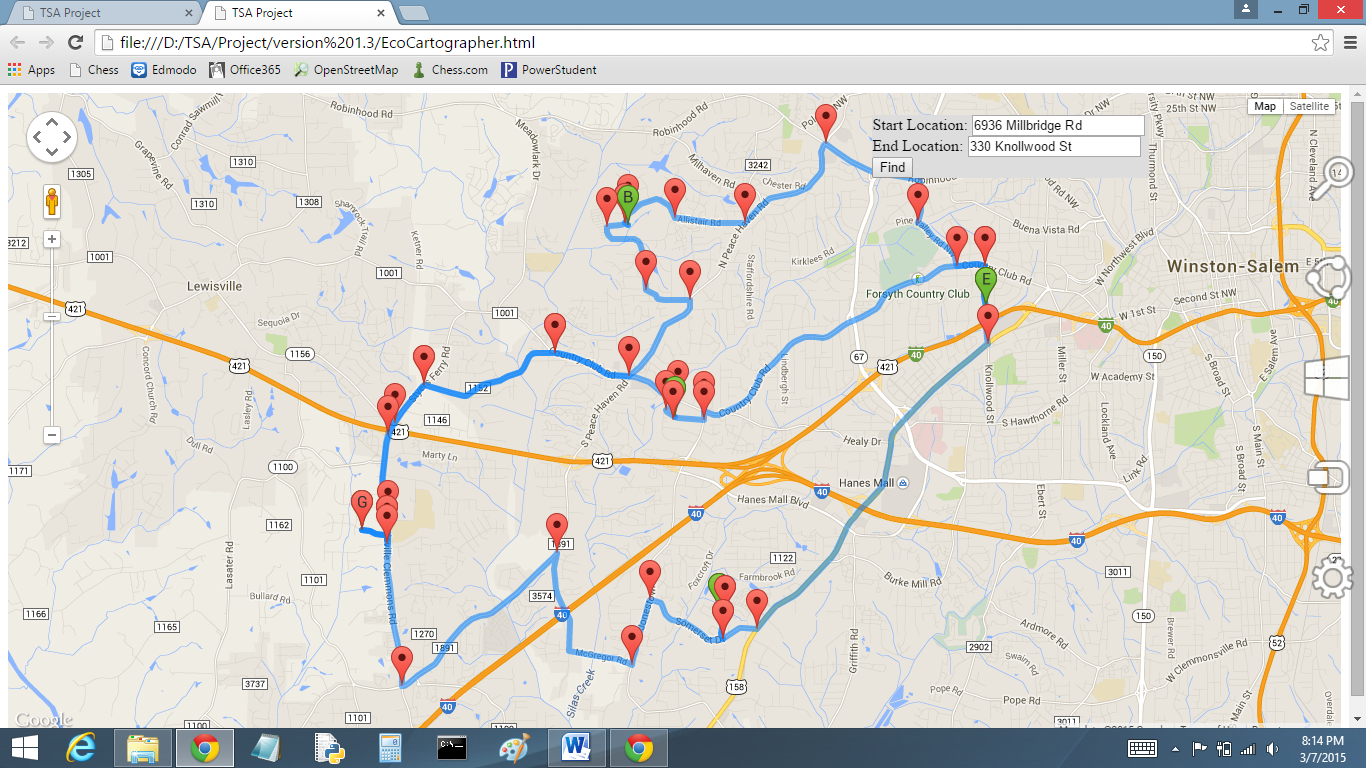
Best solution is probably a variant.

**Outputs:**

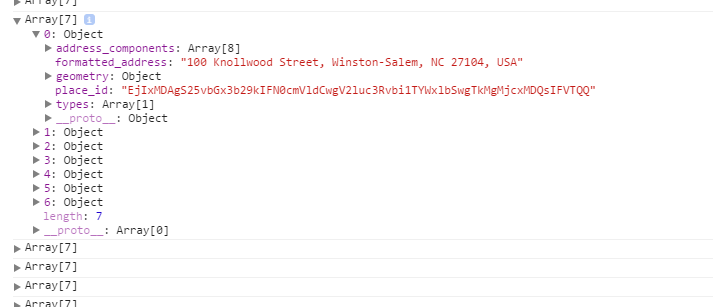
*Output 1.3.1:*



*Output 1.3.2:*



*Output 1.3.3:*



**New Code:**

function Network(sp,ep){

var dlat = sp.lat()-ep.lat(); //difference lat/lng

var dlng = sp.lng()-ep.lng();

var center = new google.maps.LatLng(sp.lat()-dlat/2, sp.lng()-dlng/2); // center between dest and orig

var cm = new google.maps.Marker({position:center, map:map}); //center marker

var hdist = Math.sqrt(dlat\*dlat + dlng\*dlng)/4; //half dist from orig to dest

//basic measures

Roads = [];

var tbgc= [];

var wpts = [{location:new google.maps.LatLng(center.lat()+dlng/4,center.lng()-dlat/4), stopover:true},

{location:ep, stopover:true},

{location:center, stopover:true},

{location:sp, stopover:true},

{location:new google.maps.LatLng(center.lat()-dlng/4,center.lng()+dlat/4), stopover:true}];

console.log(wpts);

request1 = {origin:sp, destination:ep, avoidHighways:true, waypoints:wpts, travelMode:google.maps.DirectionsTravelMode.DRIVING}

directionsService.route(request1, function(response, status){

directionsDisplay.setDirections(response);

console.log(response);

for(var i=0; i<response.routes[0].legs.length; i++){

for(var j=0; j<response.routes[0].legs[i].steps.length; j++){

var pc = response.routes[0].legs[i].steps[j].path[0];

tbgc.push(pc);

}

}

console.log(tbgc);

for(var i=0; i<tbgc.length; i++){

GC(i);

var M = new google.maps.Marker({position:tbgc[i], map:map});

}

});

function GC(i){

setTimeout(function(){

geocoder.geocode({location:tbgc[i]},function(results,status){

if(status!==google.maps.GeocoderStatus.OK){console.log(status)

}else{

console.log(results);

var R = results[0].address\_components[1].long\_name;

if(Roads.indexOf(R)===-1){

Roads.push(R);

}

}

})},1000\*i);

}

}

Version 1.4

**Summary:**

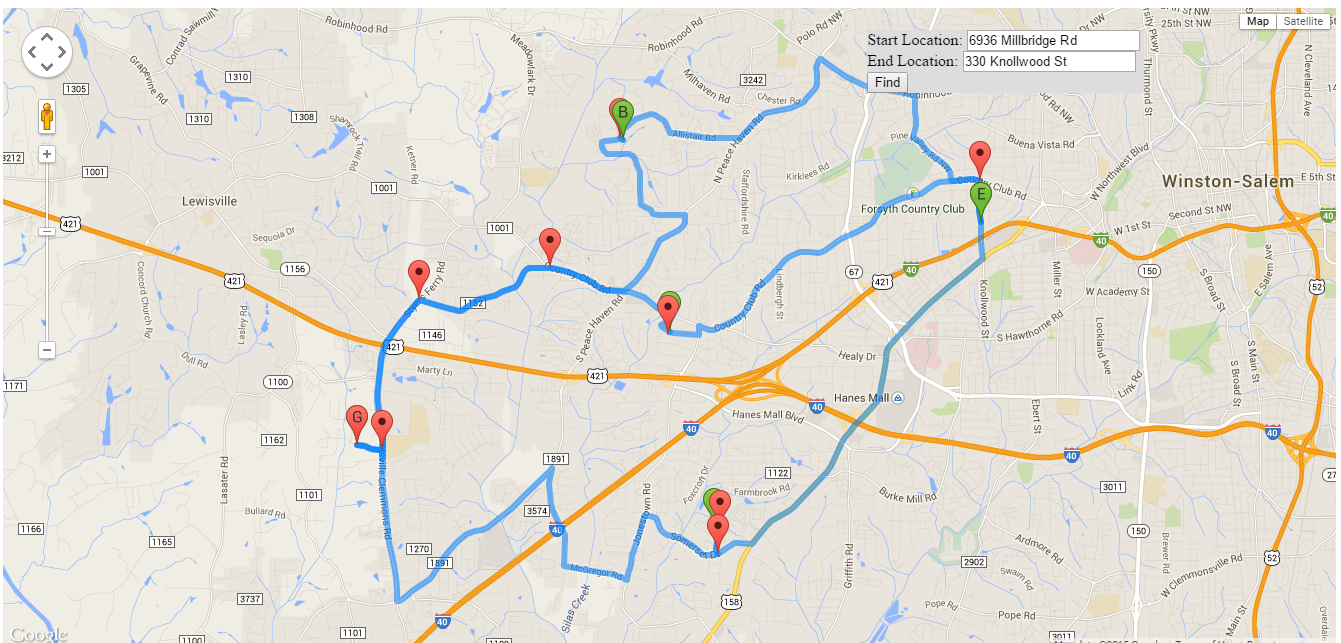
This version is a step towards building a network. It provided means of analyzing data relating to roads, and ultimately identifies places where multiple routes link up. A list of road names is collected in the Roads variable (Output 1.4.4), by parsing the directions given in the Google Maps Directions Service Output (Code 1.4.1). Intersections are determined by the frequency of the occurrence of a point (Visual on map: Output 1.4.1, Data: Output 1.4.2) (Code 1.4.2). Next step will be connecting these intersections together into arrays of points, which will be analyzed.

Connecting Points (in version 1.5)

1. Serialize array into points (var pts) (see Output 1.4.3).
2. Get list of intersections: points from pts which repeat more than two times.
3. Iterate through intersections, finding instances in pts and expanding outward in both directions, until a point occurring in intersections is found. A list of points is collected.
4. The list of points is assigned to an object represeting the intersection.
5. Use a tree-branch search method after.

**Outputs:**

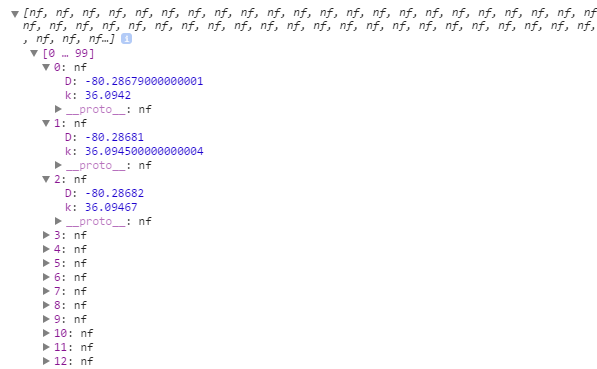
*Output 1.4.1:*



*Output 1.4.2:*



*Output 1.4.3:*

**

*Output 1.4.4:*

Roads: ["Knollwood St", "Buckingham Rd", "Country Club Rd", "Pine Valley Rd NW", "Wellington Rd", "Robinhood Rd", "N Peace Haven Rd", "Allistair Rd", "Allistair Rd", "Marble Arch Rd", "Corner Stone Ct", "Corner Stone Ct", "Marble Arch Rd", "Marble Arch Rd", "Cedar Trail", "Mountain View Rd NW", "N Peace Haven Rd", "Country Club Rd", "Phillips Bridge Rd", "Styers Ferry Rd", "Lewisville Clemmons Rd", "Millbridge Rd", "Millbridge Rd", "W Bridge Ct", "Lewisville Clemmons Rd", "August Dr", "Styers Ferry Rd", "Phillips Bridge Rd", "Country Club Rd", "Southwin Dr", "Randall St", "Randall St", "Jonestown Rd", "Jonestown Rd", "Country Club Rd", "Knollwood St", "Knollwood St", "Hanover Arms Ct", "US-158 W/S Stratford Rd", "Somerset Dr", "Ashford Dr SW", "Hollinswood Ave", "Hollinswood Ave", "Ashford Dr SW", "Ashford Dr SW", "Somerset Dr", "Jonestown Rd", "McGregor Rd", "S Peace Haven Rd", "Lewisville Clemmons Rd", "SW School Rd", "Millbridge Rd"]

**New Code:**

*Code 1.4.1:*

var not\_names = ["left","right","north","south","east","west","U-turn","northeast","southeast","northwest","southwest"];

for(var i=0; i<dirs.length; i++){ //dirs is a list of all the text directions given by the directions query

var y = dirs[i][0].split("<b>");

var z = [];

for(var j=0; j<y.length; j++){

if(y[j].indexOf("</b>")!==-1){

var a = y[j].split("</b>")[0];

if(not\_names.indexOf(a)===-1){Roads.push(a)} //removes text that would otherwise be a road name, but isn’t, i.e. “north”

}

}

}

*Code 1.4.2:*

var RoadKeys = {};

RoadCount = [];

for(var i=0; i<Roads.length; i++){

RoadKeys[Roads[i]] = 0;

}

for(var i=0; i<Roads.length; i++){

RoadKeys[Roads[i]] += 1;

}

for(var i in RoadKeys){

RoadCount.push([i,RoadKeys[i]]);

}

RoadCount.sort(function(a,b){return a[1]-b[1]});

console.log(dirs);

var pt\_keys = {};

pt\_count = [];

for(var i=0; i<pts.length; i++){

pt\_keys[pts[i]] = 0;

}

for(var i=0; i<pts.length; i++){

pt\_keys[pts[i]]++;

}

for(var i in pt\_keys){

pt\_count.push([i,pt\_keys[i]]);

}

pt\_count.sort(function(a,b){return a[1]-b[1]});

var x = pt\_count.map(function(a){return a[1]});

var pcc={};

for(var i=0; i<x.length; i++){

var num = x[i];

pcc[num] = pcc[num] ? pcc[num]+1 : 1;

}

console.log(pts);

console.log(pt\_count);

console.log(pcc);

for(var i=0; i<pt\_count.length; i++){

if(pt\_count[i][1]>3){

var p = pt\_count[i][0].slice(1,pt\_count[i][0].length-1);

p = p.split(",")

var m = new google.maps.Marker({position:new google.maps.LatLng(p[0].toString(), p[1].toString()), map:map});

}

}

Version 1.5

**Summary:**

This version generates a network of intersections which can be viewed by typing "intersections" into the console. Intersections are created by finding the indexes of the most frequent points in the serialized points list, and taking a slice out to the nearest other intersection in either direction. This makes it possible to make abstract connections from one intersection to another. An example intersection is shown in Output 1.5.1. The data will be later parsed for characteristics such as distance, elevation profile, and other things in version 1.6. Afterwards, a search algorithm can finally be applied. This is the only change from 1.4, and visually, it is the same.

**Outputs:**

*Output 1.5.1:*

**

**Code:**

intersections = {list:[], info:[]};

function new\_intersection(pt){

var p = pt[0].slice(1,pt[0].length-1);

p = p.split(",");

var LL = new google.maps.LatLng(p[0].toString(), p[1].toString());

var ind = pt[2];

ind.length = pt[1];

var I = {coords:LL, connections:[], indexes:ind};

intersections.info.push(I);

intersections.list.push(LL);

var m = new google.maps.Marker({position:LL, map:map, icon:'marker.png'});

}

//Generates list of intersections

for(var i=0; i<pt\_count.length; i++){

if(pt\_count[i][1]>2){

new\_intersection(pt\_count[i]);

}

}

var series = []; //Terminators for paths

for(var i=0; i<intersections.info.length; i++){

var I = intersections.info[i];

for(var j=0; j<I.indexes.length; j++){

series.push([I.indexes[j], I.coords]);

}

}

series.sort(function(a,b){return a[0]-b[0]});

var IL = intersections.list;

var II = intersections.info;

for(var i=0; i<series.length-1; i++){

II[IL.indexOf(series[i][1])].connections.push({point:series[i+1][1], path:pts.slice(series[i][0],series[i+1][0]+1)});

II[IL.indexOf(series[i+1][1])].connections.push({point:series[i][1], path:pts.slice(series[i][0],series[i+1][0]+1).reverse()});

}

Version 1.6

**Summary:**

12/25/2014

I calculated using basic trig and ratios that the distance between two lines of longitude given latitude can be represented by the equation: distance = diameter\_of\_earth \* cos(lat) / 360. The diameter of the earth is 40075 km at 0 deg lat, cos 0 = 1. The cos is proportionate to the radius and therefore the circumference of the earth.

12/26/2014

Integrated above formula into algorithm, and also calculated approximate elevations by finding elevations between intervals of each point. The results yielded: Algorithm: 160986, Google: 190305, which is about 0.4% difference. Considering the total is this accurate, I can assume that the general formula is correct. I do not understand what factors are influencing the google total.

This means that my algorithm is accurate enough to provide rough data. Next task will be to find data on OpenStreetMap for road speeds.

12/28/2014

Continued to experiment with Overpass API to extract OpenStreetMaps. Will need to use URLs to fill forms.

Formula for drag: *FD = cD 1/2 ρ v2 A   (2)*

*Engineer’s toolbox*

Query for OpenStreetMap:

var request = escape('(node["highway"="traffic\_signal"](36.4,-80,36.41,-79.99);way(36.4,-80,36.41,-79.99););(.\_;>;);out;');

var info = new XMLHttpRequest();

info.open(“GET”, “http://overpass-api.de/api/interpreter?data=”  
+info, true);

info.send();

var xml = info.responseXML;

//Sources from Stack Overflow and OpenStreetMap wiki; map data courtesy of OpenStreetMap.

I generated this code by searching the internet for examples on how pull xml data from OpenStreetMap onto javascript. I came across the Overpass API, which allowed me to do it. I initially did not know that the API was capable of doing what I wanted. After a couple hours of investigation, I discovered that it did exactly what I wanted; I just needed to find a way to get the information directly from the web into my script. Meanwhile, I was looking for ways to translate XML into javascript, and on Stack Overflow (where I go with my software questions as a default). The device which does this is XMLHttpRequest. I learned the usage and did some test with it. I discovered the url the submit button on the form for Overpass queries [here](http://overpass-api.de/query_form.html), and inside the top form I discovered where the data went after the form was submitted (relative url of action attribute of first form). I found the data name inside the form, and I assumed that it would be transferred as the same name to the server. I experimented with sending QL across, and it worked. It was immediately integrated into the code.

12/31/2014

Engine fuel efficiency is known as thermal efficiency, or the ratio of chemical energy in gasoline which is converted to mechanical energy in the engine. For most cars this is about 25%, meaning that 25% of the energy in the gasoline is converted into kinetic energy. <http://en.wikipedia.org/wiki/Thermal_efficiency>

Need to Know:

For each model:

* Fuel Consumption @ RPM
* Torque @ RPM
* Drag Coefficient
* Cross-Sectional Surface Area
* Number of wheels, width of wheels
* Mass

Constants:

* Density of Air
* Coefficients for Rolling Friction
* Energy Density of Gasoline
* Gravity (9.81 m/ss)

Equations:

* Drag
* Rolling Friction

The energy density of gasoline is 44.4 MJ/kg (Wikipedia). Density of air is 1.2754 kg/m3 (at sea level, 0degC). Force of Rolling Resistance is c\*m\*g, where c is coefficient, m is mass and g is gravity (9.81). Force of drag is ½\*c\*rho\*v2\*A, where c is coefficient (car “like Prius” is 0.26), rho is density of air (1.2754), v is velocity, and A is cross-sectional area.

Brake Specific Fuel Consumption (BSFC) is the ratio of work done (kWh) to the amount of fuel used (g).

By 1/15/15 I integrated Overpass data collection and geocoding. The geocoding is to match up roads in the overpass data to Google maps data. They are done by making a list of requests, and then having a single setInteral loop check if all the data has come in yet. When all the data is in, it exits to a joining function, which executes the compute() function if all the queries have come in.

These functions were initially unsuccessful because I did not take into account that the functions after several tenths of a second needed to access a value of a common iterator in the for loop, and therefore failed. An example: for(var i=0; i<5; i++){setTimeout(function(){console.log(i)},100\*i)}; The result was: (5) 5

Another related issue was that locally created setIntervals in loops were deleted, so they could not be terminated. for(var i=0; i<5; i++){var loop = setInterval(function(){if(<condition>){clearInterval(loop)};console.log(text)},100)}; The result of this was that the loops never exited. This was solved by having an array of requests which was built upon by a for loop, and then having a single setInterval loop check for completion, instead of doing everything in one loop on an individual basis.

There is an issue with street name syntax which makes matching up in some cases impossible. Example: North Peace Haven Road vs. Peace Haven Road North, Pine Valley Road North vs. Pine Valley Road, etc. Since roads with the same names with or without extra (“North”) have for the most part the same attributes, errors were thrown. This was solved by making three versions of names: (example) real: North Peace Haven Road, reversed: Peace Haven Road North, plain: Peace Haven Road. These three options covered all possible syntaxes in OpenStreetMap, and so far no errors have been caused by this. The result is shown in Output 1.6.3.

I started gathering information on how OpenStreetMap data correlates to speed limit, but I have very little information as far as that goes. There are very limited speed limit datum, and it tends to be for one or two roads out of a thousand. I wrote a short script to collect data from thousands of road samples in var over and sort it. Results were recorded in an Excel spread sheet.

In assigning stop lights to points, I discovered that the Google and OpenStreetMaps data was off by about 20/10000s of a degree, which is significant. I applied a filter to allow all points within that radius of a stop light to have a stop light on them. This is imperfect, but after testing it appears to be reasonably good. Fine-tuning will have to be done.

The solution I used to find the average road speed was this: sort street data into categories of counties with sub-categories of road types. The median of each sub-category is used as the default value. If a speed is explicitly defined, that is used, but if not, the default is used. Mean was deemed unreliable because upon testing, I discovered that residential category had an average of 56 mph, which is unheard of.

Upon further inspection, I realized that map data contained in over was incorrect, thus throwing off the data derived from it. It likely got scrambled in the parse function, perhaps with a mistake I made in variable scope. I emailed Sayari Ghosh about it.

I analyzed the raw XML for a specific example and compared it to the over variable data, and my conclusion was that the elements did not exactly overlap when data was overwritten, by the design of my algorithm. This was solved with two concepts. The first was that I made each road name in over an array which was a collection the original types of data. I also split over up into different sectors based on the Overpass API query bounds, in order to localize the speeds for sections of road somewhat. In calculating the speed limits for a road, I changed the method from the median of each occurrence of a road name to the mean, considering the number of nodes in a segment of road.

In poking around a bit in the data, I noticed that the problem aforementioned was also caused by the Overpass API (perhaps) assigning close roads to roads that have a “ref” tag (like US 421). This was fixed by assigning by default the ref name to the name of a road if the ref tag existed. After fixing this, a new problem was seen, with road names coming out like “I 40 Business;US 421”, which did not exist in any of the geocoding data. Two road names were added to over, one for each of the strings split by the semicolon.

There is an issue with this method. The data in the over variable for roads like US 421 are invalid. I think this has to do with the above technique. The data was in general weird. An example from the console: (output\_01.png). However, the speeds variable had the correct information. The problem was that the conditional statement in the for loop assigning data to over was incorrect, and over was being overwritten constantly.

There was the finding of false street names in the geocode function. I added a local function which parses the xml to find the precise name by searching instead of guessing (like the old one). The leadup to this can be found in getNameTest.js. I originally only had it set to find the “street\_address” attribute, but expanded it to “route” to include highways. I also added a few conditional statements in the local function combo to format the road names to that of Overpass API (i.e. Business 40 to I 40 Business).

The issues outlined in Outputs 1.6.5 and 1.6.6 are caused by the following phenomenon: Overpass Data had either ref or name. I conducted a test (in output\_04.png) that demonstrates the problem: unused variables have the value hang around even when they are redefined. This was fixed by resetting all the variables in the for loop at the end to undefined. I now realize how sparse the data is. I will have to add more for the general region, or expand the area of the Overpass query to get a better database for speeds.

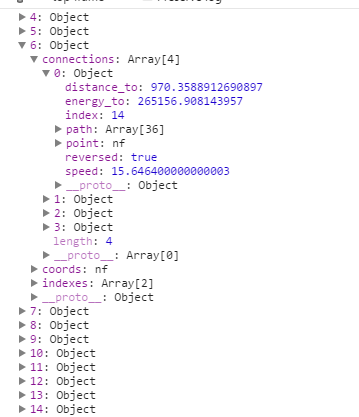
I redid the function for compiling averages to work, and also to reflect the number of nodes in a road. The final value is in m/s. I attempted to add it to each point in pts, but my initial attempt failed. It is commented off in the source code. I determined that there were three cases for assigning speed data: there is an explicitly defined speed limit, which is the first preference; there are other defined speed limits for the same road type; and finally there is a default value, which for the time being is 35 mph.

I added code which finds the amount of energy in Joules needed to move a vehicle along a path in the intersections object. I accounted for reversed roads, and added an exception if a point (rarely) does not have road data. As far as I know, this problem pertained to one point. I spent about an hour on 2/4/15 doing this. I still need to account for stoplights and signs, and acceleration.

**Outputs:**

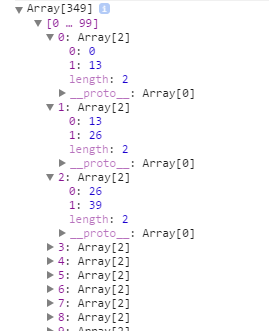
*Output 1.6.1:*

An example of an Intersection.



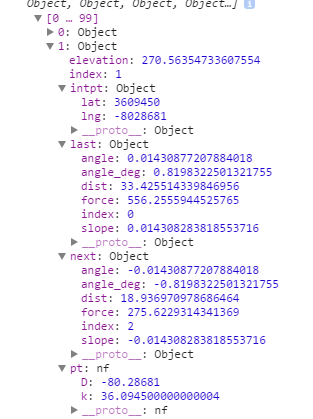
*Output 1.6.2:*

The elevation slices used for interpolation.



*Output 1.6.3:*

An example of a pts Object.





*Output 1.6.4:*

Overpass data in the over variable.

**

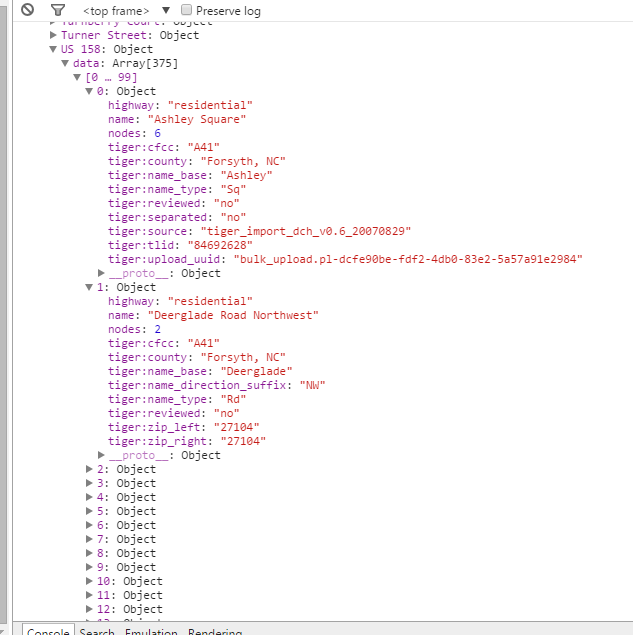
*Output 1.6.5:*

Some of the Overpass data that got mixed up.

**

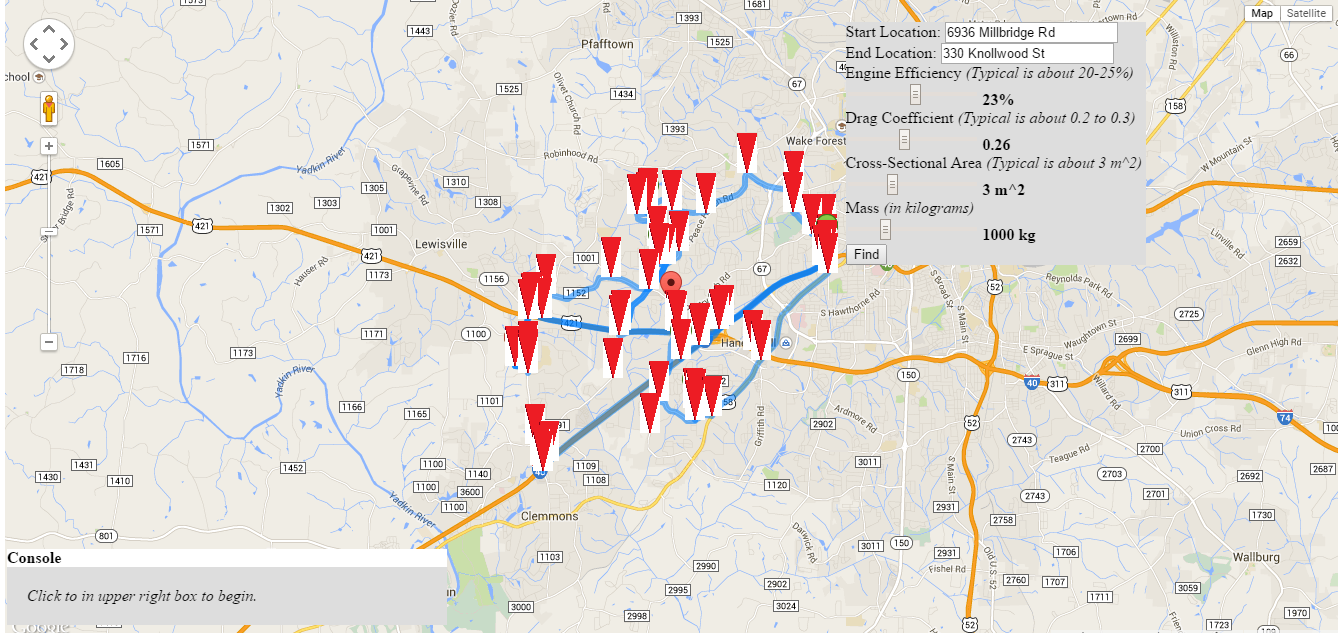
*Output 1.6.6:*

More Overpass data that got mixed up.

**

*Output 1.6.7:*

Visual display of intersections.

**

*Output 1.6.8:*

Data collected using the overcheck function (Code 1.6.7).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Search 1 | **oneway** | no | yes | undefined | sum | %no | %yes | **maxspeed** | 20 | 25 | 35 | 45 | 50 | 55 | 65 | undefined |
| **highway** |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| cycleway |  |  | 1 | 1 | 2 | 0 | 500 |  |  |  |  |  |  |  |  | 3 |
| footway |  |  | 6 | 8 | 14 | 0 | 429 |  |  |  | 1 |  |  |  |  | 14 |
| motorway |  |  | 35 |  | 35 | 0 | 1000 |  |  |  |  |  |  |  | 11 | 20 |
| motorway\_link |  | 2 | 10 |  | 12 | 167 | 833 |  |  |  |  |  |  |  |  | 11 |
| pedestrian |  |  |  | 1 | 1 | 0 | 0 |  |  |  |  |  |  |  |  | 1 |
| primary |  |  | 12 | 13 | 25 | 0 | 480 |  |  |  |  |  |  | 1 | 5 | 20 |
| primary\_link |  |  | 1 |  | 1 | 0 | 1000 |  |  |  |  |  |  |  |  | 1 |
| proposed |  |  | 1 |  | 1 | 0 | 1000 |  |  |  |  |  |  |  |  | 1 |
| residential |  | 8 | 27 | 5948 | 5983 | 1 | 5 |  | 2 | 1 | 1 | 5 |  |  | 6 | 5976 |
| secondary |  | 2 | 11 | 65 | 78 | 26 | 141 |  |  |  | 2 |  | 1 | 4 | 1 | 72 |
| service |  | 1 | 19 | 161 | 181 | 6 | 105 |  |  |  |  |  |  |  |  | 179 |
| steps |  |  | 1 | 1 | 2 | 0 | 500 |  |  |  |  |  |  |  |  | 2 |
| tertiary |  |  | 2 | 29 | 31 | 0 | 65 |  |  |  |  |  |  |  |  | 29 |
| track |  |  |  | 3 | 3 | 0 | 0 |  |  |  |  |  |  |  |  | 3 |
| trunk |  |  | 2 |  | 2 | 0 | 1000 |  |  |  |  |  |  |  | 2 |  |
| trunk\_link |  |  |  | 1 | 1 | 0 | 0 |  |  |  |  |  |  |  |  | 1 |
| unclassified |  |  | 1 | 1 | 2 | 0 | 500 |  |  |  |  |  |  |  |  | 2 |

**New Code:**

*Code 1.6.1:*

Elevation code, which uses interpolation to limit to one query. It also makes pts.next and pts.last data.

function elevation(){

//Elevation

var elev\_request = {locations:[]} //Elevation Request

var num\_elev\_pts = Math.min(350,pts.length);

for(var i=0; i<pts.length; i+=pts.length/num\_elev\_pts){

elev\_request.locations.push(pts[Math.floor(i)].pt);

}

console.log(elev\_request);

elevationService.getElevationForLocations(elev\_request, function(results, status){

if(status===google.maps.ElevationStatus.OK){

elevation\_data = results;

pt\_to\_elev\_ratio = (pts.length-1)/num\_elev\_pts

//fill in skeleton elevation data (returned in results)

for(var i=0; i<results.length; i++){

pts[Math.floor(i\*pt\_to\_elev\_ratio)].elevation = results[i].elevation

}

//set index variable in pts.next and pts.last

for(var i=1; i<pts.length-1; i++){

pts[i].index = i;

if(i!=0 && i!=pts.length-1){

pts[i].last.index = i-1;

pts[i].next.index = i+1;

}

}

pts[0].last.index = 0;

pts[0].next.index = 1;

pts[pts.length-1].last.index = pts.length-2;

pts[pts.length-1].next.index = pts.length-1;

//set distances between pts

rad = 180/Math.PI;

for(var i=1; i<pts.length-1; i++){

//coords (current, last, next for convenience with equations)

var cc = pts[i].pt;

var lc = pts[i-1].pt;

var nc = pts[i+1].pt;

pts[i].last.dist = Math.hypot( (cc.lat()-lc.lat())\*DIAMETER, (cc.lng()-lc.lng())\*Math.cos((cc.lat()+lc.lat())/2\*rad)\*DIAMETER);

pts[i].next.dist = Math.hypot( (cc.lat()-nc.lat())\*DIAMETER, (cc.lng()-nc.lng())\*Math.cos((cc.lat()+nc.lat())/2\*rad)\*DIAMETER);

pts\_dist += pts[i].last.dist;

}

var cc = pts[0].pt;

var nc = pts[1].pt;

pts[0].next.dist = Math.hypot( (cc.lat()-nc.lat())\*DIAMETER, (cc.lng()-nc.lng())\*Math.cos((cc.lat()+nc.lat())/2\*rad)\*DIAMETER);

cc = pts[pts.length-1].pt;

var lc = pts[pts.length-2].pt;

pts[pts.length-1].last.dist = Math.hypot( (cc.lat()-lc.lat())\*DIAMETER, (cc.lng()-lc.lng())\*Math.cos((cc.lat()+lc.lat())/2\*rad)\*DIAMETER);

pts\_dist += pts[pts.length-1].last.dist;

//set elevations between other points based on linear slope

var elev\_pts = [];

var elev\_slice\_pieces = [];

for(var i=0; i<pts.length; i++){

if(pts[i].elevation!==null){

elev\_pts.push(i);

}

}

for(var i=0; i<elev\_pts.length-1; i++){

elev\_slice\_pieces.push([elev\_pts[i],elev\_pts[i+1]]);

}

console.log(elev\_slice\_pieces);

for(var i=0; i<elev\_slice\_pieces.length; i++){

var start = pts[elev\_slice\_pieces[i][0]];

var end = pts[elev\_slice\_pieces[i][1]];

var slice = elev\_slice\_pieces[i];

var d\_elev = start.elevation-end.elevation; // going from start to end, /=pos & \=neg

var dist = 0;

for(var j=slice[0]+1; j<slice[1]; j++){

dist += pts[j].last.dist;

}

dist += end.last.dist;

var slope = d\_elev / dist; //rise over run

var angle = Math.asin(slope);

var angle\_deg = angle\*180/Math.PI;

var gone = 0; //counter for next loop

for(var j=slice[0]; j<slice[1]; j++){

gone += pts[j].last.dist;

pts[j].last.slope = -1\*slope;

pts[j].last.angle = -1\*angle;

pts[j].last.angle\_deg = -1\*angle\_deg;

pts[j].next.slope = slope;

pts[j].next.angle = angle;

pts[j].next.angle\_deg = angle\_deg;

pts[j].elevation = start.elevation + slope \* gone;

}

}

data\_complete++;

}else{

console.log(status);

};

});

}

*Code 1.6.2:*

Old Geocode function.

function geocodeOLD(){ //first attempt at geocoding. It was ineffective because it returned city names sometimes, because the name was taken blindly and not checked. The new function checks, and also creates formatted names with rearranged directional keywords.

console.log(pts);

function G(start,end,islast){

setTimeout(function(){

geocoder.geocode({'latLng':pts[parseInt((start+end)/2)].pt},function(results\_geo,status){

console.log(status);

var name = results\_geo[0].address\_components[1].long\_name;

for(var j=start; j<end+1; j++){

pts[j].road\_name = name;

}

});

},100\*i);

if(islast){

data\_complete++;

wait();

}

}

for(var i=0; i<ends.length-1; i++){

G(eind[i],eind[i+1],((i===eind.length-2)?true:false));

}

}

*Code 1.6.3:*

New geocode function, which collected information through XMLHttpRequest, and assigned it to the correct points.

function geocode(){

function combo(n){ //rearranges road names into several possible formats

var ILLEGALS = ['Northwest','Northeast','Southwest','Southeast','North','South','East','West'];

var output = {real:n, reversed:null, plain:null};

var name = n;

while(name.indexOf(".")!==-1){name=name.replace(".","")}; //Take out periods i.e. U.S. 421 --> US 421 for formatting

name = name.replace("Interstate","I");

if(name.indexOf("Business")!==-1){

name = name.replace("Business ","");

name = "I "+name+" Business";

}

for(var i=0; i<ILLEGALS.length; i++){

if(name.indexOf(ILLEGALS[i])===0){

name = name.replace(ILLEGALS[i]+' ','');

output.plain = name;

name += ' '+ILLEGALS[i];

output.reversed = name;

break;

}else if(name.indexOf(ILLEGALS[i])!==-1){

name = name.replace(' '+ILLEGALS[i],'');

output.plain = name;

name = ILLEGALS[i]+' '+name;

output.reversed = name;

break;

}else{

output.plain = name;

output.reversed = name;

}

}

return output;

}

/\*function getName(d){ //takes array of result XML tags

var address\_nodes;

var component\_node;

for(var i=0; i<d.length; i++){

if(d[i].getElementsByTagName("type")[0].value==="street\_address"){

address\_nodes = d[i].getElementsByTagName("address\_component");

break;

}

}

for(var i=0; address\_nodes.length; i++){

if(address\_nodes[i].getElementsByTagName("type")[0].value==="route"){

var component\_node = address\_nodes[i];

break;

}

}

return component\_node.getElementsByTagName("long\_name")[0].value;

}\*/

function getName(d){ //takes array of result XML tags as argument. Finds the proper name of a road in the function.

var a;

var s;

for(var i=0; i<d.length; i++){

var t = d[i].getElementsByTagName("type");

for(var j=0; j<t.length; j++){

if(t[j].innerHTML === "street\_address" || t[j].innerHTML === "route"){

a = d[i].getElementsByTagName("address\_component");

break;

}

}

}

if(a!==undefined){

for(var i=0; i<a.length; i++){

if(a[i].getElementsByTagName("type")[0].innerHTML === "route"){

s = a[i].getElementsByTagName("long\_name")[0].innerHTML;

break;

}

}

}else{

s = "undefined"; //needs to be string for later operations

}

return s;

}

geo = [];

var iter = 0; //counter for the timeout

for(var i=0; i<eind.length-1; i++){

setTimeout(function(){

var info = new XMLHttpRequest();

info.open("GET","https://maps.googleapis.com/maps/api/geocode/xml?latlng="+(((pts[eind[iter]+1].pt.toString()).replace(" ","")).replace("(","")).replace(")","")+"&key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE",true);

info.send();

geo.push({xml:info, done:false, range:{start:eind[iter], end:eind[iter+1]}});

console.log("GEOCODE");

if(iter===eind.length-2){

check();

}

iter++;

},200\*i);

}

function check(){ //checks for all queries being complete

console.log("reached check()");

var loop = setInterval(function(){

for(var i=0; i<geo.length; i++){

var num = 0;

if(geo[i].done===false){

if(geo[i].xml.responseXML!==null){

var name = getName(geo[i].xml.responseXML.children[0].getElementsByTagName("result"));

geo[i].done = true;

var list = combo(name);

for(var j=geo[i].range.start; j<geo[i].range.end; j++){

pts[j].road\_name = list;

}

}else{

num++;

}

}

}

if(num===0){

clearInterval(loop);

data\_complete++;

wait();

}

});

}

}

*Code 1.6.4:*

Retrieved road and traffic signal information from the OpenStreetMap database, and localized it.

function overpass(){

var ALLOWEDKEYS = ['name','highway','bicycle','foot','oneway','sidewalk','lanes','maxspeed','tiger:county'];

speeds = {}; /\*holds speed info, used to calculate averages.

data structure: {'tiger:county':{highway:{maxspeed\_1:<num of occurances>, maxspeed\_2:....}....}....}

\*/

//extract data from an XML group

function parse(d){

var output = {};

for(var j=0; j<d.length; j++){

var tags = d[j].getElementsByTagName("tag");

//Check for specific attributes

var name; var maxspeed; var county; var highway; var ref;

for(var k=0; k<tags.length; k++){

var key = tags[k].attributes.k.value;

if(key==="name"){

name = tags[k].attributes.v.value;

}else if(key==="maxspeed"){

maxspeed = tags[k].attributes.v.value;

}else if(key==="tiger:county"){

county = tags[k].attributes.v.value;

}else if(key==="highway"){

highway = tags[k].attributes.v.value;

}else if(key==="ref"){

ref = tags[k].attributes.v.value;

}

}

var roads = [undefined];

if(name!==undefined){

roads = name.split(";");

}else if(ref!==undefined){

roads = ref.split(";");

}

/\*//add attributes to output

for(var k=0; k<tags.length; k++){

var key = tags[k].attributes.k.value;

if(ALLOWEDKEYS.indexOf(key)!==-1){

var value = tags[k].attributes.v.value;

for(var road=0; road<roads.length; road++){

output[roads[road]][key] = value;

}

}

}\*/

var new\_data = {};

for(var t=0; t<tags.length; t++){

var key = tags[t].attributes.k.value;

new\_data[key] = tags[t].attributes.v.value;

}

var nodes = d[j].getElementsByTagName("nd").length;

new\_data.nodes = nodes;

//add name to output

for(var r=0; r<roads.length; r++){

if(!(roads[r] in output)){

output[roads[r]] = {data:[], speed:null};

}

output[roads[r]].data.push(new\_data);

}

//add info to speeds to later calculate averages

if(county && maxspeed && highway){

if(!(county in speeds)){

speeds[county] = {};

}

if(!(highway in speeds[county])){

speeds[county][highway] = {data:[], average:null};

}

for(var r=0; r<roads.length; r++){

speeds[county][highway].data.push({speed:parseInt(maxspeed)\*METERPS, nodes:nodes, name:roads[r], multiple\_names:roads, raw:new\_data});

}

}

//reset variables

name = undefined; ref = undefined; maxspeed = undefined; highway = undefined; county = undefined;

}

return output;

}

over = [];

requests = [];

console.log(bounds);

//Get Road Data

for(var i=bounds.south; i<bounds.north; i+=0.04){

for(var j=bounds.west; j<bounds.east; j+=0.04){

var info = new XMLHttpRequest();

info.open("GET","http://overpass-api.de/api/interpreter?data="+escape('way["highway"]('+i+','+j+','+(i+0.04)+','+(j+0.04)+');out;'),true);

try{info.send();}catch(err){alert("There was a problem with the Overpass API query. "+err+". Please try again")}

requests.push({bounds:{south:i, north:i+0.04, west:j, east:j+0.04}, xml:info, done:false});

}

}

//Get Traffic Signal Data

var signal\_request = new XMLHttpRequest();

signal\_request.open("GET","http://overpass-api.de/api/interpreter?data="+escape('node["highway"="traffic\_signals"]('+bounds.south+','+bounds.west+','+bounds.north+','+bounds.east+');out;'),true);

signal\_request.send();

requests.push({bounds:"signals", xml:signal\_request, done:false});

console.log(requests);

var loop = setInterval(function(){ //iterates through requests, checks for response

var num = 0; //if 0 after for loop, exits setInterval

for(var i=0; i<requests.length; i++){

if(requests[i].done===false){

if(requests[i].xml.responseXML!==null){

if(requests[i].bounds!=="signals"){

var streets = parse(requests[i].xml.responseXML.children[0].getElementsByTagName("way"));

over.push({bounds:requests[i].bounds, data:streets});

}else{

signals = requests[i].xml.responseXML.children[0].getElementsByTagName("node");

}

requests[i].done = true; //marks as 'processed'

}else{

num++;

}

}

}

if(num===0){

clearInterval(loop);

console.log(over);

data\_complete++;

}

},100);

}

*Code 1.6.5:*

Function which waited until geocoding, elevation, and Overpass queries were all done until moving on to the nex function.

function wait(){

if(data\_complete===3){

compute();

}else{

setTimeout(wait,100);

}

}

*Code 1.6.6:*

Function which contained the functions needed to build intersections and assign road data.

function compute(){

function add\_signals(){//Integrate OpenStreetMap Data into pts

//signals list

siglist = [];

for(var i=0; i<signals.length; i++){

siglist.push({lat:parseInt(signals[i].attributes.lat.value\*100000), lng:parseInt(signals[i].attributes.lon.value\*100000)});

}

//iterate through pts, add signals

for(var i=0; i<pts.length; i++){

//assign stop lights to pts

for(var j=0; j<siglist.length; j++){

if(Math.abs(siglist[j].lat-pts[i].intpt.lat)<20 && Math.abs(siglist[j].lng-pts[i].intpt.lng)<20){

pts[i].stop = "light";

}else if(eind.indexOf(i)!==-1){

pts[i].stop = "sign";

}

}

}

}

function add\_speeds(){

//calculate speed averages

function average(d,type){ //returns median speed limit

var list = [];

d.map(function(a){for(var i=0; i<a.nodes; i++){list.push(a.speed)}}); //make speed average based on number of nodes in road

list.sort();

console.log(list);

return list[parseInt(list.length/2)]; //median of data

}

for(var i in speeds){

for(var j in speeds[i]){

speeds[i][j].average = average(speeds[i][j].data,j);

}

speeds[i].default = (speeds[i].residential!==undefined)?speeds[i].residential.average:35\*METERPS;

}

//assign data to roadname

for(var i=0; i<over.length; i++){

var b = over[i].bounds;

var info = over[i].data;

for(var j=0; j<pts.length; j++){

var lat = pts[j].pt.lat();

var lng = pts[j].pt.lng();

if(b.north>lat && b.south<lat && b.west<lng && b.east>lng){

var rn = pts[j].road\_name;

if(rn.real in info){

pts[j].road\_data = info[rn.real];

}else if(rn.reversed in info){

pts[j].road\_data = info[rn.reversed];

}else if(rn.plain in info){

pts[j].road\_data = info[rn.plain];

}

//add speed limit

if(pts[j].road\_data!==null){

if(pts[j].road\_data.data.some(function(a){return (a.maxspeed)?true:false})){ //has expicit speed

var sum = 0; var count = 0;

pts[j].road\_data.data.map(function(a){if(a.maxspeed!==undefined){sum += parseInt(a.maxspeed)\*METERPS\*a.nodes; count += a.nodes}});

pts[j].road\_data.speed = sum/count;

}else if(pts[j].road\_data.data[0].highway in speeds[pts[j].road\_data.data[0]["tiger:county"]]){ //has data from other roads of it's type

pts[j].road\_data.speed = speeds[pts[j].road\_data.data[0]["tiger:county"]][pts[j].road\_data.data[0].highway].average;

}else{ //NOTHING

pts[j].road\_data.speed = speeds[pts[j].road\_data.data[0]["tiger:county"]].default;

}

}else{

pts[j].road\_data = {data:[], speed:35\*METERPS};

}

}

}

}

//calculate force to maintain velocity at a point

for(var i=0; i<pts.length-1; i++){

var Fpar = phys.g \* phys.mass \* Math.sin(pts[i].next.angle);

var Ffriction = phys.g \* phys.mass \* Math.cos(pts[i].next.angle) \* phys.rolling;

var Fdrag;

try{Fdrag = 0.5 \* phys.CD \* phys.air \* phys.area \* Math.pow((pts[i].road\_data.speed+pts[i+1].road\_data.speed)/2,2);}catch(err){console.log(i)};

pts[i].next.force = Fpar + Ffriction + Fdrag; //total force to maintain velocity in Newtons

pts[i+1].last.force = -1\*Fpar + Ffriction + Fdrag;

}

}

function intersection\_gen(){

var pt\_keys = {};

pt\_count = [];

pt\_consec = [];

for(var i=0; i<pts.length; i++){

pt\_keys[pts[i].pt] = [0,[]];

if(i<pts.length-1){

if(pts[i].pt===pts[i+1].pt){

pt\_consec.push(pts[i].pt);

}

}

}

for(var i=0; i<pts.length; i++){

pt\_keys[pts[i].pt][0]++;

pt\_keys[pts[i].pt][1].push(i);

}

for(var i in pt\_keys){

pt\_count.push([i,pt\_keys[i][0],pt\_keys[i][1]]);

}

pt\_count.sort(function(a,b){return a[1]-b[1]});

var x = pt\_count.map(function(a){return a[1]});

var pcc={};

for(var i=0; i<x.length; i++){

var num = x[i];

pcc[num] = pcc[num] ? pcc[num]+1 : 1;

}

console.log(pts);

console.log(pt\_count);

console.log(pcc);

//Relevant to v.1.5.

intersections = {list:[], info:[]};

function new\_intersection(pt){

var p = pt[0].slice(1,pt[0].length-1);

p = p.split(",");

var LL = new google.maps.LatLng(p[0].toString(), p[1].toString());

var ind = pt[2];

ind.length = pt[1];

var I = {coords:LL, connections:[], indexes:ind};

intersections.info.push(I);

intersections.list.push(LL);

var m = new google.maps.Marker({position:LL, map:map, icon:'marker.png'});

}

//Generates list of intersections

for(var i=0; i<pt\_count.length; i++){

if(ends.indexOf(pt\_count[i][0])!==-1){

new\_intersection(pt\_count[i]);

}

}

}

function intersection\_build(){

var series = []; //Terminators for paths

for(var i=0; i<intersections.info.length; i++){

var I = intersections.info[i];

for(var j=0; j<I.indexes.length; j++){

series.push([I.indexes[j], I.coords]);

}

}

series.sort(function(a,b){return a[0]-b[0]});

var ilist = intersections.list;

var iinfo = intersections.info;

for(var i=0; i<series.length-1; i++){

iinfo[ilist.indexOf(series[i][1])].connections.push({reversed:false, point:series[i+1][1], index:ilist.indexOf(series[i+1][1]), path:pts.slice(series[i][0],series[i+1][0]+1)});

iinfo[ilist.indexOf(series[i+1][1])].connections.push({reversed:true, point:series[i][1], index:ilist.indexOf(series[i][1]), path:pts.slice(series[i][0],series[i+1][0]+1).reverse()});

}

console.log(iinfo);

for(var i=0; i<iinfo.length; i++){

var paths = iinfo[i].connections;

var indexes = paths.map(function(a){return a.index});

for(var j=0; j<paths.length; j++){

var energy\_to = 0;

var speed = 0;

var dist = 0;

var speed\_dist = 0;

if(paths[j].reversed===false){

paths[j].path.slice(0,paths[j].path.length-2).map(function(a){

energy\_to += a.next.force \* a.next.dist;

dist += a.next.dist;

try{speed += a.road\_data.speed \* a.next.dist; speed\_dist += a.next.dist}catch(err){}/\*in case road\_data is undefined\*/});

}else{

paths[j].path.slice(1,paths[j].path.length-1).map(function(a){

energy\_to += a.last.force \* a.last.dist;

dist += a.last.dist;

try{speed += a.road\_data.speed \* a.last.dist; speed\_dist += a.last.dist}catch(err){}});

//energy to move along section of road in Joules (n\*m)

}

intersections.info[i].connections[j].energy\_to = energy\_to;

intersections.info[i].connections[j].speed = speed/dist;

intersections.info[i].connections[j].distance\_to = dist;

}

}

}

add\_signals();

add\_speeds();

intersection\_gen();

intersection\_build();

}

*Code 1.6.7:*

Test code which could be used to find trends in Overpass data. Only compatible with early Overpass code.

function overcheck(a,b){

var over\_keys = Object.keys(over);

var s = {};

for(var i=0; i<over\_keys.length; i++){

console.log("CHECK");

var value\_a\_key = over[over\_keys[i]][a];

var value\_b\_key = over[over\_keys[i]][b];

if(!(value\_a\_key in s)){

s[value\_a\_key] = {};

}

if(!(value\_b\_key in s[value\_a\_key])){

s[value\_a\_key][value\_b\_key] = 0;

}

s[value\_a\_key][value\_b\_key]++;

}

return s;

}

*Code 1.6.8:*

Improvement to the old geocoding system, and could accurately find the road name from returns of Google Geocoding.

function getName(d){ //takes array of result XML tags

console.log(d);

var a;

var s;

for(var i=0; i<d.length; i++){

console.log(d[i].getElementsByTagName("type"));

if(d[i].getElementsByTagName("type")[0].innerHTML === "street\_address"){

a = d[i].getElementsByTagName("address\_component");

console.log(a);

break;

}

}

for(var i=0; i<a.length; i++){

if(a[i].getElementsByTagName("type")[0].innerHTML === "route"){

s = a[i].getElementsByTagName("long\_name")[0].innerHTML;

console.log(s);

break;

}

}

return s;

}

var geo = new XMLHttpRequest();

geo.open("GET","https://maps.googleapis.com/maps/api/geocode/xml?latlng=36,-80&key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE",true);

geo.send();

geo.onload = function(){console.log(getName(geo.responseXML.children[0].getElementsByTagName("result")))};

*Code 1.6.9:*

XMLHttpRequest test HTML document. Interfaced with through the console.

<html>

<head>

<script>

var info;

var request;

function query(q){

request = escape(q);

info = new XMLHttpRequest();

info.open("GET", "http://overpass-api.de/api/interpreter?data="+request, true);

info.send();

var i=0;

}

</script>

</head>

</html>

*Code 1.6.10:*

New overpass function, which did not have the data carry-over problem of the last function.

var over; //{} for storing data by road name

var speeds; //see definition in function for info

var requests; //iterated through to check for completion of all requests; contains XMLHttpRequests

//these variables need to be global to be accessed by developer or other functions

function overpass(){

var ALLOWEDKEYS = ['name','highway','bicycle','foot','oneway','sidewalk','lanes','maxspeed','tiger:county'];

speeds = {}; /\*holds speed info, used to calculate averages.

data structure: {'tiger:county':{highway:{maxspeed\_1:<num of occurances>, maxspeed\_2:....}....}....}

\*/

//extract data from an XML group

function parse(d){

for(var j=0; j<d.length; j++){

var tags = d[j].getElementsByTagName("tag");

//Check for specific attributes

var name; var maxspeed; var county; var highway;

for(var k=0; k<tags.length; k++){

var key = tags[k].attributes.k.value;

if(key==="name"){

name = tags[k].attributes.v.value;

if(name==="Lewisville Clemmons Road"){

console.log(d[j]);

}

}else if(key==="maxspeed"){

maxspeed = tags[k].attributes.v.value;

}else if(key==="tiger:county"){

county = tags[k].attributes.v.value;

}else if(key==="highway"){

highway = tags[k].attributes.v.value;

}

}

//add name to over

if(!(name in over)){

over[name] = {speed:null};

}

//add attributes to over

for(var k=0; k<tags.length; k++){

var key = tags[k].attributes.k.value;

if(ALLOWEDKEYS.indexOf(key)!==-1){

var value = tags[k].attributes.v.value;

over[name][key] = value;

}

}

//add info to speeds to later calculate averages

if(county && maxspeed && highway){

if(!(county in speeds)){

speeds[county] = {};

}

if(!(highway in speeds[county])){

speeds[county][highway] = {data:[], average:null};

}

speeds[county][highway].data.push({speed:maxspeed,name:name});

}

}

}

over = {};

requests = [];

var bounds = {north:36.12, south:36.03, west:-80.4, east:-80.28} //rough bounds of Winston-Salem. Real bounds derived from Google Maps route

//Get Road Data

for(var i=bounds.south; i<bounds.north; i+=0.04){

for(var j=bounds.west; j<bounds.east; j+=0.04){

//not yet optimized for non-XMLHttpRequest browsers

var info = new XMLHttpRequest();

info.open("GET","http://overpass-api.de/api/interpreter?data="+escape('way["highway"]('+i+','+j+','+(i+0.04)+','+(j+0.04)+');out;'),true);

info.send();

requests.push([[i,j].toString(),info,false]); //array of requests

}

}

//Get Traffic Signal Data

var signal\_request = new XMLHttpRequest();

signal\_request.open("GET","http://overpass-api.de/api/interpreter?data="+escape('node["highway"="traffic\_signals"]('+bounds.south+','+bounds.west+','+bounds.north+','+bounds.east+');out;'),true);

signal\_request.send();

requests.push(["signals",signal\_request,false]);

console.log(requests);

var loop = setInterval(function(){ //iterates through requests, checks for responses

var num = 0; //if 0 after for loop, exits setInterval

for(var i=0; i<requests.length; i++){

if(requests[i][2]===false){

if(requests[i][1].responseXML!==null){

if(requests[i][0]!=="signals"){

parse(requests[i][1].responseXML.children[0].getElementsByTagName("way"));

}else{

signals = requests[i][1].responseXML.children[0].getElementsByTagName("node");

}

requests[i][2] = true; //marks as 'processed'

}else{

num++;

}

}

}

if(num===0){ //when complete

clearInterval(loop);

console.log(over);

}

},100);

}

Version 1.7

**Summary:**

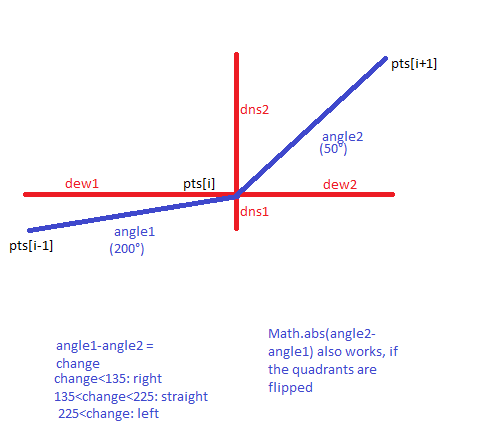
First thing done on Thursday, 2/5/15 at the TSA meeting was pruning intersection connections which would clog up the algorithm, which include duplicates and connections with .distance\_to = 0.

After that, I made the inputs for start and end locations variable.

I tested out the last improvement with using “1000 N Peace Haven Road, Winston-Salem” as the end location. I had built in an exception to deal with pts that had no road\_data, but I had no idea what was causing it. I rebuilt my algorithm to distribute road\_data more efficiently, but it still did not solve the problem. Then, after noticing large swaths of pts having no road\_data even though there was a valid road name, I checked the coordinates of one of the points, and it was not inside the bounds of any of the overpass data. This is displayed in Output 1.7.1. This was done the night of Saturday, 2/7/15. I realized that the bounds did not account for all the route queries, and I wrote a short function to find the bounds of all the points. This was done on Sunday 2/8/15. I also realized that in order to write a Djikstra algorithm I would need to know the index of the start and end intersection. The start is the first and last elements of pts, but I do not know the index of the end, and therefore cannot search for it.

Wednesday 2/11/15 and Thursday 2/12/15 I created a short function to make all the possible paths through the intersections, compiling a list of lists of indexes. The algorithms starts at a point, and gets a list of the indexes that connect. Then, it iterates through that list, and branches off to the indexes that it has not gone to before, and repeats this process until it either has already gone to all possible points or has reached the end. If it reaches the end, it returns a list of indexes and a cost. After the process is finished, it compares the energy cost and chooses the route with the lowest one. I needed to build another function to filter out mistakes from some problem I will identify later. An example of the list of intersection indexes is in Output 1.7.2. Output 1.7.3 is a list of all the branches created.

I added instructions on Monday and Tuesday, which at a change of road name, decides the angle of the turn and makes text for instructions. The values for each turn were designed with the assistance of the chart below. It also draws a Polyline on the map, which two points for start and end. Output 1.7.4 is an example of this. Output 1.7.5 shows the instructions for the optimal path, along with other data.

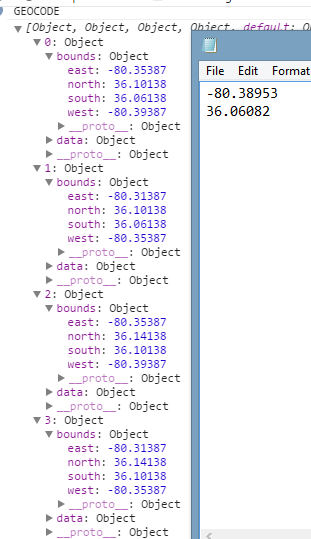


Finally, I made the input box turn transparent when the mouse is not over it, to improve visibility of the map.

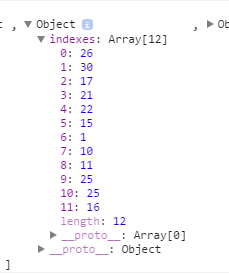
Icons from: <http://miftyisbored.com/a-complete-list-of-standard-google-maps-marker-icons/>

**Outputs:**

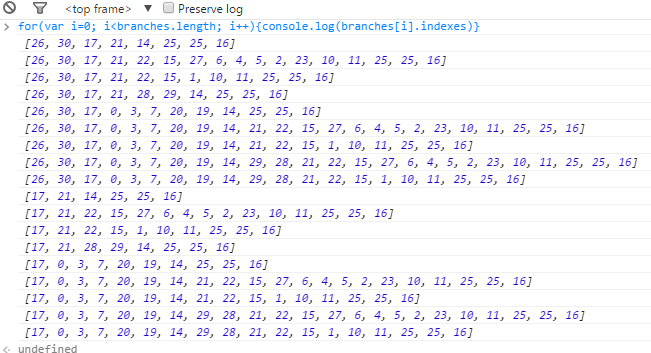
*Output 1.7.1:*

**

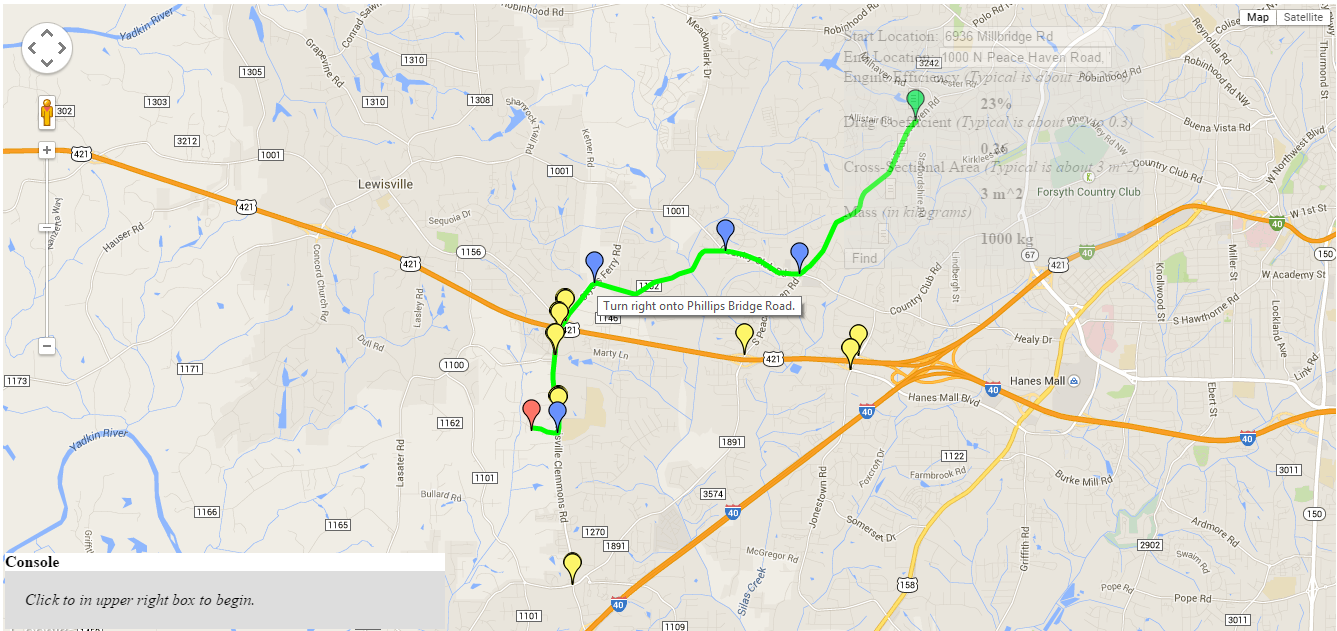
*Output 1.7.2:*

**

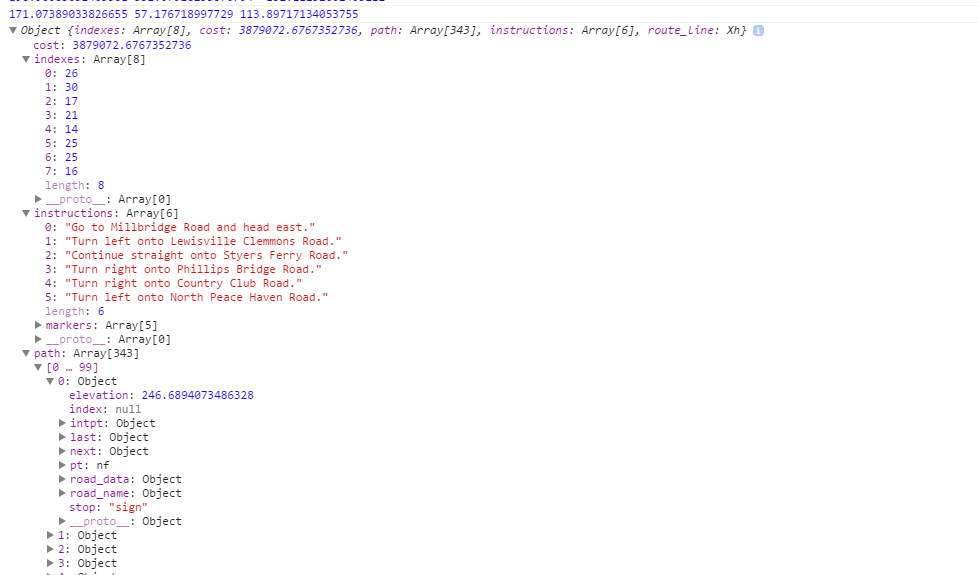
*Output 1.7.3:*

**

*Output 1.7.4:*

**

*Output 1.7.5:*

**

**New Code:**

*Code 1.7.1:*

New variable-start/end location feature. This function is triggered when the button is pushed.

function GetRouteFromInput(){

var start = document.getElementById('start\_input').value;

var end = document.getElementById('end\_input').value;

geocoder.geocode({address:start},function(results\_start,status\_start){

if(status\_start===google.maps.GeocoderStatus.OK){

var start\_coords = results\_start[0].geometry.location;

geocoder.geocode({address:end},function(results\_end,status\_end){

if(status\_end===google.maps.GeocoderStatus.OK){

var end\_coords = results\_end[0].geometry.location;

console.log(start\_coords,end\_coords);

var start\_marker = new google.maps.Marker({position:start\_coords, map:map, icon:'http://maps.google.com/mapfiles/ms/icons/red.png', title:start});

var end\_marker = new google.maps.Marker({position:end\_coords, map:map, icon:'http://maps.google.com/mapfiles/ms/icons/green.png', title:end});

Network(start\_coords,end\_coords);

}

});

}

});

}

*Code 1.7.2:*

Function which builds up a list of branches through intersections.

function Branch(index,lastIndexes,start,target){

var possible = intersections.info[index].connections.map(function(a){return a.index});

for(var B=0; B<possible.length; B++){

if(possible[B]===target){ //is at end

if(lastIndexes[0] === start){

branches.push({indexes:lastIndexes.concat(index,target), cost:GetCost(lastIndexes.concat(index,target))});

}

break;

}else if(lastIndexes.indexOf(possible[B])===-1){ //is not in current path

Branch(possible[B],lastIndexes.concat(possible[B]),start,target);

}

}

}

*Code 1.7.3:*

Gets the cost of a list of intersections.

function GetCost(path){

var total = 0;

for(var Pi=0; Pi<path.length-1; Pi++){

var conind = intersections.info[path[Pi]].connections.map(function(a){return a.index});

if(conind.indexOf(path[Pi+1])!==-1){

total += intersections.info[path[Pi]].connections[conind.indexOf(path[Pi+1])].energy\_to;

}

}

return total;

}

*Code 1.7.4:*

Modification to Code 1.7.3 for branches operating on itself.

branches.GetCost = function(){

for(var i=0; i<this.length; i++){

var total = 0;

for(var Pi=0; Pi<this[i].indexes.length-1; Pi++){

var conind = intersections.info[this[i].indexes[Pi]].connections.map(function(a){return a.index});

if(conind.indexOf(this[i].indexes[Pi+1])!==-1){

total += intersections.info[this[i].indexes[Pi]].connections[conind.indexOf(this[i].indexes[Pi+1])].energy\_to;

}

}

this[i].cost = total;

}

}

branches.Filter = function(start){

for(var Pi=0; Pi<this.length; Pi++){

if(this[i].indexes[0] !== start){

this[i] = null;

}

}

while(this.indexOf(null)!==-1){

this.splice(this.indexOf(null),1);

}

}

*Code 1.7.5:*

Instructions function.

var end\_index; var start\_index;

add\_signals();

add\_speeds();

intersection\_gen();

intersection\_build();

console.log(start\_index,end\_index);

Branch(start\_index,[start\_index],start\_index,end\_index);

var costs = branches.map(function(a){return a.cost});

optimal = branches[costs.indexOf(MIN(costs))], costs.indexOf(MIN(costs));

optimal.path = [];

for(var i=0; i<optimal.indexes.length-1; i++){

conind = intersections.info[optimal.indexes[i]].connections.map(function(a){return a.index});

if(conind.indexOf(optimal.indexes[i+1])!==-1){

var path = intersections.info[optimal.indexes[i]].connections[conind.indexOf(optimal.indexes[i+1])].path;

optimal.path = optimal.path.concat(path);

}

}

for(var i=0; i<optimal.path.length-1; i++){

if(optimal.path[i].pt.lat()===optimal.path[i+1].pt.lat() && optimal.path[i].pt.lng()===optimal.path[i+1].pt.lng()){ //check for duplicate consecutive points, remove the first occurance

optimal.path[i] = null;

}

}

while(optimal.path.indexOf(null)!==-1){

optimal.path.splice(optimal.path.indexOf(null),1);

}

optimal.instructions = [];

optimal.instructions.markers = [];

var first = "Go to "+optimal.path[0].road\_name.real+" and head ";

var pt1 = optimal.path[0].pt;

var pt2 = optimal.path[1].pt;

var lat = pt2.lat() - pt1.lat();

var lng = pt2.lng() - pt1.lng();

var direction;

if(lat>0){

if(lat>Math.abs(lng)){

direction = "north";

}else{

if(lng>0){

direction = "east";

}else{

direction = "west";

}

}

}else{

if(Math.abs(lat)>Math.abs(lng)){

direction = "south";

}else{

if(lng>0){

direction = "east";

}else{

direction = "west";

}

}

}

first += direction;

first += "."

optimal.instructions.push(first);

for(var i=1; i<optimal.path.length-1; i++){

if(optimal.path[i-1].road\_name.real!==optimal.path[i].road\_name.real){

var text = "";

var pt1 = optimal.path[i-1].pt;

var pt2 = optimal.path[i].pt;

var pt3 = optimal.path[i+1].pt;

var lat1 = pt1.lat()-pt2.lat();

var lng1 = pt1.lng()-pt2.lng();

var lat2 = pt3.lat()-pt2.lat();

var lng2 = pt3.lng()-pt2.lng();

var dns1 = lat1 \* DIAMETER;

var dns2 = lat2 \* DIAMETER;

var dew1 = lng1 \* Math.abs(Math.cos((pt1.lng()+pt2.lng())/2\*RAD)\*DIAMETER);

var dew2 = lng2 \* Math.abs(Math.cos((pt2.lng()+pt3.lng())/2\*RAD)\*DIAMETER);

var angle1 = RAD \* Math.atan(dns1/dew1);

var angle2 = RAD \* Math.atan(dns2/dew2);

if(dns1<0){

if(dew1<0){angle1+=180}else{angle1+=360}

}else{

if(dew1<0){angle1+=180}

}

if(dns2<0){

if(dew2<0){angle2+=180}else{angle2+=360}

}else{

if(dew2<0){angle2+=180}

}

var change = angle1-angle2;

console.log(angle1,angle2,change);

if(Math.abs(change)<225 && Math.abs(change)>135){

text += "Continue straight ";

}else if((change>0 && change<135) || (change>-360 && change<-225)){

text += "Turn left ";

}else{

text += "Turn right ";

}

text += "onto ";

text += optimal.path[i].road\_name.real;

text += ".";

optimal.instructions.push(text);

optimal.instructions.markers.push(new google.maps.Marker({position:pt2, map:map, icon:"http://maps.google.com/mapfiles/ms/icons/blue.png", title:text}));

}

}

optimal.route\_line = new google.maps.Polyline({

path:optimal.path.map(function(a){return a.pt}),

geodesic:true,

strokeColor:"#00ff00",

strokeOpacity:1.0,

strokeWeight:5});

optimal.route\_line.setMap(map);

console.log(optimal);

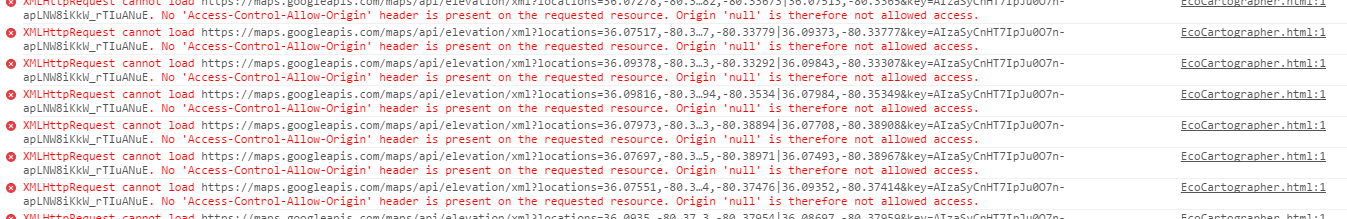
Version 2.1

**Summary:**

Made elevation requests for every point, instead of one query. Still need to calibrate time between requests and size of requests. Noticed from experimentation that the optimal is about 300 pts @ 2000 ms, with ceiling for pts @ 350. Originally used XMLHttpRequest, but "No-access-control-origin" error kept occuring, so stuck with the Maps API method. These errors are shown in Output 2.1.1. This was tried first, but is was too slow, so I decided to try XML.

**Outputs:**

*Output 2.1.1:*

**

**New Code:**

*Code 2.1.1:*

function elevation(){

function MakeRequest(url,delay,range){

var query\_timeout = setTimeout(function(){

var request = new XMLHttpRequest();

request.onload = function(){

var data = this.responseXML.children[0].getElementsByTagName("result");

for(var i=0; i<data.length; i++){

pts[i+range.min].elevation = parseFloat(data[i].getElementsByTagName("elevation")[0].value);

}

completion\_counter++;

}

request.open("GET",url,true);

request.send();

elev\_requests.push(request);

},delay);

}

//Elevation

var elev\_requests = [];

var completion\_counter = 0;

var expected\_requests = 0;

for(var i=0; i<Math.ceil(pts.length/100)\*100; i+=100){

var length = Math.min(100,pts.length-i);

var url = "https://maps.googleapis.com/maps/api/elevation/xml?locations=";

for(var j=0; j<length; j++){

url += parseInt(pts[i+j].pt.lat()\*100000)/100000 + "," + parseInt(pts[i+j].pt.lng()\*100000)/100000 + ((j!==length-1)?"|":"");

}

url += "&key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE";

expected\_requests++;

MakeRequest(url,i\*2,{min:i, max:i+length});

console.log(url);

}

var completion\_check = setInterval(function(){

if(completion\_counter === expected\_requests){

clearInterval(completion\_check);

SetData();

}

},100);

function SetData(){

for(var i=0; i<pts.length-1; i++){

//set index variable in pts.next

pts[i].index = i;

pts[i].next.index = i+1;

//set distance data

var cc = pts[i].pt; //current

var nc = pts[i+1].pt; //next

pts[i].next.dist = Math.hypot( (cc.lat()-nc.lat())\*DIAMETER, (cc.lng()-nc.lng())\*Math.cos((cc.lat()+nc.lat())/2\*RAD)\*DIAMETER);

//set slope data

var delev = pts[i+1].elevation - pts[i].elevation;

var slope = delev/pts[i].next.dist;

pts[i].next.slope = slope;

pts[i].next.angle = Math.atan(slope);

pts[i].next.angle\_deg = Math.atan(slope)\*RAD;

}

for(var i=1; i<pts.length; i++){

//set index variable in pts.last

pts[i].last.index = i-1;

//set distance data

var cc = pts[i].pt; //current

var lc = pts[i-1].pt; //last

pts[i].last.dist = Math.hypot( (cc.lat()-lc.lat())\*DIAMETER, (cc.lng()-lc.lng())\*Math.cos((cc.lat()+lc.lat())/2\*RAD)\*DIAMETER);

//set slope data

var delev = pts[i-1].elevation - pts[i].elevation;

var slope = delev/pts[i].last.dist;

pts[i].last.slope = slope;

pts[i].last.angle = Math.atan(slope);

pts[i].last.angle\_deg = Math.atan(slope)\*RAD;

}

pts[pts.length-1].index = pts.length-1;

data\_complete++;

}

}

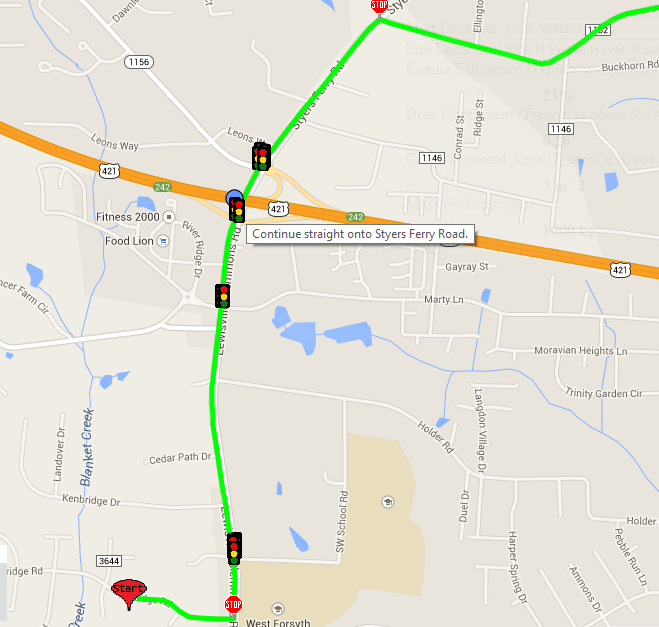
Version 2.2

**Summary:**

Stoplights could get doubled up, so I made stoplights assign properly to pts. Outputs 2.2.1 and 2.2.3 show the problem: there are clusters of stoplights. This involves checking through eind first, and assigning any intersections stoplights if they are close. If there are no intersections, all points on the map within 0.0002 degrees are assigned a stoplight. Then, consecutive stoplights are trimmed down to one. This information is more accurately carried over into the instructions display. Finished on 8:55 PM 2/24/2015. Made markers for the instructions on the output.

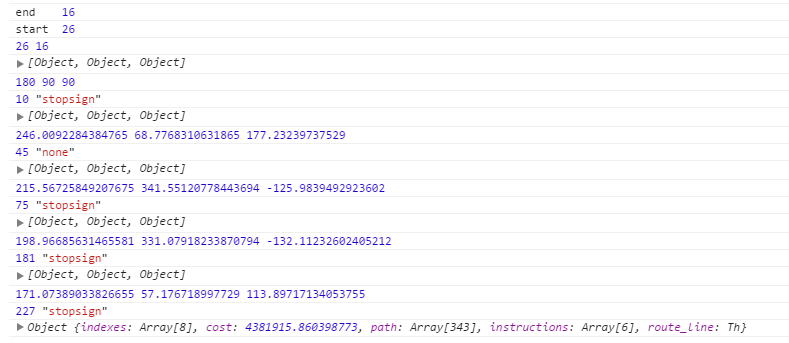
**Outputs:**

*Output 2.2.1:*



*Output 2.2.2:*

Problem with stoplights not being detected in the final output.



*Output 2.2.3:*



*Output 2.2.4:*

An example of the instructions output with the problem in 2.2.2 corrected.



**New Code:**

*Code 2.2.1:*

New code to add signasl by proximity to intersections. If it is not close, it adds to any points in the radius, and then trims off consecutive lights.

function add\_signals(){//Integrate OpenStreetMap Data into pts

//signals list

siglist = [];

for(var i=0; i<signals.length; i++){

siglist.push({lat:parseFloat(signals[i].attributes.lat.value), lng:parseFloat(signals[i].attributes.lon.value)});

}

//iterate through pts, add signals

/\*for(var i=0; i<pts.length; i++){

//assign stop lights to pts

var matched = false;

for(var j=0; j<siglist.length; j++){

if(Math.abs(siglist[j].lat-pts[i].pt.lat())<0.001 && Math.abs(siglist[j].lng-pts[i].pt.lng())<0.001){

pts[i].stop = "light";

pts[i].speed = 0;

try{

pts[i-1].stop = "light";

pts[i-1].speed = 0;

}catch(err){};

matched = true;

break;

}

}

if(!matched && eind.indexOf(pts[i].index)!==-1){

pts[i].stop = "sign";

pts[i].speed = 0;

try{

pts[i-1].stop = "sign";

pts[i].speed = 0;

}catch(err){};

//will need to replace with smart sign detection

}

}\*/

for(var i=0; i<eind.length; i++){

pts[eind[i]].stop = "sign"; //will change to smarter detection

try{pts[eind[i]-1].stop = "sign"}catch(err){};

}

for(var i=0; i<siglist.length; i++){

var matched = false;

for(var j=0; j<eind.length; j++){

if(Math.abs(siglist[i].lat-pts[eind[j]].pt.lat())<0.001 && Math.abs(siglist[i].lng-pts[eind[j]].pt.lng())<0.001){

pts[eind[j]].stop = "light";

pts[eind[j]].speed = 0;

try{

pts[eind[j]-1].stop = "light";

pts[eind[j]-1].speed = 0;

}catch(err){};

matched = true;

}

}

if(!matched){

for(var j=0; j<pts.length; j++){

if(Math.abs(siglist[i].lat-pts[j].pt.lat())<0.0002 && Math.abs(siglist[i].lng-pts[j].pt.lng())<0.0002){

pts[j].stop = "light";

pts[j].speed = 0;

}

}

}

}

for(var i=0; i<pts.length-1; i++){ //trim off extra lights in a series

if(pts[i].stop === "light" && pts[i+1].stop === "light" && !pts.isIntersection){

pts[i].stop = null;

}

}

}

Version 2.3

**Summary:**

I was working on the console display on 8:58 AM 2/25/2015 and I noticed in an XML document that Google Maps Route objects now have distance and time for steps, not just legs as it had originally been. This is shown in Output 2.3.1. This obviates the need for all of the Overpass API stuff that I have been doing, with calculating averages and stuff. I will strip all of the Overpass API code and replace it with the Google Maps Results. I will still need stoplight data. This will also remove all need for Geocoding, since it was used to correlate Overpass data to Google Maps Data. I will still perform a few queries to get road names after the most efficient route has been calculated.

I found out on 7:14 PM 2/26/2015 after reconfiguring the program to work with the new data that the speeds are inaccurate from what was originally had, and since almost all of them are below the actual limit, that it had to do with acceleration and deceleration from traffic signals. I tried to come up with a solution by comparing distances and times and such to try to find a correlation, but I have no certainty as to how to adjust to this because I have very little other data to work with. I asked a question about it on StackOverflow. For now I will assume that the speeds are correct.

Now I need to integrate acceleration and deceleration into the code.

I also noticed with the Elevation API that there is always the 'No Access-Control-Allow-Origin Header' when an XMLHttpRequest is made, but never when it is for Geocoding. I wonder if this is a bug, or built in. Until I can use XMLHttpRequest, I will not be able to speed up the elevation process.

I also looked for fuel efficiency data on http://www.fueleconomy.gov/feg/which\_tested.shtml. Later the same night, I reworked the intersections generation to search for frequencies in the path that are distinguished from other ones, meaning that they jump out from the background. This will be more reliable and less convoluted than the old method, which I barely understand anymore.

The next day, I wrote an alternate function for XMLHttpRequest in elevation, but I still got the same error.

I began work on building in acceleration and deceleration. I began by making a function to generate a path for each candidate solution, which are chosen by being close to the optimal in cost. Each candidate has a number of speed changes made to it based on the speed transition or presence of stoplight or stopsign, and the direction of turn. To get the turn, I made a global function which returns the angle from 3 points (I already had the code written from the instructions generator). I took a picture of the whiteboard I used to plan all of this out. I updated the next/last attribute of pts in a connection in an intersection that is reversed. I also made the tally in connections.force\_to not take into account negative forces (since it would cause an increase in velocity), and replace them with zero. The function I create the more accurate cost tallies up the original force minus the drag, which is recalculated. New drag is put up in its place. For now, the excess force generated by deceleration is simply subtracted, because for now it is too complicated to factor in all of the continuities. This will likely happen later. It took several tries to debug it.

I needed a solution to display the data, and to do this I initially experimented with generating an HTML page in the DOM and also uploading a template through and iframe and editting that with DOM. The latter failed because of cross-domain issues from iframe.contentDocument. The solution that I will use is to make a template page, and open a new tab for the user with this page, and have the parameters sent via URL. They will be stringified by JSON and escaped, then unescaped and parsed by JSON to have mostly intact data. I may use a static Google Map to display the route info. URL to figure out display of instructions: http://stackoverflow.com/questions/14070105/pre-fill-form-field-via-url-in-html. 4:05 PM 2/28/2015

I continued through the next week to generate a url to pass to a separate HTML page. The HTML page loads a google map, and has a div where the total distance can go, as well as a div that can have a list appended to it. I decided that the list should have a turn icon (which were not made at this point), and the icon for the thing at the turn (like a stoplight). It is minimally formatted, and I emailed Akshat to improve the formatting.

On Thursday 3/5/15, I sent emails to Collin and Akshat as a substitute for the meeting to coordinate work.

I had I final idea on 5:09 PM 3/7/2015 to minimize the amount of elevation data collected. It is to make a list of unique points and reference the index of the occurances of the same point in the pts array. It should (based on previous estimations) decrease the number of queries by half. I needed to adjust the elevation function so that it compiled requests by the point counts. In addition to making references to the points themselves, I also made a variable which is a list of the references. The function cycles through those, and sends the point as well as the references to the Request function, which upon receiving the information, assigns all the points in the indexes index that elevation. In the Millbridge to N Peace Haven example, the number of queries decreased from 11 to 4. It worked on the first attempt, after code debug.

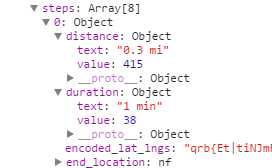
Finally, I replaced XMLHttpRequest with Ajax in jQuery. I was researching ways to get elevation data from Elevation API and I discovered that it might be possible using jQuery/Ajax. I experimented with JSONP, but it failed. I looked at jQuery documentation, and discovered that Ajax is completely cross-browser compatible. I developed queries by testing them in the console, and

I cleaned up the code in script, placing manipulated variables in the Network function, but leaving constants and the map variable as global. Extraneous functions were deleted.

Finished 7:02 PM 3/10/2015

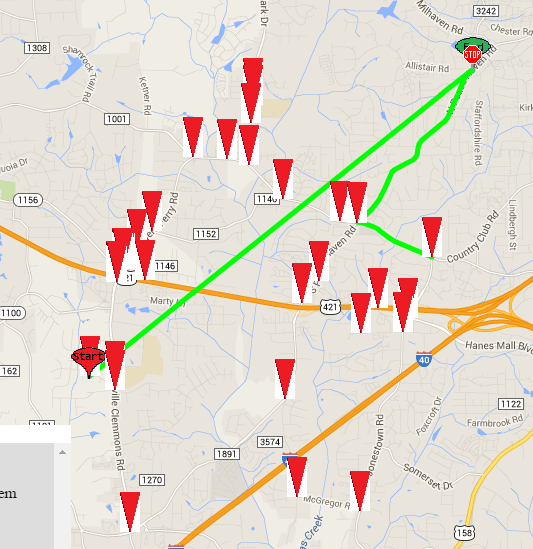
**Outputs:**

*Output 2.3.1:*

**

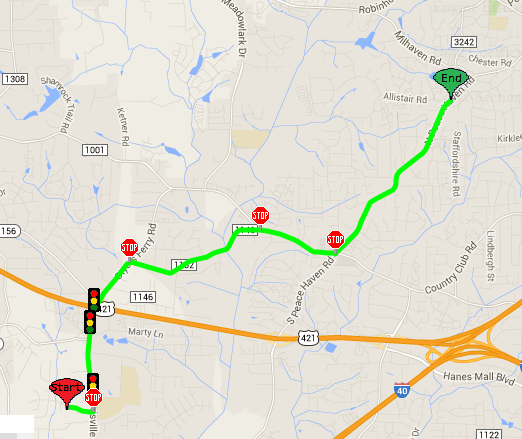
*Output 2.3.2:*

Error with the path. Later discovered that the queries for getting the route got reversed, so the route line was distorted.



*Output 2.3.3:*

An example of an output with no bias towards stop sign direction.



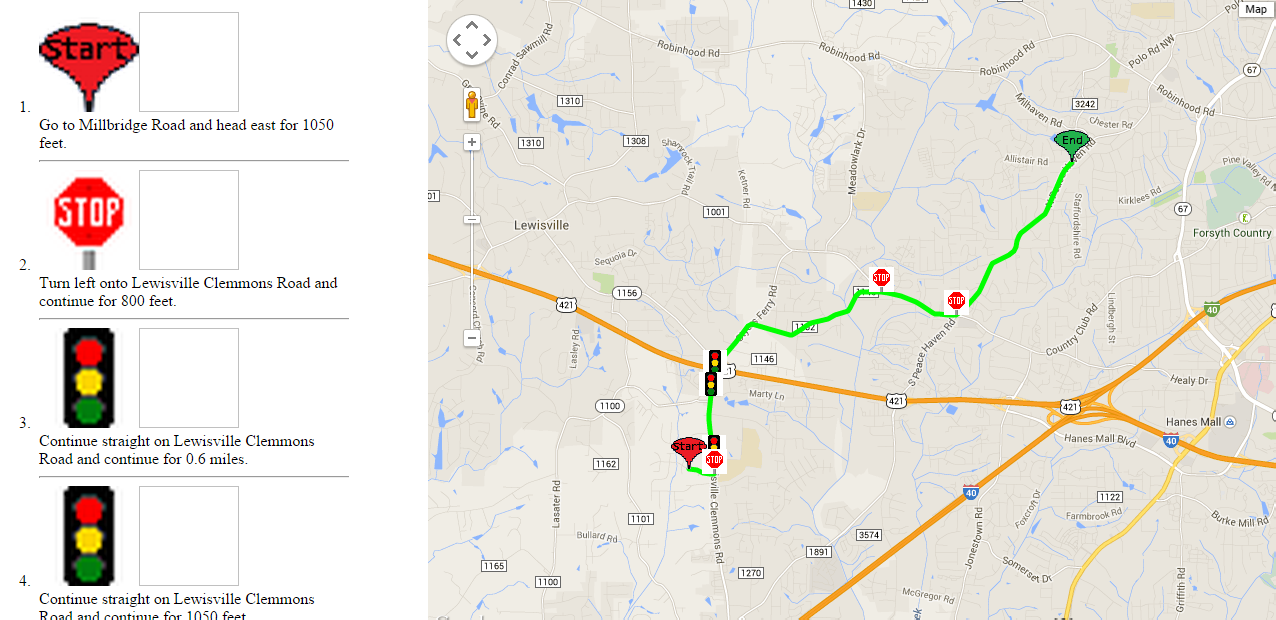
*Output 2.3.4:*

Test in the console with Ajax. This was the successful one; the others got various error messages.



*Output 2.3.5:*

An output in the instructions page.



**New Code:**

*Code 1.3.1:*

New code to get branches and find quick costs and detailed costs.

function Branch(index,lastIndexes,start,target){

var possible = intersections.info[index].connections.map(function(a){return a.index});

for(var B=0; B<possible.length; B++){

if(possible[B]===target){ //is at end

if(lastIndexes[0] === start){

branches.push({indexes:lastIndexes.concat(target), quick\_cost:GetQuickCost(lastIndexes.concat(target))});

}

break;

}else if(lastIndexes.indexOf(possible[B])===-1){ //is not in current path

Branch(possible[B],lastIndexes.concat(possible[B]),start,target);

}

}

}

function GetQuickCost(path){

var total = 0;

for(var Pi=0; Pi<path.length-1; Pi++){

var conind = intersections.info[path[Pi]].connections.map(function(a){return a.index});

if(conind.indexOf(path[Pi+1])!==-1){

total += intersections.info[path[Pi]].connections[conind.indexOf(path[Pi+1])].energy\_to;

}

}

return total;

}

function GetPath(indexes){

output = [];

for(var i=0; i<indexes.length-1; i++){

var conind = intersections.info[indexes[i]].connections.map(function(a){return a.index});

if(conind.indexOf(indexes[i+1])!==-1){

var path = intersections.info[indexes[i]].connections[conind.indexOf(indexes[i+1])].path;

output = output.concat(path);

}

}

//check for duplicate consecutive points, remove the first occurance

for(var i=0; i<output.length-1; i++){

if(output[i].pt.lat()===output[i+1].pt.lat() && output[i].pt.lng()===output[i+1].pt.lng()){

output[i] = null;

}

}

while(output.indexOf(null)!==-1){

output.splice(output.indexOf(null),1);

}

return output;

}

function GetDetailedCost(path){

function VelocityAtDistanceAccel(dist,start\_vel){

var start\_dist = Math.pow(start\_vel,2)/(ACCELRATE\*2);

return Math.sqrt(2 \* ACCELRATE \* (dist+start\_dist));

}

function VelocityAtDistanceDecel(dist,start\_vel){

var dist\_to\_decel = Math.pow(start\_vel,2)/(2\*ACCELRATE);

var accel\_dist = dist\_to\_decel - dist;

return VelocityAtDistanceAccel(accel\_dist,0);

}

var energy = 0; //output

//assign new speeds based on turns and speed changes

for(var i=1; i<path.length-1; i++){

if(path[i].stop === "light"){

path[i].new\_speed = 0;

}else if(path[i].stop === "sign"){ //at an intersection

var change = path[i].speed - path[i-1].speed;

var direction = ANGLE(path[i-1].pt,path[i].pt,path[i+1].pt);

if((direction>=225 && direction<360) || (direction>=-135 && direction<0)){ //is right turn

if(change<=0){

path[i].new\_speed = 15\*METERPS;

}else{

path[i].new\_speed = 0;

}

}else if((direction>135 && direction<225) || (direction<-135 && direction>-225)){

path[i].new\_speed = path[i].speed;

}else{

if(change===0){

path[i].new\_speed = 15\*METERPS;

}else{

path[i].new\_speed = 0;

}

}

}else{

path[i].new\_speed = path[i].speed;

}

}

path[0].new\_speed = 0;

path[path.length-1].new\_speed = 0;

//recalculate force to next

for(var i=0; i<path.length-1; i++){

energy += ((path[i].next.force===NaN || path[i].next.force<0)?0:path[i].next.force \* path[i].next.dist);

if(path[i].new\_speed !== path[i+1].new\_speed){

var speed\_change = path[i+1].new\_speed - path[i].new\_speed;

var dist\_to\_accel = Math.pow(path[i+1].speed,2)/(2\*ACCELRATE) - Math.pow(path[i].speed,2)/(2\*ACCELRATE);

energy += dist\_to\_accel \* phys.mass \* ACCELRATE; //energy to accelerate vehicle (m^2\*kg/s^2)

energy -= 0.5 \* phys.CD \* phys.air \* phys.area \* Math.pow(path[i].speed,2) \* dist\_to\_accel; //take away old drag

if(dist\_to\_accel>0){ //there is acceleration

for(var j=0; j<dist\_to\_accel; j++){

energy += 0.5 \* phys.CD \* phys.air \* phys.area \* Math.pow(VelocityAtDistanceAccel(j,path[j].new\_speed),2); //force per meter = energy

}

}else{

for(var j=0; j<dist\_to\_accel; j++){

energy += 0.5 \* phys.CD \* phys.air \* phys.area \* Math.pow(VelocityAtDistanceDecel(j,path[j].new\_speed),2);

}

}

}

}

return energy;

}

function compute(){

//calculate force for points to go to next/last

for(var i=0; i<pts.length-1; i++){

pts[i].next.force = Force(pts[i].next,pts[i].speed);

}

for(var i=1; i<pts.length; i++){

pts[i].last.force = Force(pts[i].last,pts[i].speed);

}

add\_signals();

intersection\_gen();

intersection\_build();

console.log(start\_index,end\_index);

Branch(start\_index,[start\_index],start\_index,end\_index);

var quick\_costs = branches.map(function(a){return a.quick\_cost});

optimal\_quick\_cost = branches[quick\_costs.indexOf(MIN(quick\_costs))].quick\_cost;

candidates = branches.map(function(a){return ((a.quick\_cost<=optimal\_quick\_cost\*1.3)?a:null);});

while(candidates.indexOf(null)!==-1){

candidates.splice(candidates.indexOf(null),1);

}

for(var i=0; i<candidates.length; i++){

candidates[i].path = GetPath(candidates[i].indexes);

candidates[i].cost = GetDetailedCost(candidates[i].path);

}

var candidates\_costs = candidates.map(function(a){return a.cost});

optimal = candidates[candidates\_costs.indexOf(MIN(candidates\_costs))];

geocode();

}

*Code 2.3.2:*

Instructions page JavaScript and HTML.

<html>

<head>

<style>

.h1 {font-size:40px; text-align:center;}

#total\_distance {font-size:20px;}

#list\_wrapper {width:350px; float:left;}

#map-canvas {float:right; }

</style>

<script type="text/javascript" src="http://maps.googleapis.com/maps/api/js?key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE"></script>

<script>

var map;

window.onload = function(){

var hash = window.location.hash.substr(1).split('&');

console.log(hash);

for(var i=0; i<hash.length; i++){

var kv = hash[i].split("=");

var key = kv[0];

var value = kv[1];

if(key==="summary"){

var data = JSON.parse(unescape(value));

document.getElementById("total\_distance\_spot").innerText = data.dist;

var map\_canvas = document.getElementById('map-canvas');

map\_canvas.style.width = window.innerWidth - 450;

map\_canvas.style.height = window.innerHeight;

map = new google.maps.Map(map\_canvas, {center:{lat:parseFloat(data.map.center.lat), lng:parseFloat(data.map.center.lng)}, zoom:data.map.zoom});

var coords = data.path\_polyline.split("|");

var path = coords.map(function(a){var parts = a.split(","); return new google.maps.LatLng(parseFloat(parts[0]),parseFloat(parts[1]))});

var polyline = new google.maps.Polyline({

path:path,

geodesic:true,

strokeColor:"#00ff00",

strokeOpacity:1.0,

strokeWeight:5});

polyline.setMap(map);

var end\_marker = new google.maps.Marker({position:path[path.length-1], icon:"\_\_end\_marker\_\_.png", title:"Destination", map:map});

}else if(key==="instructions"){

var data = JSON.parse(unescape(value));

for(var j=0; j<data.length; j++){

NewBlock(data[j]);

var marker = new google.maps.Marker({position:new google.maps.LatLng(data[j].pt.lat,data[j].pt.lng), icon:data[j].icon, title:data[j].text, map:map});

console.log(marker);

}

}

}

}

function NewBlock(a){

var li = document.createElement("li");

var div = document.createElement("div");

var icon = document.createElement("img");

icon.src = a.icon;

icon.width = 100;

icon.height = 100;

var turn = document.createElement("img");

turn.src = "";

turn.width = 100;

turn.height = 100;

var text\_container = document.createElement("div");

var text\_span = document.createElement("span");

var text = document.createTextNode(a.text);

text\_span.appendChild(text);

text\_container.appendChild(text\_span);

div.appendChild(icon);

div.appendChild(turn);

div.appendChild(text\_container);

div.appendChild(document.createElement("hr"));

li.appendChild(div);

document.getElementById("list").appendChild(li);

}

</script>

</head>

<body>

<div id="header">

<b><h1>Instructions</h1></b>

<div id="total\_distance">Distance: <i><span id="total\_distance\_spot"></span></i></div>

</div>

<div id="list\_wrapper">

<ol id="list">

</ol>

</div>

<div id="map-canvas">

</div>

</body>

</html>

*Code 2.3.3:*

Builds up list of indexes of points that share a common coordinate.

function PointReferences(){

var points = {};

for(var i=0; i<pts.length; i++){

if(!(pts[i].string in points)){

points[pts[i].string] = [];

}

points[pts[i].string].push(i);

}

for(var i=0; i<pts.length; i++){

pts[i].isElevationQuery = (points[pts[i].string][0] === i)?true:false;

pts[i].references = points[pts[i].string];

}

output = [];

for(var i in points){

output.push({indexes:points[i], pt:pts[points[i][0]].pt});

}

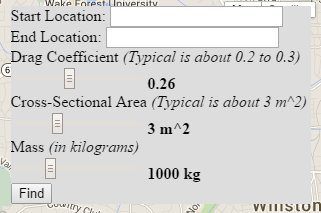
return output;

}

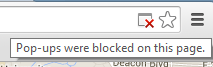
## End-User Documentation

Google Chrome recommended for usage with EcoCartographer. Firefox is functional but not ideal. Internet Explorer and Safari do not work at all.

**Operation Instructions**

1. Data Entry: Locations
   1. Enter your starting location in the top right corner box under starting location, and do the same for end location.
   2. These should be as specific as possible, unique addresses, and also not more than about 10 miles apart (otherwise the algorithm will take too long to complete). An example of a good search would be: Start Location: 567 Main Street, Winston-Salem, NC or End Location: Lowes Foods, Pfafftown.
2. Data Entry: Mass
   1. Look up the Mass of your model of vehicle from the manufacturer’s specifications. Enter that value into the appropriate slider in the input box.
   2. In the future, there will be a drop-down menu from which you can chose a default type of vehicle with approximate values for all three types of physical data.
   3. A mass is in kilograms, which is the scientific standard. One can convert pounds into kilograms by dividing the number of pounds by 2.2. Also in the future, there will be scientific and standard measurement options.
3. Data Entry: Frontal Area and Drag Coefficient
   1. Estimate the Frontal Area of the vehicle, and the Drag Coefficient.
   2. Frontal area is the amount of area covered by the front profile of your vehicle. A height of 1.5 meters and a width of 2 meters for your vehicle approximates to 1.5 \* 2 meters, or 3 meters. Feet can be converted into meters by multiplying by 0.3, and inches to centimeters by multiplying by 2.5.
   3. Drag coefficient is a quantity without a unit to determine the effect of drag on an object. Most cars are between 0.2 and 0.3; 0.2 is sleeker, 0.3 is boxier; a Prius is around 0.26.
   4. A drop-down menu with approximate values for different vehicle types is being planned.
4. Waiting
   1. The algorithm needs time to gather elevation and geocoding data.
   2. It will typically take anywhere from about 10 seconds to about a minute for a result to be returned, due to query limits.
5. Output
   1. The output will be displayed on the map on the main page.

* The route is shown by the green line.
* The red and green markers at the ends of the green line are the start and end points, respectively.
* There are icons along the green line. These show things that happen at different points along the route. There are three types of icons: stop light, stop sign, and regular markers (blue).
* Hovering the mouse over these icons will show text instructions containing the type of turn, name of the new road, and distance to the next icon.
  1. There is also a new page containing a list of the written instructions, as well as a new map with the same information as described above.
  2. This will appear as a popup. Make sure your browser has popups for this page enabled. If it is blocked, there will usually be an icon around the area of the browser text box. In this example below, in Google Chrome, the icon with the "X" over it can be clicked and a menu will pop up. Click on the URL underlined in blue and the page will pop up in a new window. For best results, click the Maximize button for the window.



* 1. This page has all of the same information described above, just in a list format. There are still the traffic light/sign icons, with the addition of turn icons. The overall distance is shown on the top.
  2. This page is printable. Simply press Crtl + P and select the best options. Printing the map itself can be tricky.

**Disclaimers**

* Written instructions may not be correct, nor may be the type of intersection displayed in the icons.
* There may be unnecessary turns or routes.
* The route may contain illegal one-way road usage, or missing U-turns. Use the nearest appropriate road instead. Use your common sense.
* This program will generate a map for the most fuel efficient route, but this will not necessarily be the fastest or least-distance route.

*For questions related to the function of Google Maps please refer to the button listed as "Google Maps Help".*

# Evaluation

The project is complete in the sense that it functions well as a proof-of-concept model. EcoCartographer overall is well structured and has a solid user interface. The program is sophisticated in that it works effectively and accurately. It is powerful enough to find a route that is 10 miles long in a reasonable time window, but it can theoretically find routes many times longer than that. However, there are several aspects of the project which can be improved.

This program can be further improved by the following:

* A main limiting factor is that the elevation queries through the Google Maps JavaScript API take too much time. Most of the run time is spent waiting, as to not exceed the query limit and get OVER\_QUERY\_LIMIT messages instead of data. If the Elevation API can be accessed directly, then the time needed to find the route will reduce to around 10% of what it is now.
* Factoring in the efficiency of the engine given a required force output. This change will enhance the accuracy of the algorithm by making the model reflect real life more. Due to the unavailability of ubiquitous and precise data correlating fuel efficiency to engine output, it was not possible to factor in this phenomenon. If data can be obtained from the manufacturers of vehicles, it would be possible to create a database of this information and use it to improve the accuracy of the algorithm.
* Creating drop-down menus for the easier selection of vehicles, and obviating the need for users to look up the information themselves. It would have a list of generic vehicle types, with approximate values for mass, drag-coefficient, and surface area. This would be to improve the user experience.
* Instructions can be inaccurate, with routes going on illegal one-way roads, impossible turns being made, and stop signs and stop lights not being where they should be. This is essentially a data problem. Once data can be generated that can at least predict one-way roads, it will be possible to filter out some of these errors and eliminate those inaccuracies.
* Better formatting for the front-end in general, because it is currently minimalistic in design, especially the output page and the auxiliary pages.
* The methods for finding intersections with stop signs can be dramatically improved, since the method designed works passably, but not particularly accurately. It is based off of a very simple conditional statement having to do with the direction of the turn and speeds of the two roads, which is a generalization.
* Compatibility for Internet Explorer and Safari, and testing on Apple and Linux operating systems.

The work Collin did was on the website design and styling. It is a result of (his words) tedious trial and error, a process that he would have liked to speed up. Going for a professional feeling, the webpage design was meant to be simple but functional. Going back he would have liked to have found a way to add more color to the webpage to create a vibrant but professional feel to catch the attention of the user. The overall user friendliness of the website is well done for the most part, but some of the units or specific factors may be hard for the average user to understand.

We believe this project is sophisticated in concept, and can be improved significantly through further development to eliminate ambiguous and redundant logic, improve efficiency at the time and algorithmic levels, and make additions and modifications. It achieves the goal of being easily obtainable once it is hosted on a server. In all of these respects, it solves the identified problem.

# References

* Baxter, Eric. "What's the Difference between Torque and Horsepower? - HowStuffWorks." *HowStuffWorks*. HowStuffWorks, n.d. Web. 13 Jan. 2015. <http://auto.howstuffworks.com/difference-between-torque-and-horsepower.htm>.
* Engineering ToolBox. "Engineering ToolBox." *Engineering ToolBox*. Engineering ToolBox, n.d. Web. 24 Jan. 2015. <http://www.engineeringtoolbox.com/>.
* Google. "Google Maps JavaScript API V3 Reference." *Google Developers*. Google, n.d. Web. 17 Mar. 2015. <https%3A%2F%2Fdevelopers.google.com%2Fmaps%2Fdocumentation%2Fjavascript%2Freference>.
* JQuery Foundation. "JQuery." *JQuery*. JQuery Foundation, n.d. Web. 10 Mar. 2015. <http://jquery.com/>.
* Office of Transportation and Air Quality, U.S. Environmental Protection Agency, U.S. Department of Energy. "Fuel Economy." *Fuel Economy*. U.S. Department of Energy, n.d. Web. 26 Feb. 2015. <http://www.fueleconomy.gov>.
* OpenStreetMap. "Overpass API." *OpenStreetMap Wiki*. OpenStreetMap, n.d. Web. 29 Jan. 2015. <http://wiki.openstreetmap.org/wiki/Overpass\_API>.
* Refsnes Data. "HTML, CSS, JavaScript." *W3Schools Online Web Tutorials*. Refsnes Data, n.d. Web. 16 Mar. 2015. <http://www.w3schools.com/>.
* Schiff, Jennifer L. "13 Simple Tips for Improving Your Web Design." *CIO*. CXO Media Inc., 4 June 2014. Web. 23 Jan. 2015. <http://www.cio.com/article/2385250/online-marketing/13-simple-tips-for-improving-your-web-design.html>.
* Siegel, Ethan. "What Does Torque in a Car Do?" *Starts With A Bang*. ScienceBlogs LLC, 21 Apr. 2009. Web. 31 Dec. 2014. <http://scienceblogs.com/startswithabang/2009/04/21/what-does-torque-in-a-car-do/>.
* Stack Exchange, Inc. "StackOverflow." *StackOverflow*. Stack Exchange, Inc., n.d. Web. 16 Mar. 2015. <http%3A%2F%2Fstackoverflow.com%2F>.
* Wikipedia. "Brake Specific Fuel Consumption." *Wikipedia*. Wikimedia Foundation, n.d. Web. 17 Jan. 2015. <http://en.wikipedia.org/wiki/Brake\_specific\_fuel\_consumption>.