Version 3.1

**Summary:**

This version reduced computation time in the intersections function group, by reducing the number of for loops and function calls. Furthermore, a method for removing dead-end intersections was introduced, which decreased function calls further. Additionally, a method for detecting the directions of roads was created by adding extra directions data. Finally, an attempt was made to use the Elevation API instead of the Service, because of the decreased time between requests and the theoretical increase in queries at a time.

**Notes:**

I downloaded Notepad++ 2.7.5 on 3/24/15 after looking some on the internet. I configured it so that it would automatically save changes.

I also brainstormed ways to improve the efficiency of the algorithm. It comes in three ways: Use of Elevation API and some interpolation, reversing the directions of the queries to generate one-way roads and also change it so that intersections can only connect forwards, not backwards also, and streamlining the Branch algorithm. I will conduct research into how to extract the information from the Geocoding URL without violating CORS. As for interpolation, I had the idea that it could be done, since I have to simultaneously include provisions for the query generating results for all pts with the same coordinate; it may or may not be more efficient. For the intersections/Branch algorithm improvements, I came up with the idea of lopping off intersections that only have one connection, and merging intersections into one long path if they only have two connections. This will drastically cut down on computing time, and therefore decrease the amount of time needed to complete the calculations. Also, I had the idea of making intermediate significant points, so that the Branch optimization process is split into two steps. Significant intersections will have the most fuel efficient route found between it and the other nearby major intersections, and then those will ultimately have the algorithm applied to them. It will sacrifice some accuracy, but this is a heuristic change, and will likely be necessary for the algorithm to speed up.

3/24/15: 1 hour

3/25/15: 45 min

3/26/15: 45 min

I got a new editor, which is called Notepad++

3/27/15: 1.5 hours

3/28/15: 1 hour

3/31/15: 2 hours

I had the idea that it would be least complicated to find *only* the intersections that connected without data, because a) it would take more computing power to resort the data twice and b) be more complicated to code. This way that I found, I find all the intersections from the original series data, and build lists of connections. Then, I strip intersections altogether that have only 1 or 2 connections, and rebuild the series data from the remaining intersections. Then, normal computation carries on.

This method appears to work, but does not noticeably lower amount of time required. Added an intersection limit.

I added a while loop to continually remove forbidden intersections until none remained, as demonstrated in output 3.1.3. I also discovered that the connections arrays did not reset after each time, so they had artificially high numbers of intersections, nor did they count the same connection index only once. This was fixed, and the result was that only two intersections (start and end) remained. This was actually caused by an error in logic. The error was corrected, and the original basic while loop was replaced. The result is seen in output 3.1.2. That does not represent all the intersections.

After the algorithm was done, the number of intersections was reduced by at least 60%, as seen in outputs 3.1.4 and 3.1.5.

I timed the amount of time it takes to run parts of the program (recorded in output 3.1.1):

* Intersections: generation, adding signals, compiling data: 3193 ms
* Branches: calculation of: 13 ms
* Optimal: sorting through branches, finding the best route: 25 ms

*Sample: 6936 Millbridge Road to 330 Knollwood Street*

After some modifications (removing functions defined in loops and removing all traces of points having once been reversed) and the adding of backwards connections in addition to forwards to not remove quite so many intersections:

* Intersections: 3230 ms
  + Intersection\_gen(): 33 ms
  + Add\_signals(): 2346 ms
  + Intersection\_build(): 851 ms

I then made all functions inside the intersection\_build() function inside the Network function scope, and removed extra logic from the add\_signals() function, because the logic was not crucial to the performance of the algorithm. It essentially just double-checked things.

* Intersections: 929 ms
  + Intersection\_gen(): 39 ms
  + Add\_signals: 178 ms
  + Intersection\_build(): 713 ms

This is satisfactory. I now measured the branch algorithm for the same route, and it took more than 3 seconds to complete. I decided to shorten the maximum distance for the route to be 1.4 times the linear distance. The new time is in 900 ms range, which is also acceptable compared to more than 3 seconds.

I also made all of the routes used in the directions service go both ways, and made the intersections algorithm search only forward, so it should only choose routes which the vehicle would go forward on, instead of illegal turns or wrong sides of the road.

4/1/15: 2.5 hours

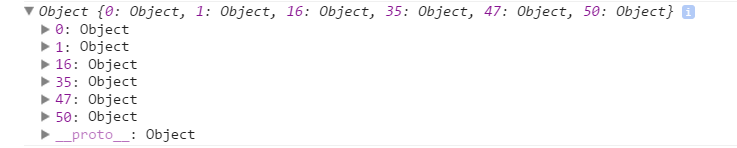
Time: 9.5 hours

**Outputs:**

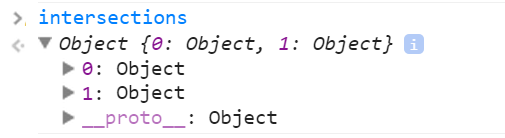
*Output 3.1.1:*



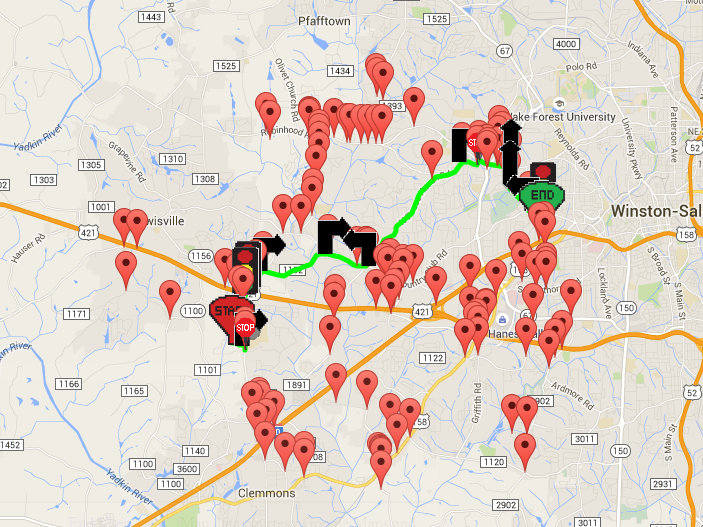
*Output 3.1.2:*



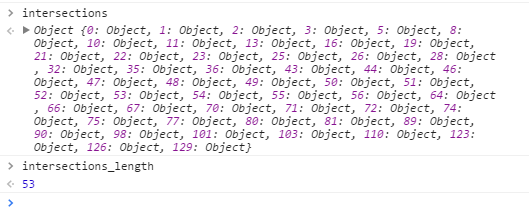
*Output 3.1.3:*



*Output 3.1.4:*



*Output 3.1.5:*



**New Code:**

*Code 3.1.1:*

New code for generating intersections.

***function*** intersection\_build**(){** //Get data for intersections

intersections\_strip**(**2**);**

console**.**log**(**intersections**);**

//rebuild series without deleted intersections

***var*** series **=** series\_build**();**

//assign data to final intersections

***for*(*var*** i ***in*** intersections**){**

intersections**[**i**].**connections **=** **{};**

**}**

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

***if*(**series**[**i**].**int\_index**!==**series**[**i**+**1**].**int\_index**){**

intersections**[**series**[**i**].**int\_index**].**connections**[**series**[**i**+**1**].**int\_index**]** **=** **{**

index**:**series**[**i**+**1**].**int\_index**,**

start**:**series**[**i**].**pt\_index**,**

end**:**series**[**i**+**1**].**pt\_index**+**1**,**

point**:**intersections**[**series**[**i**].**int\_index**].**coords**,**

path**:[]**

**};**

**}**

**}**

//add point data to intersections

//need to use JSON to make copy in order to safely reverse next/last

***for*(*var*** i ***in*** intersections**){**

***for*(*var*** j ***in*** intersections**[**i**].**connections**){**

intersections**[**i**].**connections**[**j**].**path **=** pts**.**slice**(**intersections**[**i**].**connections**[**j**].**start**,**intersections**[**i**].**connections**[**j**].**end**).**map**(**ReturnNewPt**);**

**}**

**}**

***var*** closest\_end **=** 1**;**

***var*** closest\_start **=** 0**;**

***for*(*var*** i ***in*** intersections**){**

***var*** c **=** intersections**[**i**].**connections**;**

***for*(*var*** j ***in*** c**){**

***var*** new\_slice **=** c**[**j**].**path**.**slice**(**0**,**c**[**j**].**path**.**length**-**1**);**

***var*** data **=** Sums**(**new\_slice**);**

c**[**j**].**energy\_to **=** data**.**energy\_to**;**

c**[**j**].**speed **=** data**.**speed**;**

c**[**j**].**distance\_to **=** data**.**distance\_to**;**

c**[**j**].**time\_to **=** data**.**time\_to**;**

**}**

**}**

**}**

*Code 3.1.2:*

Functions that support Code 3.1.1.

***function*** series\_build**(){** //makes a serialized list correlating intersection index to point index

***var*** output **=** **[];** //Terminators for paths

***for*(*var*** i ***in*** intersections**){**

***for*(*var*** j ***in*** intersections**[**i**].**indexes**){**

output**.**push**({**pt\_index**:**intersections**[**i**].**indexes**[**j**],** int\_index**:**i**});**

**}**

**}**

output**.**sort**(*function*(**a**,**b**){*return*** a**.**pt\_index**-**b**.**pt\_index**});**

console**.**log**(**output**);**

***return*** output**;**

**}**

***function*** intersections\_strip**(**maximum**){** //Strips intersections with connections.length <= maximum

//remove extra intersections

***var*** series**;**

***var*** output **=** ***true*;**

//reset the connections array for each intersection

***for*(*var*** i ***in*** intersections**){**

intersections**[**i**].**connections **=** **[];**

**}**

//get series for original intersections

series **=** series\_build**();**

//compile preliminary list of connections for each intersection.

//Array is easier to search through than object.

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

//does connects to itself nor duplicate existing connections

***if*(**series**[**i**].**int\_index**!==**series**[**i**+**1**].**int\_index**){**

***if*(**intersections**[**series**[**i**].**int\_index**].**connections**.**indexOf**(**series**[**i**+**1**].**int\_index**)===-**1**){**

intersections**[**series**[**i**].**int\_index**].**connections**.**push**(**series**[**i**+**1**].**int\_index**);**

intersections**[**series**[**i**+**1**].**int\_index**].**connections**.**push**(**series**[**i**].**int\_index**);**

**}**

**}**

**}**

***for*(*var*** i ***in*** intersections**){**

//is dead-end (1 connection) or part of a larger path (2 connections)

//is not start or end

***if*(**intersections**[**i**].**connections**.**length **<=** maximum **&&** i**!==**"0" **&&** i**!==**"1"**){**

***delete*** intersections**[**i**];**

output **=** ***false*;**

**}**

**}**

***return*** output**;**

**}**

***function*** ReturnNewPt**(**a**){**

***var*** b **=** JSON**.**parse**(**JSON**.**stringify**(**a**));**

b**.**pt **=** a**.**pt**;**

***return*** b**;**

**};**

***function*** Sums**(**path**){**

***var*** energy **=** 0**;**

***var*** speed **=** 0**;**

***var*** dist **=** 0**;**

***for*(*var*** S**=**0**;** S**<**path**.**length**;** S**++){**

energy **+=** **((**path**[**S**].**next**.**dist**===**0**)?**0**:**Math**.**max**(**path**[**S**].**next**.**force**,** 0**)\***path**[**S**].**next**.**dist**);**

dist **+=** path**[**S**].**next**.**dist**;**

speed **+=** path**[**S**].**speed **\*** path**[**S**].**next**.**dist**;**

**}**

***return*** **{**energy\_to**:**energy**,** speed**:**speed**/**dist**,** distance\_to**:**dist**,** time\_to**:**dist**/(**speed**/**dist**)};**

**}**

*Code 3.1.3:*

Optimizations for add\_signals to decrease time

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

//iterate through pts, add signals

***var*** int\_indexes **=** **[];**

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

***if*(**pts**[**i**].**isIntersection**){**

pts**[**i**].**stop **=** "sign"**;**

int\_indexes**.**push**(**i**);**

**}**

**}**

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

***var*** matched **=** ***false*;**

***for*(*var*** j**=**0**;** j**<**int\_indexes**.**length**;** j**++){**

***if*(**Math**.**abs**(**signals**[**i**].**lat**()-**pts**[**int\_indexes**[**j**]].**pt**.**lat**())<**0.001 **&&** Math**.**abs**(**signals**[**i**].**lng**()-**pts**[**int\_indexes**[**j**]].**pt**.**lng**())<**0.001**){**

pts**[**int\_indexes**[**j**]].**stop **=** "light"**;**

matched **=** ***true*;**

**}**

**}**

***if*(!**matched**){**

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

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

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

***for*(*var*** k ***in*** pts**[**j**].**references**){**

pts**[**k**].**stop **=** "light"**;**

**}**

***break*;**

**}**

**}**

**}**

**}**

**}**

*Code 3.1.4:*

New scheme for Google Directions.

//list of intersections used for directions service

***var*** wpts1 **=** **[** **{**location**:**wpts\_options**[**2**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**3**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**1**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**4**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**0**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**5**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**7**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**6**],** stopover**:*true*}];**

***var*** wpts2 **=** **[** **{**location**:**wpts\_options**[**1**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**7**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**2**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**6**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**3**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**5**],** stopover**:*true*},**

**{**location**:**wpts\_options**[**4**],** stopover**:*true*},**

**{**location**:**center**,** stopover**:*true*}];**

***var*** wpts3 **=** **[** **{**location**:**center**,** stopover**:*true*}];**

***var*** dir\_requests **=** **[**

**{**origin**:**sp**,** destination**:**ep**,** avoidHighways**:*true*,** waypoints**:**wpts1**,** travelMode**:**google**.**maps**.**DirectionsTravelMode**.**DRIVING**},**

**{**origin**:**ep**,** destination**:**sp**,** avoidHighways**:*true*,** waypoints**:**wpts1**.**reverse**(),** travelMode**:**google**.**maps**.**DirectionsTravelMode**.**DRIVING**},**

**{**origin**:**sp**,** destination**:**ep**,** avoidHighways**:*true*,** waypoints**:**wpts2**.**reverse**(),** travelMode**:**google**.**maps**.**DirectionsTravelMode**.**DRIVING**},**

**{**origin**:**ep**,** destination**:**sp**,** avoidHighways**:*true*,** waypoints**:**wpts2**,** travelMode**:**google**.**maps**.**DirectionsTravelMode**.**DRIVING**},**

**{**origin**:**sp**,** destination**:**ep**,** avoidHighways**:*false*,** waypoints**:**wpts3**,** travelMode**:**google**.**maps**.**DirectionsTravelMode**.**DRIVING**}**

**];**

Version 3.2

**Summary:**

This version optimized elevation more by generating intersections first that would be included, and then generating elevation data that is in those intersections used. This led to a small net decrease in elevation data. Finally, a Python Bottle server was made to facilitate the Elevation API in order to decrease queries and take advantage of lower query limit times.

**Notes:**

Next step was to optimize the elevation data gathering. First, I re-wrote the program so that the intersections were generated first, then elevation data retrieved for the relevant points. This took about 4 hours to do, because there were issues with finding problems in code, and figuring out the best way to reconfigure the program. Intersections\_build() was split up, because the first part gets relevant intersections, and the second part assigns pts to the paths. Since a copy of the point is made, it was not possible to assign copies and then change the data.

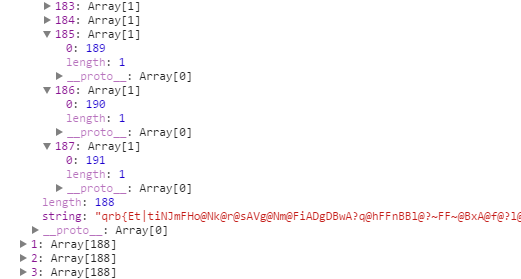
I also experimented with bottle.py beginning on April 8th. I wrote a short program to make a local server (which in theory could be hosted from a domain) to retrieve the elevation data from the Elevation API (since CORS restrictions do not apply in Python) and then pass it on to the Ajax request from script.js. This took about 3 hours, with experimenting and then needing to find and add the appropriate response headers (since initially the ‘no “Access-Control-Allow-Origin’ error message was present). I used the Bottle Documentation as a source to fix this problem on April 12, 2015.

To decrease the number of queries used, I looked up the format for the Encoded Polyline Algorithm. I tried to make a parser for that in Python, but I had problems with rendering negatives in binary. I later discovered a built-in function under google.maps.geometry.encoding.encodePath() that accomplished this. I substituted this format in lieu of “lat,lng|lat,lng|…”, but there were errors.

**Outputs:**

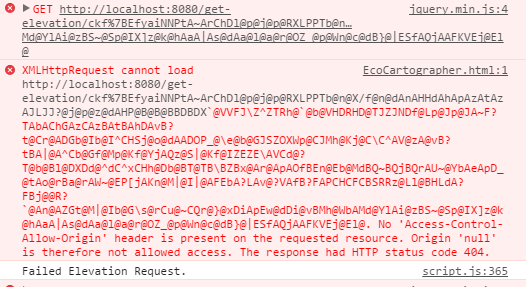
*Output 3.2.1:*

Method for generating encoded polylines.

**

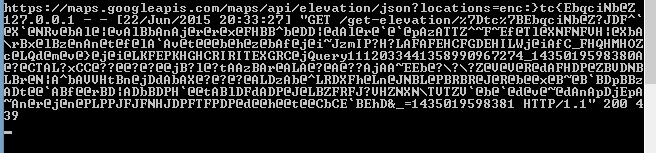
*Output 3.2.2:*

An output from the console when there was an error with the query caused by CORS.

**

*Output 3.2.3:*

Output from the Python that shows an error with the URL sent by Ajax in jQuery.

**

**New Code:**

*Code 3.2.1:*

New split up intersection\_build() to generate elevation data from used intersections.

***function*** intersection\_build**(){** //Get data for intersections

intersections\_strip**(**1**);**

console**.**log**(**intersections**);**

//rebuild series without deleted intersections

***var*** series **=** series\_build**();**

//assign data to final intersections

***for*(*var*** i ***in*** intersections**){**

intersections**[**i**].**connections **=** **{};**

**}**

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

***if*(**series**[**i**].**int\_index**!==**series**[**i**+**1**].**int\_index**){**

***if*(!(**series**[**i**+**1**].**int\_index ***in*** intersections**[**series**[**i**].**int\_index**].**connections**)){**

***for*(*var*** j**=**series**[**i**].**pt\_index**;** j**<**series**[**i**+**1**].**pt\_index**+**1**;** j**++){**

pts**[**j**].**isInIntersection **=** ***true*;**

**}**

intersections**[**series**[**i**].**int\_index**].**connections**[**series**[**i**+**1**].**int\_index**]** **=** **{**

index**:**series**[**i**+**1**].**int\_index**,**

start**:**series**[**i**].**pt\_index**,**

end**:**series**[**i**+**1**].**pt\_index**+**1**,**

point**:**intersections**[**series**[**i**].**int\_index**].**coords**,**

path**:[]**

**}**

**};**

**}**

**}**

elevation**();**

**}**

***function*** intersection\_build\_part2**(){**

//add point data to intersections

//need to use JSON to make copy in order to safely reverse next/last

***for*(*var*** i ***in*** intersections**){**

***for*(*var*** j ***in*** intersections**[**i**].**connections**){**

intersections**[**i**].**connections**[**j**].**path **=** pts**.**slice**(**intersections**[**i**].**connections**[**j**].**start**,**intersections**[**i**].**connections**[**j**].**end**).**map**(**ReturnNewPt**);**

**}**

**}**

***var*** closest\_end **=** 1**;**

***var*** closest\_start **=** 0**;**

***for*(*var*** i ***in*** intersections**){**

***var*** c **=** intersections**[**i**].**connections**;**

***for*(*var*** j ***in*** c**){**

***var*** new\_slice **=** c**[**j**].**path**.**slice**(**0**,**c**[**j**].**path**.**length**-**1**);**

***var*** data **=** Sums**(**new\_slice**);**

c**[**j**].**energy\_to **=** data**.**energy\_to**;**

c**[**j**].**speed **=** data**.**speed**;**

c**[**j**].**distance\_to **=** data**.**distance\_to**;**

c**[**j**].**time\_to **=** data**.**time\_to**;**

**}**

**}**

compute**();**

**}**

*Code 3.2.2:*

JavaScript to send/get data to/from bottle server.

***function*** BottleRequest**(**data**,** delay**){**

***var*** waiter **=** setTimeout**(*function*(){**

$**.**ajax**({**

method**:**"GET"**,**

url**:**"http://localhost:8080/get-elevation/"**+**data**.**string**,**

dataType**:**'json'**,**

success**:*function*(**a**){**ElevationCallback**(**a**,**data**)},**

error**:*function*(){**ElevationError**(**data**)}**

**});**

**},**delay**);**

**}**

***function*** ElevationCallback**(**elevation**,**points**){**

***if*(**elevation**.**status **===** "OK"**){**

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

***var*** refs **=** points**[**i**];**

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

pts**[**refs**[**j**]].**elevation **=** elevation**.**results**[**i**].**elevation**;**

**}**

**}**

completion\_counter**++;**

elev\_value**.**innerText **=** parseInt**(**Math**.**round**(**completion\_counter**/**expected\_requests**\***100**))+**"%"**;**

**}*else*{**

ElevationError**(**points**);**

**}**

**}**

***function*** ElevationError**(**data**){**

failed\_counter**++;**

failed\_requests**.**push**(**data**);**

console**.**log**(**"Failed Elevation Request."**);**

**}**

*Code 3.2.3:*

Python bottle.py server. This would be run locally on a computer.

**from** bottle **import** Bottle**,** route**,** run**,** static\_file**,** get**,** request**,** response**,** hook

**from** urllib **import** urlopen**,** unquote

@hook**(**'after\_request'**)**

**def** **enable\_cors():**

response**.**headers**[**'Access-Control-Allow-Origin'**]** **=** '\*'

@route**(**'/get-elevation/<query>'**)**

**def** **func(**query**):**

query **=** 'https://maps.googleapis.com/maps/api/elevation/json?locations=enc:'**+**query

**print** query

data **=** urlopen**(**query**)**

**return** data**.**read**()**

run**(**host**=**'localhost'**,** port**=**8080**,** debug**=True)**

Version 4.1

**Summary:**

This version included preliminary translation into Python, as well as a number of innovations from version 3. On a structural level, the Python module was organized into sub-modules encompassing general methods, data classes, query classes, IO, and miscellaneous functions, as well as external modules such as Six and Polyline. The program makes use of tuples instead of objects and specific data structures for entities like Pts and Intersections, and was designed in such a way as to utilize the iterative abilities of Python.

The innovations included a greater degree of Object Oriented Programming; gathering elevation data only used in a relatively small number of reasonable routes, instead of every point reference; a separate module (added during development) to house query classes; using road names gathered in the Directions API instead of geocoding; and drastic speed increases, because Python is more efficient than JavaScript in general, and because of the data structures used in the program.

**Notes:**

*Reasons for switching to Python:*

I started around 4/23/15 with the translation of the JavaScript into Python. This was done for the following reasons:

1. Python would be much faster for crunching data. I noticed that there was a ton of lag time with JavaScript on just short routes (as demonstrated with the timing done in version 3), and that expansion of routes would only work with a better language for speed. Python is infamous for its efficiency with loops and arrays, and since this program is essentially a big data analyzer, Python would be well adapted.
2. That has further carry over with the user experience, since the user does not have to wait nearly as long.
3. I needed a proxy server anyway (using Python) to get data using the APIs, and reduce time to get that data.
4. It would be a good opportunity to simplify the program down to its essential elements, and also experiment with slightly more object-oriented programming. Also, the States project was very disorganized in its structure, so rewriting the program with function flow in mind would help that problem.
5. I already am reasonably good at Python (certainly as good as JavaScript).

There are some drawbacks, though:

1. The urllib library is nowhere near as efficient as Ajax is with getting data from remote websites. It is likely that the decreased speed of getting the information (since Python is not asynchronous, like JavaScript) combined with the theoretical greater number of requests for the same amount of data would outweigh the time gained by getting to go at ten times the request speed.
2. There must be a way to relay data between the HTML application shell and the Python script.

*Structure:*

The program was built in this basic structure:

* Classes.py: contains class constructors that represent components of the project that have attributes and there are many of them, such as LatLngs, Pts, Intersections, Connections (to be nested inside intersections), and Bounds. They all have methods associated with them, like distanceTo for LatLngs.
* Util.py: had miscellaneous commands like array operations or formatting functions.
* Ecio.py: input/output stream, which writes console items to an HTML file, as well as parse final outputs into JSON, similarly to Google Directions API.
* Query.py: Classes which control the API requests. It handles errors and timing (to not go over query limits and repeat until success is achieved).
* Main.py: has all the commands for the program. It is essentially a collection of functions that all return (eventually) to the main function. There is not the daisy-chaining and global variables used in the JavaScript; everything is passed by arguments and returns. The functions could theoretically be unified into a single, very long function, but they are broken up to reduce cyclomatic complexity, and separate tasks for the benefit of the programmer.
* The program is not broken up into more than these submodules to simplify the imports needed.
* In other words, there are five modules because the types of tasks in the program naturally broke up that way. Also, having everything more or less crammed into one module would make it difficult to find things, like in the JavaScript versions.
* There are two additional downloaded modules: Polyline, which in turn requires Six. Polyline is used to convert latitude/longitudes into Encoded Polyline Format, which is used for Google Maps API queries. Six is a supporting module.

*Observations*

In general, it takes far less time to handle data. For one, using a 2 distance multiplier instead of 1.4 and no intersection count limit in Python, despite the seeming complications, has an average time of 0.02 to 0.03 seconds, whereas in JavaScript is was about 0.03 to 0.04 seconds. Also, the total time to compile intersections is considerably less than the JavaScript times, averaging about 0.2 to 0.3 seconds as opposed to 0.7 to 0.8 seconds.

Python is much easier to use than JavaScript, because the loops are easier to use with structures like tuples, and there are more useful and plentiful built-in functions, which drastically reduces run time. Additionally, this built-in nature of Python make the code faster to write and easier to read. It can also simplify to the point that I have difficulty understanding data structures, since I use almost exclusively arrays, and there is no good way to view internal data structure like the JavaScript console has. A major breakthrough was made on 5/10/15, when there were paths made out of intersections generated using a recursive function similar to Branch().

By 5/24/15, force was calculated and optimal routes were beginning to be found. (Output 4.1.1)

This query is clearly significantly larger than the other one the old, Javascript algorithm could handle. It also did the calculations in a decent time.

On 5/25/2015, the route in output 4.1.2 was generated.

When I overlaid the pts and intersections, I discovered that there were two possible ways to go from where the route deviates to where it rejoins with the expected route: (in output 4.1.3)

*Other Procedures:*

On Thursday, 5/7/14 during the TSA meeting, Andy Martinek helped us set up an account for the FTP server devtano.com. devtano.com is the website our advisor, Mr. Pantano, created to facilitate the TSA functions for our chapter. Subsequently, the states version of the project was uploaded, and the main page was edited to create a link to the project.

On Thursday, 5/14/15 during the TSA meeting, Collin and I drafted inquiries to auto companies requesting data on the engines.

On Friday, 5/15/15, all four team members met at Ali’s house, where we worked for 2 hours. The work done was to set up a shell for the website, understand how the website was going to be structured, and come up with ideas for implementing this. I did editing work on the HTML and CSS pages Akshat had prepared. We also came up with the following ideas for the website, which need to be implemented:

* To make a full-page shell for the Google Map interface, this would be given to me when it was finished to integrate the scripting.
* To make a single HTML file with the header and footer, and have an iframe be the content. This would remove redundancy with copying the header to each document, and therefore simplify the process of editing and also the uploading to the FTP server.
* To redo the End-User Documentation formatting, to match that of the templates Akshat had already created.
* Once preliminary versions of these documents were created, they would be uploaded to devtano.com, so that editing could be made conveniently be all members of the team.

On Thursday, 5/21/15, there was another meeting. Only Daniel, Ali and Akshat were present. Daniel attempted to find a way to accurately calculate force factoring in acceleration and deceleration. A plan was also formulated for reducing query sizes, with the circular net used to generate a Google Maps Directions API Query being changed to a diamond, since ellipses would be difficult to calculate. This would remove excess data that would likely not be used, and would especially speed up elevation calculation, since there would be significantly (arbitrary guess: 15-30%) fewer points to calculate elevation data for. It would also take less time to gather the Directions API data.

On Sunday, 5/24/15, I spent about 2.5 hours trying to find a way to have the Python script run from my Raspberry Pi, laptop or devtano.com so that it could be accessed remotely. I know how to do a local host, but that is irrelevant. I posted a question on Stack Overflow, asking for detailed instructions for setting up the script on the Host Gator site.

By Tuesday, 5/26/15, a preliminary version of the Python was finished. Some ideas for optimizations:

* May 9
  + make ellipse instead of sphere
  + chunk the google-optimized route and solve for several midpoints
  + make initial routes, choose most promising and remove unnecessary points (DONE: 4.1)
  + interpolate elevation data (DONE: 4.2)
  + calculate fuel saved, money saved, etc. (DONE: 4.2)
* May 23
  + create another module for classes to control query rates (improve readability, accuracy) (DONE: 4.1)
  + collect road data from returned query, not geocoding (DONE: 4.1)
  + make instructions for multiple routes, but choose drastically different (instead of slightly different, append-something-here routes) (Asked about on Stack Overflow. Attempted in 4.2)

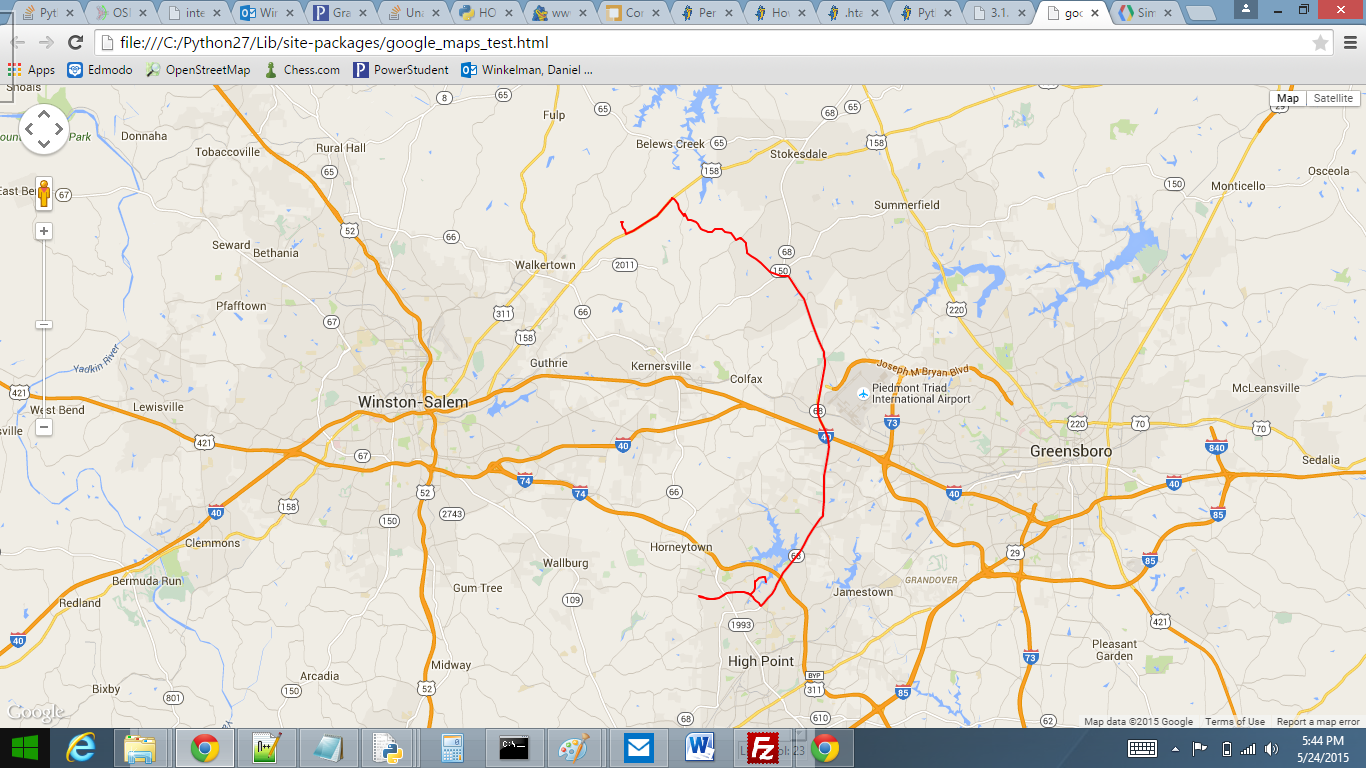
*Next Versions:*

The next version will cut down on the amount of elevation data collected by using interpolation and compressed networks. This will additionally reduce time importing the Directions API data.

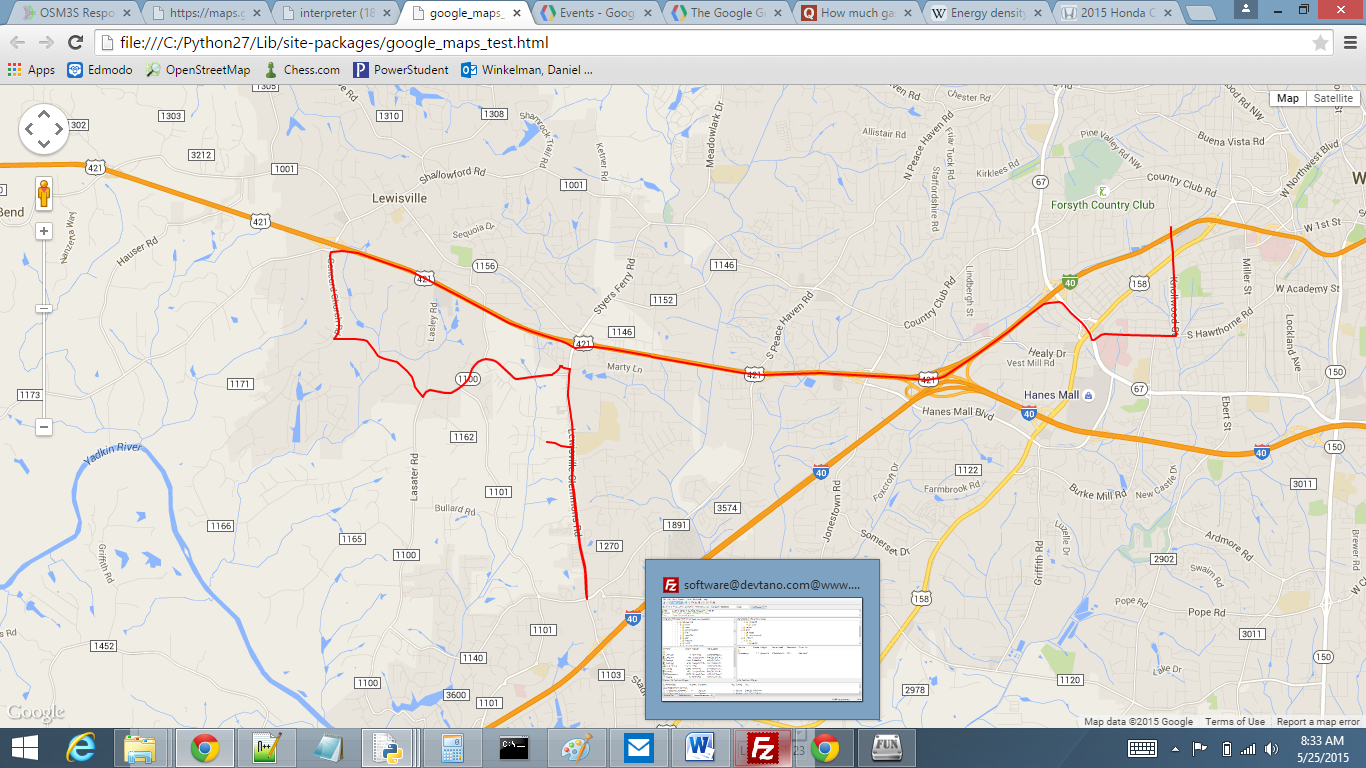
The version after that will improve pruning measures, and not calculate detailed forces, instructions, etc. for very similar routes. This will be done by comparing the number of shared consecutive intersections.

**Outputs:**

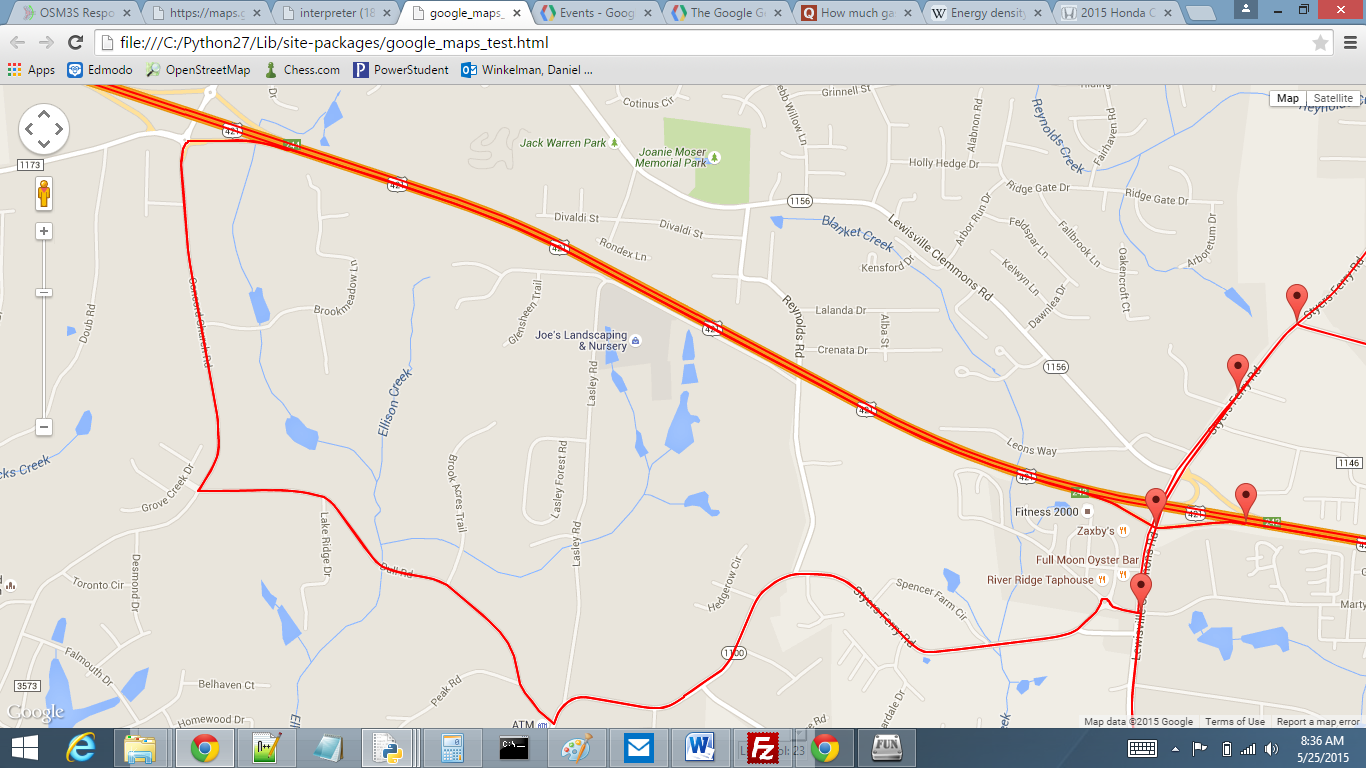
*Output 4.1.1:*



*Output 4.1.2:*



*Output 4.1.3:*



**New Code:**

*Code 4.1.1:*

Most basic classes used. LatLng is a coordinate representation with some math methods (like distance, angles, etc.); Bounds are used to generate an Overpass API query; Pt is the main data structure in the algorithm, and makes up the polylines used.

**class** **Pt(**object**):**

'''Used for operations of coordinates in the network'''

**def** **\_\_init\_\_(**self**,** latLng**,** index**,** speed**,** roadname**,** isIntersection**=False):**

'''Used for operations of coordinates in the network. Accepts LatLng object, integer and boolean reflecting status in the initial network returned'''

self**.**pt **=** latLng

self**.**index **=** index

self**.**isInterection **=** isIntersection

self**.**string **=** latLng**.**toStringURL**()**

self**.**isStoplight **=** **False**

self**.**isStopsign **=** isIntersection

self**.**roadname **=** roadname

self**.**elevation **=** 0

self**.**speed **=** speed

self**.**references **=** **None**

self**.**next **=** **None**

self**.**last **=** **None**

self**.**temp\_speed **=** speed

**def** **getEnergyNext(**self**,** vehicle**):**

'''Get force to another Pt. Recommend adjacent.'''

angle **=** math**.**atan**((**self**.**next**.**elevation **-** self**.**elevation**)/**self**.**pt**.**distanceTo**(**self**.**next**.**pt**))** #positive angle is a rise

Fpar **=** GRAVITY **\*** vehicle**.**mass **\*** math**.**sin**(**angle**)** #force of moving the vehicle up/down elevations

Ffriction **=** GRAVITY **\*** vehicle**.**mass **\*** math**.**cos**(**angle**)** **\*** ROLLINGRESISTANCE #force of rolling resistance

Fdrag **=** 0.5 **\*** vehicle**.**cd **\*** AIRDENSITY **\*** vehicle**.**area **\*** self**.**speed**\*\***2 #force of drag

**return** **(**Fpar **+** Ffriction **+** Fdrag**)** **\*** self**.**pt**.**distanceTo**(**self**.**next**.**pt**)** #total energy to go length of road, maintaining velocity in Joules

**class** **Bounds(**object**):**

'''Used to create a box from coordinates'''

**def** **\_\_init\_\_(**self**,**north**,**south**,**east**,**west**):**

self**.**north **=** north

self**.**south **=** south

self**.**west **=** west

self**.**east **=** east

**def** **isIn(**self**,**latLng**):**

'''Checks if LatLng is inside bounds'''

**if** self**.**north **>=** latLng**.**lat **>=** self**.**south **and** self**.**east **>=** self**.**lng **>=** self**.**west**:**

**return** **True**

**else:**

**return** **False**

**def** **toOverpassString(**self**):**

'''Returns a string to be used in Overpass API query'''

**return** '(%f,%f,%f,%f)' **%** **(**self**.**south**,**self**.**west**,**self**.**north**,**self**.**east**)**

**class** **LatLng(**object**):**

'''Used for mathemetical representation of latitude-longitude coordinates'''

**def** **\_\_init\_\_(**self**,**lat**,**lng**):**

'''Used for mathemetical representation of latitude-longitude coordinates'''

self**.**lat **=** float**(**lat**)**

self**.**lng **=** float**(**lng**)**

**def** **toStringURL(**self**):**

'''Returns string used for Google Maps URLs'''

**return** str**(**round**(**self**.**lat**,**5**))** **+** ',' **+** str**(**round**(**self**.**lng**,**5**))**

**def** **toStringParenth(**self**):**

'''Returns string like toStringURL with parenthesis'''

**return** '(' **+** self**.**toStringURL**()** **+** ')'

**def** **distanceTo(**self**,**a**):**

'''Returns the distance between LatLng used to call operation, and another LatLng, in meters'''

'''http://www.movable-type.co.uk/scripts/latlong.html (More accurate, spherical representation).'''

R **=** 6371000**;**

phi1 **=** math**.**radians**(**self**.**lat**)**

phi2 **=** math**.**radians**(**a**.**lat**)**

deltaphi **=** math**.**radians**(**a**.**lat **-** self**.**lat**)**

deltalam **=** math**.**radians**(**a**.**lng **-** self**.**lng**)**

b **=** math**.**sin**(**deltaphi**/**2**)** **\*** math**.**sin**(**deltaphi**/**2**)** **+** math**.**cos**(**phi1**)** **\*** math**.**cos**(**phi2**)** **\*** math**.**sin**(**deltalam**/**2**)** **\*** math**.**sin**(**deltalam**/**2**)**

c **=** 2 **\*** math**.**atan2**(**math**.**sqrt**(**b**),** math**.**sqrt**(**1**-**b**))**

**return** R **\*** c

**def** **angleAsCenter(**self**,**a**,**b**):**

'''Returns the degree angle between 2 LatLngs and the LatLng used to call operation, with the caller in the middle'''

lat1**,** lat2 **=** a**.**lat**-**self**.**lat**,** b**.**lat**-**self**.**lat

lng1**,** lng2 **=** a**.**lng**-**self**.**lng**,** b**.**lng**-**self**.**lng

dns1**,** dns2 **=** lat1 **\*** DIAMETER**,** lat2 **\*** DIAMETER

dew1 **=** lng1 **\*** abs**(**math**.**cos**(**math**.**radians**(** **(**a**.**lat **+** self**.**lat**)/**2 **)))** **\*** DIAMETER

dew2 **=** lng2 **\*** abs**(**math**.**cos**(**math**.**radians**(** **(**b**.**lat **+** self**.**lat**)/**2 **)))** **\*** DIAMETER

angle1**,** angle2 **=** math**.**degrees**(**math**.**atan2**(**dns1**,**dew1**)),** math**.**degrees**(**math**.**atan2**(**dns2**,**dew2**))**

**if** dns1 **<** 0**:** angle1 **+=** 360

**if** dns2 **<** 0**:** angle2 **+=** 360

**return** angle2 **-** angle1

**def** **directionAngle(**self**,**a**,**b**):**

'''Returns text direction of angle between self and 2 LatLngs'''

angle **=** self**.**angleAsCenter**(**a**,**b**)**

**if** angle **<** **-**225 **or** 0 **<** angle **<** 135**:**

**return** 'right'

**elif** **-**135 **<** angle **<** 0 **or** 225 **<** angle**:**

**return** 'left'

**else:**

**return** 'straight'

**def** **bearing(**self**,** a**):**

'''Returns south/north/east/west of line, based on linear distance'''

lat **=** a**.**lat **-** self**.**lat

lng **=** a**.**lng **-** self**.**lng

dns **=** lat **\*** DIAMETER

dew **=** lng **\*** abs**(**math**.**cos**(**math**.**radians**(** **(**a**.**lat **+** self**.**lat**)/**2 **)))** **\*** DIAMETER

angle **=** math**.**degrees**(**math**.**atan2**(**dns**,** dew**))**

**if** 135 **>=** angle **>** 45**:** **return** 'north'

**elif** 45 **>=** angle **>** **-**45**:** **return** 'east'

**elif** **-**45 **>=** angle **>** **-**135**:** **return** 'south'

**else:** **return** 'west'

*Output 4.1.2:*

Generates list of URLs for the Directions API, given start/end point arguments using the LatLng class.

**def** **GetDirectionsQueries(**start**,** end**):**

t0 **=** time**.**time**()**

'''Generates a list of Google Directions API URLs from the start and end coordinates.'''

'''Working version: 4/23/2015, 4/26/2015 (1 hour)'''

dlat **=** start**.**lat **-** end**.**lat #vector latitude

dlng **=** start**.**lng **-** end**.**lng #vector longitude

center **=** LatLng**(**start**.**lat **-** dlat**/**2**,** start**.**lng **-** dlng**/**2**)** #average of start and end

radius **=** math**.**hypot**(**dlat**,** dlng**)/**2 #distance between start and end /2

angle **=** math**.**radians**(**90**)**

**try:**

angle **=** math**.**atan**(**dlat**/**dlng**)**

**except:** #if dlng == 0, then angle is 90 or 270, depending on start.lat < or > end.lat

**if** start**.**lat **>** end**.**lat**:** #going south; default is set to north, so no else is necessary

angle **=** math**.**radians**(**270**)**

wpts **=** **[]** #coordinates for points along edges of circular net to be made

**for** i **in** range**(**0**,**360**,**45**):**

wpts**.**append**(**LatLng**(**

center**.**lat **+** math**.**sin**(**math**.**radians**(**i **+** angle**))\***radius**,**

center**.**lng **+** math**.**cos**(**math**.**radians**(**i **+** angle**))\***radius

**))**

route\_stem1 **=** 'https://maps.googleapis.com/maps/api/directions/json?origin=' **+** start**.**toStringURL**()** **+** '&destination=' **+** end**.**toStringURL**()** **+** '&waypoints='

route\_stem2 **=** 'https://maps.googleapis.com/maps/api/directions/json?origin=' **+** end**.**toStringURL**()** **+** '&destination=' **+** start**.**toStringURL**()** **+** '&waypoints='

wpts1 **=** **[**3**,**1**,**4**,**0**,**5**,**7**,**6**]**

wpts2 **=** **[**7**,**2**,**6**,**3**,**5**,**4**]** #both sets of coordinates go roughly perpendicular to each other

wpts1r**,** wpts2r **=** wpts1**,** wpts2

wpts1r**.**reverse**()**

wpts2r**.**reverse**()**

routes **=** **[**route\_stem1 **+** '|'**.**join**(**map**(lambda** a**:** wpts**[**a**].**toStringURL**(),** wpts1**))** **+** '&avoid=highways&key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE'**,**

route\_stem2 **+** '|'**.**join**(**map**(lambda** a**:** wpts**[**a**].**toStringURL**(),** wpts1r**))** **+** '&&avoid=highwayskey=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE'**,**

route\_stem1 **+** '|'**.**join**(**map**(lambda** a**:** wpts**[**a**].**toStringURL**(),** wpts2r**))** **+** '&avoid=highways&key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE'**,**

route\_stem2 **+** '|'**.**join**(**map**(lambda** a**:** wpts**[**a**].**toStringURL**(),** wpts2**))** **+** '&avoid=highways&key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE'**,**

route\_stem1 **+** '&key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE'

**]** #reverse used to check for one-way roads

t1 **=** time**.**time**()**

**print** 'GetDirectionsQueries(): ' **+** str**(**t1**-**t0**)**

**return** routes

*Output 4.1.3:*

Function to construct pts variable, which is used as the basis for most functions. The triple-quoted section was the original method for timing the queries.

**def** **GetPoints(**start**,** end**):**

'''Generates route. Returns url to HTML page to load instructions. Accepts LatLng Object as arguments'''

routes **=** GetDirectionsQueries**(**start**,**end**)**

'''t2 = time.time()

responses = []

counter = 0

for i in routes:

if counter\*0.2 + t2 < time.time(): #check if 0.2 seconds has elapsed since last query made

responses.append(eval(urlopen(i).read())) #get data from url, parse, and add to variable

print 'Done: '+i

else:

print 'Waiting...'

time.sleep(t2 + 0.2 - time.time())

try:

responses.append(eval(urlopen(i).read()))

except IOError:

print 'Query Failed. Try again later.'

print 'Done: '+i

counter += 1

t3 = time.time()

print 'Get Directions from Google Directions API: ' + str(t3-t2)'''

responses **=** map**(**Director**.**directions**,** routes**)**

'''Finished here on 4/26/2015'''

t4 **=** time**.**time**()**

lats**,** lngs **=** **[],** **[]** #used to calculate Bounds of all points

pts\_start **=** **[]** #list of Pt objects

lats\_append**,** lngs\_append **=** lats**.**append**,** lngs**.**append #used to reduce number of . references

pts\_start\_append **=** pts\_start**.**append

**for** response **in** responses**:** #iterate through all the data

**for** leg **in** response**[**'routes'**][**0**][**'legs'**]:**

**for** step **in** leg**[**'steps'**]:**

p **=** Polyline**.**decode**(**step**[**'polyline'**][**'points'**])** #retrieve data from all steps attributes' polylines

speed **=** 35 **\*** MPHtoMS #default speed value

roadname **=** util**.**roadFormat**(**step**[**'html\_instructions'**])**

**try:**

speed **=** math**.**ceil**(**float**(**step**[**'distance'**][**'value'**])** **/** step**[**'duration'**][**'value'**]** **\*** **(**5**\***MPHtoMS**))** **/** **(**5**\***MPHtoMS**)** #gets speed, rounds up 5 mph

**except** ZeroDivisionError**:**

**None**

lats\_append**(**p**[**0**][**0**])**

lngs\_append**(**p**[**0**][**1**])**

pts\_start\_append**((**LatLng**(**p**[**0**][**0**],**p**[**0**][**1**]),**speed**,**roadname**,True))**

**for** pt **in** p**[**1**:-**1**]:**

lats\_append**(**pt**[**0**])**

lngs\_append**(**pt**[**1**])**

pts\_start\_append**((**LatLng**(**pt**[**0**],**pt**[**1**]),**speed**,**roadname**,False))**

final **=** Polyline**.**decode**(**responses**[-**1**][**'routes'**][**0**][**'legs'**][-**1**][**'steps'**][-**1**][**'polyline'**][**'points'**])[-**1**]**

lats\_append**(**final**[**0**])**

lngs\_append**(**final**[**1**])**

pts\_start\_append**((**LatLng**(**final**[**0**],**final**[**1**]),**0**,True))**

bounds **=** Bounds**(**max**(**lats**),**min**(**lats**),**max**(**lngs**),**min**(**lngs**))** #make bounds object to get Overpass data in this range

pts **=** map**(lambda** a**:** Pt**(**a**[**1**][**0**],** a**[**0**],** a**[**1**][**1**],** a**[**1**][**2**]),** enumerate**(**pts\_start**))** #make pts array with enumerate (index needed as argument)

**for** i**,** index **in** zip**(**pts**[**1**:-**1**],** range**(**1**,**len**(**pts**)-**1**)):**

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

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

pts**[**0**].**next **=** pts**[**1**]**

pts**[-**1**].**last **=** pts**[-**2**]**

t5 **=** time**.**time**()**

**print** 'Generate pts list: ' **+** str**(**t5**-**t4**)**

'''Finished here 4/29/15 after 1.25 hours'''

**return** pts**,** bounds

*Code 4.1.4:*

Code to generate a list of pts that share the same coordinates

**def** **References(**pts**):**

'''Makes a list of lists of indexes in pts of pts that have the same coordinates'''

strings **=** map**(lambda** a**:** a**.**string**,** pts**)**

points **=** **{}**

**for** index**,** i **in** enumerate**(**strings**):**

**if** **not** i **in** points**:**

points**[**i**]** **=** **[]**

points**[**i**].**append**(**index**)**

**for** i **in** pts**:**

i**.**references **=** points**[**i**.**string**]**

**return** points**.**values**()**

*Code 4.1.5:*

**def** **Overpass(**pts**,** references**,** bounds**):**

#make url

beginning **=** 'http://overpass-api.de/api/interpreter?data='

query **=** quote**(**'node["highway"="traffic\_signals"]'**+**bounds**.**toOverpassString**()+**';out;'**)**

data **=** **None**

**while** data **==** **None:** #get Overpass API data until successful

**try:** data **=** urlopen**(**beginning **+** query**).**read**()**

**except:** **print** 'Overpass Failed. Trying Again.'

data **=** ET**.**fromstring**(**data**)**

#iterate through nodes, and make some temporary data objects for them to speed things up

**for** node **in** data**.**findall**(**'node'**):**

lat**,** lng **=** float**(**node**.**attrib**[**'lat'**]),** float**(**node**.**attrib**[**'lon'**])**

pt **=** LatLng**(**lat**,** lng**)**

closest**,** dist **=** **None,** 1000

**for** ref **in** references**:**

**if** 0.001 **>** pts**[**ref**[**0**]].**pt**.**lat **-** lat **>** **-**0.001 **and** 0.001 **>** pts**[**ref**[**0**]].**pt**.**lng **-** lng **>** **-**0.001**:** #in reasonable range

**if** pt**.**distanceTo**(**pts**[**ref**[**0**]].**pt**)** **<** dist**:** #is closest so far

#if so, update information

closest **=** ref

dist **=** pt**.**distanceTo**(**pts**[**ref**[**0**]].**pt**)**

**if** closest **!=** **None:** #sometimes pts don't match

**for** ref **in** closest**:** #assign isStoplight to true for all references for closest point

pts**[**ref**].**isStoplight **=** **True**

#update stopsign

**for** ref **in** references**:**

**for** index **in** ref**:**

**if** pts**[**index**].**isStopsign**:**

**for** index **in** ref**:**

pts**[**index**].**isStopsign **=** **True**

**break**

**return** pts

*Code 4.1.6:*

Makes a list of indexes of points that could be intersections. The triple-quoted sections were old methods that did not work.

**def** **OriginalIntersections(**pts**):**

'''Get original pts that can be intersections. Returns straight list of indexes.'''

'''ptcount = [pts[0].references]

indexes = [0]

ptcount\_append = ptcount.append

indexes\_append = indexes.append

frequencies = map(lambda a: len(a.references), pts)

#add possible intersections, all occurances and duplicates

for index, i in enumerate(frequencies[2:-2]):

if i > frequencies[index-2] and i > frequencies[index+2]:

ptcount\_append(pts[index].references)

indexes\_append(index)

ptcount\_append(pts[-1].references)

indexes\_append(len(pts)-1)

print len(ptcount), ptcount[:10]

#strip duplicates from the array

ptcount = util.uniquify(ptcount)

#get indexes in pts of possible intersections

indexes = reduce(list.\_\_add\_\_, (zip([index]\*len(mi),mi) for index, mi in enumerate(ptcount))) #TypeError: reduce() of empty sequence with no initial value

#filter duplicates

new\_ptcount = []

new\_ptcount\_append = new\_ptcount.append

for i in enumerate(ptcount):

if ptcount.indexof(i[1]) == i[0]:

new\_ptcount\_append(i[1])

return ptcount, indexes'''

'''#data structure: pt\_index, string\_of\_coords, references, intersection\_id

ptcount = [(0, pts[0].pt.toStringURL(), pts[0].references, 0)]

ptcount\_append = ptcount.append

frequencies = map(lambda a: len(a.references), pts)

counter = 1

for index, i in enumerate(frequencies[2:-2]):

if frequencies[index] > frequencies[index-2] or frequencies[index] > frequencies[index+2]: #stands out in frequency, probable intersection

ptcount\_append((index, pts[index].pt.toStringURL(), pts[index].references, counter))

counter += 1

ptcount\_append((len(pts)-1, pts[-1].pt.toStringURL(), pts[-1].references, counter))

return ptcount'''

freqs **=** map**(lambda** a**:** len**(**a**.**references**),** pts**)**

indexes **=** **[**0**]** **+** filter**(lambda** a**:** freqs**[**a**]>**freqs**[**a**-**1**]** **or** freqs**[**a**]>**freqs**[**a**+**1**],** range**(**1**,**len**(**freqs**)-**1**))**

**if** len**(**pts**[-**1**].**references**)** **>** 1**:** indexes **+=** **[**len**(**pts**)-**1**]**

**else:** indexes **+=** **[**len**(**pts**)-**2**]**

all\_indexes **=** map**(lambda** a**:** pts**[**a**].**references**,** indexes**)**

**return** sorted**(**util**.**uniquify**(**util**.**flatten**(**all\_indexes**)))**

*Code 4.1.7:*

Joins the output from code 4.1.6 into a dictionary of point coordinates as keys, each with another similar dictionary as their value, equating a connecting point to an index.

**def** **IntersectionsJoin(**indexes**):**

'''Requires sorted input. Joins the indexes given by OriginalIntersections or IntersectionsTrim into a dictionary using the intersection latlng strings as keys.'''

'''inter = {}

for pt, string, ref, index in ind[:-1]:

if not string in inter:

inter[string] = {}

if not ind[index+1][1] in inter[string]:

inter[string][ind[index+1][1]] = ind[index+1]

return inter'''

inter **=** **{}**

zipped **=** zip**(**indexes**,** map**(lambda** a**:** pts**[**a**].**pt**.**toStringURL**(),** indexes**),** range**(**len**(**indexes**)))**

**for** index**,** string**,** i **in** zipped**[:-**1**]:**

**if** **not** string **in** inter**:**

inter**[**string**]** **=** TempIntersection**({},** index **==** 0**)**

**if** **not** zipped**[**i**+**1**][**1**]** **in** inter**[**string**]:**

inter**[**string**][**zipped**[**i**+**1**][**1**]]** **=** **[]**

inter**[**string**][**zipped**[**i**+**1**][**1**]]** **+=** **[(**index**,** zipped**[**i**+**1**][**0**])]**

#add end to inter

inter**[**zipped**[-**1**][**1**]].**isEnd **=** **True**

**return** inter

*Code 4.1.8:*

Trims intersections that do not have enough connections to be considered intersections. This was not actually used, since it removed too many intersections and made it impossible to find a route from start to end through the intersections.

**def** **IntersectionsTrim(**inter**):**

'''Removes intersections that do not have more than 1 connection (since they are redundant).

Possibly will do more later, like searching for loops.

Returns straight list of indexes.'''

new **=** filter**(lambda** a**:** any**((**len**(**a**[**1**])>**2**,** a**[**1**].**isStart**,** a**[**1**].**isEnd**)),** inter**.**items**())** #has more than two intersections

series **=** **[]**

series\_append **=** series**.**append

**for** key**,** value **in** new**:**

**for** intersection **in** value**.**values**():**

'''tup = intersection[1:4]

for pt in intersection[2]:

series\_append(tuple([pt])+)'''

series **+=** map**(lambda** a**:** a**[**0**],** intersection**)**

**return** sorted**(**util**.**uniquify**(**series**))**

*Code 4.1.9:*

**class** **Path(**list**):**

**def** **\_\_init\_\_(**self**,** pts**,** vehicle**):**

list**.**\_\_init\_\_**(**self**,** map**(**copy**,** pts**))**

**for** i**,** j **in** zip**(**pts**[:-**1**],** pts**[**1**:]):**

i**.**next**,** j**.**next **=** j**,** i #update next/last for solve()

self**.**energy**,** self**.**distance**,** self**.**time**,** self**.**penalties **=** self**.**solve**(**vehicle**)**

**def** **getInstructions(**self**,** vehicle**):**

self**.**instructions **=** **[]**

slice\_start **=** 0

slice\_end **=** 0

**for** pt **in** self**[**0**:-**2**]:**

slice\_end **+=** 1

**if** pt**.**roadname **!=** pt**.**next**.**roadname **or** pt**.**next**.**isStoplight**:**

self**.**instructions**.**append**(**

Instruction**(**

self**[:**slice\_end**],**

self**[**0**].**pt**.**bearing**(**self**[**1**].**pt**),**

'start'**,**

vehicle

**)**

**)**

slice\_start **=** slice\_end

**break**

**for** pt **in** self**[**slice\_start**:-**2**]:**

slice\_end **+=** 1

**if** pt**.**roadname **!=** pt**.**next**.**roadname **or** pt**.**next**.**isStoplight**:**

feature **=** 'turn'

**if** self**[**slice\_start**].**temp\_speed **==** 0**:** feature **=** 'stopsign'

**if** self**[**slice\_start**].**isStoplight**:** feature **=** 'stoplight'

self**.**instructions**.**append**(**

Instruction**(**

self**[**slice\_start**:**slice\_end**],**

self**[**slice\_start**].**pt**.**directionAngle**(**self**[**slice\_start**-**1**].**pt**,** self**[**slice\_start**+**1**].**pt**),**

feature**,**

vehicle

**)**

**)**

slice\_start **=** slice\_end

feature **=** 'turn'

**if** self**[**slice\_start**].**temp\_speed **==** 0**:** feature **=** 'stopsign'

**if** self**[**slice\_start**].**isStoplight**:** feature **=** 'stoplight'

self**.**instructions**.**append**(**

Instruction**(**

self**[**slice\_start**:],**

self**[**slice\_start**].**pt**.**directionAngle**(**self**[**slice\_start**-**1**].**pt**,** self**[**slice\_start**+**1**].**pt**),**

feature**,**

vehicle

**)**

**)**

**def** **solve(**self**,** vehicle**):**

'''Changes Pt.temp\_speed based on turn angle and traffic features'''

MPH15 **=** 6.7056 #15 mph in m/s

MASS **=** vehicle**.**mass

penalties **=** 0

energy **=** 0

distance **=** 0

time **=** 0

#manipulate speeds

self**[**0**].**temp\_speed **=** 0

self**[**0**].**features **=** **(**'start'**)**

self**[-**1**].**temp\_speed **=** 0

self**[-**1**].**features **=** **(**'end'**)**

**for** pt **in** self**[**1**:-**1**]:**

**if** **not** pt**.**isStoplight**:**

**if** **not** pt**.**isStopsign**:**

pt**.**temp\_speed **=** pt**.**speed #reset temp\_speed to original speed

**else:**

schange **=** pt**.**next**.**speed **-** pt**.**speed #positive: accel, negative: decel, 0: same

angle **=** pt**.**pt**.**directionAngle**(**pt**.**last**.**pt**,** pt**.**next**.**pt**)** #returns angle of turn as 'right', 'left' or 'straight'

**if** angle **==** 'right'**:**

**if** schange **<=** 0**:**

pt**.**temp\_speed **=** MPH15

**else:**

pt**.**temp\_speed **=** 0

penalties **+=** 10

**elif** angle **==** 'left'**:**

**if** schange **==** 0**:**

pt**.**temp\_speed **=** MPH15

**else:**

pt**.**temp\_speed **=** 0

penalties **+=** 15

**else:**

pt**.**temp\_speed **=** pt**.**speed

**else:**

pt**.**temp\_speed **=** 0

penalties **+=** 20

energy **+=** max**(**0**,** pt**.**last**.**getEnergyNext**(**vehicle**)** **+** MASS**\*(**pt**.**temp\_speed**-**pt**.**last**.**temp\_speed**)\*\***2**)** #changed for loop compatibility

dist **=** pt**.**last**.**pt**.**distanceTo**(**pt**.**pt**)**

distance **+=** dist

low**,** high **=** min**(**pt**.**last**.**temp\_speed**,** pt**.**temp\_speed**),** max**(**pt**.**last**.**temp\_speed**,** pt**.**temp\_speed**)**

time\_to\_accel **=** **(**high **-** low**)** **/** ACCELERATIONRATE

dist\_to\_accel **=** low**\***time\_to\_accel **+** 0.5**\***ACCELERATIONRATE**\***time\_to\_accel**\*\***2

time **+=** time\_to\_accel **+** **(**dist**-**dist\_to\_accel**)/**high

**return** energy**,** distance**,** time**,** penalties

**class** **Intersection(**dict**):**

'''Used as nodes in the network'''

**def** **\_\_init\_\_(**self**,** references**,** pt**,** isStart**=False,** isEnd**=False):**

'''References is a list of points; connections are a list of Connections; pt is a LatLng'''

dict**.**\_\_init\_\_**(**self**,{})**

self**.**references **=** references

self**.**pt **=** pt

self**.**isStart **=** isStart

self**.**isEnd **=** isEnd

self**.**key **=** pt**.**toStringURL**()**

*Code 4.1.10:*

Makes a dictionary of Intersections objects (with embedded Connections) from the dictionary produced in code 4.1.7.

**def** **IntersectionsBuild(**inter**,** pts**):**

'''Makes Intersections objects (with Connections) in a dictionary.'''

intersections **=** **{}**

ptslast **=** len**(**pts**)-**1

**if** len**(**pts**[-**1**].**references**)** **==** 1**:** ptslast **=** len**(**pts**)-**2

#makes dictionary of intersections (using string of LatLng as key) first

**for** key**,** value **in** inter**.**items**():**

pt **=** pts**[**value**.**values**()[**0**][**0**][**0**]]**

intersections**[**key**]** **=** Intersection**(**

map**(lambda** a**:** pts**[**a**],** pt**.**references**),**

pt**.**pt**,**

0 **in** pt**.**references**,**

ptslast **in** pt**.**references

**)**

#uses intersections to append connections to other intersections

**for** index**,** ints **in** inter**.**items**():**

**for** cindex**,** con **in** ints**.**items**():**

**if** index **!=** cindex**:**

con **=** sorted**(**con**,** key**=lambda** a**:** a**[**1**]-**a**[**0**])** #take route with least number of points (if there are different versions)

intersections**[**index**][**cindex**]** **=** Connection**(**pts**[**con**[**0**][**0**]:**con**[**0**][**1**]+**1**],** intersections**[**index**],** intersections**[**cindex**])**

**return** intersections

*Code 4.1.11:*

This is a function that serves as a shell to the recursive function Branch. It begins at the start intersection (which has an attribute marking it as such from previous functions, as well as end), and works through each intersection’s connections, going to the next intersection, until it reaches the end intersection. At that point, it saves the route’s intersection indexes to a list, and it continues on. There is a distance limit built in, which is twice the linear distance between the start intersection and end intersection. This is because it is unlikely a very long route will be an optimal route, and needing to calculate literally every possible route wastes time. Additionally, it cannot go to the same intersection twice, since by definition a route that retraces ground twice is always less efficient than once.

**def** **OptimalDistance(**intersections**,** factor**=**2**):**

'''Intersections as taken from Intersections() and start/end as keys are taken to get candidate routes based on distance.

Factor is multiplier of linear distance from start to end.'''

**def** **Branch(**previous**,** current**,** distance**):**

'''Recursive function to build up routes. Previous is list of indexes, current is an Intersection.'''

**for** key**,** connection **in** current**.**items**():** #iterate through connections

**if** connection**.**end**.**key **not** **in** previous**:** #not already arrived at

new\_distance **=** distance **+** connection**.**distance

**if** connection**.**end**.**isEnd**:** #arrives at the end

solutions\_append**((**previous **+** **[**current**.**key**,** connection**.**end**.**key**],** new\_distance**))**

**elif** distance **+** connection**.**end**.**pt**.**distanceTo**(**current**.**pt**)** **<** max\_dist**:** #is still under distance limit

Branch**(**previous **+** **[**current**.**key**],** connection**.**end**,** new\_distance**)**

#if no possible options, exits this recurrance of the function

t0 **=** time**.**time**()**

start **=** filter**(lambda** a**:** a**.**isStart**,** intersections**.**values**())[**0**]**

end **=** filter**(lambda** a**:** a**.**isEnd**,** intersections**.**values**())[**0**]**

max\_dist **=** start**.**pt**.**distanceTo**(**end**.**pt**)** **\*** factor

solutions **=** **[]**

solutions\_append **=** solutions**.**append

#format = ([keys], distance (meters))

Branch**([],** start**,** 0**)**

t1 **=** time**.**time**()**

**print** t1 **-** t0

**return** solutions

*Code 4.1.12:*

Runs the functions in sequence from codes 4.1.6 to 4.1.10. It also makes a list of valid intersections that are reduced from the recursive function that produces routes from intersections (4.1.11).

**def** **Intersections(**pts**):**

'''Returns a dictionary of Intersections with Connections, with strings as keys.'''

intind **=** OriginalIntersections**(**pts**)**

net **=** IntersectionsJoin**(**intind**)**

trimmed **=** IntersectionsTrim**(**net**)**

rejoined **=** IntersectionsJoin**(**trimmed**)**

intersections **=** IntersectionsBuild**(**rejoined**,** pts**)**

intsInRange **=** util**.**uniquify**(**util**.**flatten**(**map**(lambda** a**:** a**[**0**],** OptimalDistance**(**intersections**))))** #makes list of intersections within reasonable distance to start/end

ultimate\_trim **=** sorted**(**util**.**flatten**(**map**(lambda** a**:** intersections**[**a**].**references**[**0**].**references**,** intsInRange**)))**

**return** IntersectionsBuild**(**IntersectionsJoin**(**ultimate\_trim**),** pts**)**

*Code 4.1.12:*

Makes a list of references (from code 4.1.3) that are included in the routes returned by code 4.1.11. This is to cut down on the amount of elevation data needed, by only collecting from routes that will be considered.

**def** **ValidReferences(**intersections**):**

refs **=** **[]**

**for** inter **in** intersections**.**values**():**

**for** con **in** inter**.**values**():**

refs **+=** map**(lambda** a**:** a**.**references**,** con**)**

**return** util**.**uniquifyTuples**(**refs**)**

*Code 4.1.13:*

These three functions calculate the energy needed to traverse a connection, and then use that information to make a rough estimate of the amount of energy to traverse entire routes. This is applied to all routes. Finally, the last function accesses a Path method to get the detailed energy of a select number of routes, since this is a very complex function.

**def** **ComputeEnergy(**intersections**,** vehicle**):**

**for** inter **in** intersections**.**values**():**

**for** con **in** inter**.**values**():**

con**.**energy **=** con**.**Energy**(**vehicle**)**

**return** **None**

**def** **OptimalEnergyInitial(**intersections**,** routes**):**

'''Get optimal route using pre-generated routes.'''

#get list of most promising routes

output **=** **[]**

**for** route**,** distance **in** routes**:**

total **=** 0

**for** index**,** inter **in** enumerate**(**route**[:-**1**]):**

total **+=** intersections**[**inter**][**route**[**index**+**1**]].**energy

output**.**append**((**route**,** total**))**

output **=** sorted**(**output**,** key**=lambda** a**:** a**[**1**])**

**return** output**[:**min**(**len**(**output**),** 10**)]**

**def** **OptimalEnergyDetailed(**intersections**,** routes**,** vehicle**):**

'''Calculates more in-depth forces for each route'''

#generate single path for each route

new\_routes **=** **[]**

**for** route**,** energy **in** routes**:**

#compile path

path **=** **[]**

**for** index**,** inter **in** enumerate**(**route**[:-**1**]):**

path **+=** intersections**[**inter**][**route**[**index**+**1**]].**path**[:-**1**]** #leave out last in each path, since duplicate of next path

path **+=** **[**intersections**[**route**[-**2**]][**route**[-**1**]].**path**[-**1**]]**

new\_routes**.**append**(**Path**(**path**,** vehicle**))**

**return** new\_routes

*Output 4.1.14:*

This is an example of a query class. The query function is contained in a class to both act as a shell for the recursive query function (it calls itself again if it fails), and also to control its timing. There are similar functions for Elevation, Directions and Overpass APIs.

**class** **Geocoder(**object**):**

**def** **\_\_init\_\_(**self**):**

self**.**next\_query\_time **=** time**.**time**()**

**def** **geocode(**self**,** address**):**

'''Returns LatLng of Geocoding API'''

url **=** 'https://maps.googleapis.com/maps/api/geocode/json?address='**+**address**.**replace**(**' '**,**'+'**,**len**(**address**))+**'&key=AIzaSyCnHT7IpJu0O7n-apLNW8iKkW\_rTIuANuE'

current\_time **=** time**.**time**()**

**if** current\_time **<** self**.**next\_query\_time**:** #before earliest possible query time

time**.**sleep**(**self**.**next\_query\_time **-** current\_time**)** #wait until next query time

**try:**

data **=** eval**(**urlopen**(**url**).**read**())**

self**.**next\_query\_time **=** current\_time **+** 0.2

**if** data**[**'status'**]** **==** 'OK'**:**

**print** 'Geocode:'**,**address

**return** LatLng**(**float**(**data**[**'results'**][**0**][**'geometry'**][**'location'**][**'lat'**]),** float**(**data**[**'results'**][**0**][**'geometry'**][**'location'**][**'lng'**]))**

**else:**

**print** 'Geocode Failed (if).'**,**address

**return** self**.**geocode**(**address**)**

**except:**

**print** 'Geocode Failed (except).'**,**address

**return** self**.**geocode**(**address**)**

Version 4.2

**Summary:**

Added input/output interfacing, so that the application can be called from PHP (popen) or command line, and write real-time console data to HTML and output final instructions to HTML and JSON (like Google Directions API).

Interpolation was finally added, which utilized a method of using the *connections*, not intersections, which were used in the preliminary list of reasonable routes. This decreased the amount of elevation data needed by roughly 70%, when removed connections and a (roughly) 45% decrease in total elevation is considered.

Increased speed in key areas: assigning stoplights to points by chunking and iterating through smaller datasets; finding relevant intersections to get elevation data for by using sets instead of indexing lists; and other small tweaks.

**Notes:**

I ran a test of the algorithm for a 40 mile distance (through IDLE) (routes/test20/console.html, output 4.2.2) and the algorithm took minutes to run. There was also an error that is in Output 4.2.1. It was discovered that some roads did not have a name. This I defaulted to “Unnamed Road” later.

I added in time recording to measure which parts of the algorithm most needed improvement. I began with “6936 Millbridge Road, Clemmons, NC” and “330 Knollwood Street, Winston-Salem, NC”, which is the standard testing route I use (~8 miles). Results as found in test 28/console.html. There were only two functions in the algorithm that took over 0.1 seconds to execute, and those were the functions for assigning road data to pts (and making the points) and the function for iterating through the overpass stoplights and assigning them to pts. This route (visually) takes around 20-22 seconds to run in the program (including queries).

In query.Overpasser.overpass, I split the if statement that decided if the point was in bounds into two if statements, and the time went from 0.158 seconds to 0.120 seconds. The increased cyclomatic complexity was probably offset by making considerably fewer calculations.

I ran tests in IDLE comparing map, for loop comprehensions and for loops with appends. The code was:

**def** **usingMap(**coords**):**

t0 **=** time**.**time**()**

output **=** map**(lambda** a**:** Pt**(**LatLng**(**a**[**1**][**0**],**a**[**1**][**1**]),** a**[**0**],** 35**,** 'stuff'**),** enumerate**(**coords**))**

**return** time**.**time**()-**t0

**def** **usingForEnumerate(**coords**):**

t0 **=** time**.**time**()**

output **=** **[]**

output\_append **=** output**.**append

**for** index**,** **(**lat**,** lng**)** **in** enumerate**(**coords**):**

output\_append**(**Pt**(**LatLng**(**lat**,**lng**),** index**,** 35**,** 'stuff'**))**

**return** time**.**time**()-**t0

**def** **usingForLCE(**coords**):**

t0 **=** time**.**time**()**

output **=** **[**Pt**(**LatLng**(**lat**,**lng**),** index**,** 35**,** 'stuff'**)** **for** index**,** **(**lat**,** lng**)** **in** enumerate**(**coords**)]**

**return** time**.**time**()-**t0

The average results were (after 20 consecutive runs for each):

* Using Map: 0.114
* Using For Loop List Comprehension: 0.113
* Using For Loop: 0.355

I ran the long route again with the timers in place (“6936 Millbridge Road, Clemmons, NC” to “Greensboro, NC”, about 40 miles [64 km]). The biggest one by far was Interpolations, which took 1 minute 51 seconds to complete. This is probably because there 1730 routes, and each one had about 50 intersections. Each intersection has two “in” operations on a list of connections eventually with (probably) a couple hundred items. I can reduce this by using a set instead of a list and organizing the intersection pairs alphabetically (since order does not matter and that will reduce the “in” operations from 2 to 1).

The changes I ended up making were to make elevations\_out and interpolations\_out into sets, since there was a problem with intersection dictionary references if the order was mixed up in any way. I also made pairs a set, and did not worry about having duplicates of reversed intersections. The two aforementioned sets were there to handle that problem, and also obviate the need for costly util.uniquify function calls at the end (since it would have to sort through about 3,000 elements, meaning 9,000,000 comparisons or so). The time for the short route went down from test29 at 0.045 to 0.005 at test33. Also, ValidReferences() was taken out of the script, since it is now obsolete.

On the long route, the initial 111 second time was reduced to 0.095 seconds in test36 (output 4.2.3). However, there was a 400 error returned by Google Elevation API. This did not happen with the same code to the short route.

Another problem is the amount of time it takes to process the Overpass data. On the long route (test34), it took 13.3 seconds for the stoplights to be matched up. This was reduced to 10.2 seconds in test36 (output 4.2.4). In summary, I changed it so that once a close pt was found, the two adjacent points are tested. The closest is chosen for the stoplight, and a break is made.

I made a test of using abs and -a < b-c < a. The test was made using the following functions:

**def** **absval(**a**):**

x **=** **[**random**.**random**()** **for** \_ **in** range**(**a**)]**

t0 **=** time**.**time**()**

y **=** **[**i **for** i **in** x **if** 0.05 **>** abs**(**i**-**0.5**)]**

**return** time**.**time**()-**t0

**def** **comparisons(**a**):**

x **=** **[**random**.**random**()** **for** \_ **in** range**(**a**)]**

t0 **=** time**.**time**()**

y **=** **[**i **for** i **in** x **if** 0.05 **>** 0.5**-**i **>** **-**0.05**]**

**return** time**.**time**()-**t0

* Absolute Value: 0.0021 seconds
* Comparisons: 0.01965 seconds

This was implemented, but made no significant difference.

I had been using the original set of intersections, which in this example, would be 24,917. The combined elev\_refs and inteprol\_refs are 11,375 long, which is less than half. In test42, the time for Overpass matching was 4.44 seconds.

I had a final idea to split up the references into a dictionary of chunks based on coordinates. This is accomplished by using a round function to determine the 0.1 x 0.1 rectangle the coordinate falls in, and add it to the correct dictionary element. This narrows down the possible references to a manageable number (anywhere from near-zero to about 1500 in the long route). It took 0.076 seconds to make the chunks, and 0.127 seconds to assign Overpass data. This is finally satisfactory. (Output 4.2.5)

Finally, there is still the problem with getting invalid responses from Google Maps Elevation API. This was solved by reducing the query length from 300 points to 200 points, as per the suggestion of a user on Stack Overflow.

*Interpolation - Original Idea:*

I deduced that the rules for interpolation must be the following:

* Every point in pts to get elevation for must be within two indexes of another pt to get elevation for
* It is preferable to have distance of 2 whenever possible; minimize 1s
* Only points that have included references may have elevation data gathered for them
* All pts must either have direct elevation data, or be within 1 index of two other pts with direct elevation data.

Therefore, an algorithm must include the following:

* A table that has values for the status of each point
* The start, end, and breaking points must have direct elevation data
* The table should be updated with references constantly (i.e. ANY added interpolation point must be spread out, since it is duplicate data)

After attempting to build a solution, I deemed that this would be too complicated, in light of the idea I had of using the connections on intersections.

*Interpolation – New Idea:*

I had the idea of using the intersections data, since it is already broken up at this point into what I do and don’t need, and it does not have the problem of updating intersections.

There will be an array that will hold key pairs of connections. The reverses will not be permitted to exist. The necessary pairs will be taken from the list of pre-generated routes, so that unneeded connections not used by two used intersections will not be counted. I had not accounted for that in the previous algorithm, because I just thought of that.

I implemented this algorithm the night of 6/8/15, and tested it on 6/9/15, but an error with my program caused an OVER\_QUERY\_LIMIT error (daily limit reached: confirmed) because the queries were unacceptable, but still registered, and caught in an infinite loop on the server. On 6/10/15, I fixed this problem by making each query 300 pts long, instead of that while loop.

This approach works. I ran two tests. The first was my visual inspection of the first forty points: all of them made sense and the interpolations worked fine. I then tested to make sure all the points in the first element of recalc had an elevation: they did. I repeated this test with pts, and it returned false.

This implementation was successful. Between the two new developments (only connections needed, every other points), the number of queries for my test route was reduced from 17 to 5, which translates to a huge time advantage.

On 6/20/15, I added in re-query limits on all queries to make sure they do not accidentally fire continuously. I actually did that the day before, and I had no way of stopping devtano.com from executing the script in an infinite loop. Thankfully, the server terminated the script after x amount of time, but it would have gone through my query limits had I not disabled my API keys temporarily.

**Outputs:**

*Output 4.2.1:*

Traceback (most recent call last):

File "D:\TSA\devtano.com\software\eco\python\main.py", line 449, in <module>

route.getInstructions(vehicle)

File "D:\TSA\devtano.com\software\eco\python\classes.py", line 80, in getInstructions

vehicle

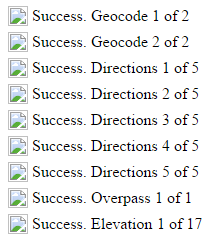
File "D:\TSA\devtano.com\software\eco\python\classes.py", line 39, in \_\_init\_\_

self.string = 'Continue straight onto ' + path[0].roadname + ' and continue for ' + util.mileFormat(meters=self.distance)

TypeError: cannot concatenate 'str' and 'NoneType' objects

*Output 4.2.2:*

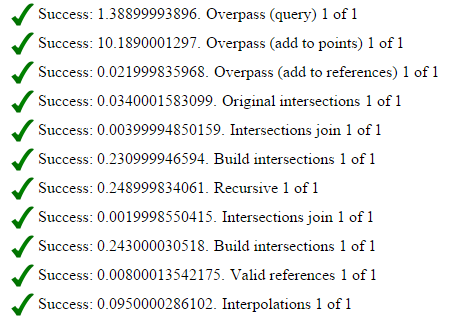
Sample output from the first iteration of the console (without timings). Images appear in the actual application.

**

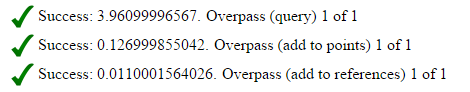
*Output 4.2.3:*

**

*Output 4.2.4:*

**

*Output 4.2.5:*

**

**New Code:**

*Code 4.2.1:*

**class** **Console(**object**):**

new\_text **=** '''

<div class="output %s">

<table>

<tr>

<td class="icon-td">

<div class="icon">

<img src="images/%s.png">

</div>

</td>

<td class="status-td">

<div class="status %s">

%s.

</div>

</td>

<td class="type-td">

<div class="type">

%s

</div>

</td>

<td class="outof-td">

<div class="outof">

%i of %i

</div>

</td>

</tr>

</table>

</div>

'''

**def** **\_\_init\_\_(**self**,** path**):**

self**.**path **=** path

**def** **add(**self**,** event**,** num**=**1**,** outof**=**1**,** success**=True,** error**=**''**):**

event\_class**,** img\_src**,** status\_class**,** status\_text**,** type\_text **=** **None,** **None,** **None,** **None,** **None**

event\_class **=** event**.**lower**()**

type\_text **=** event**.**capitalize**()**

**if** success**:**

img\_src **=** 'check'

status\_class **=** 'success'

status\_text **=** 'Success' **+** error

**else:**

img\_src **=** 'failed'

status\_class **=** 'error'

status\_text **=** 'Error: ' **+** error

f **=** file**(**self**.**path**,**'a'**)**

f**.**write**(**self**.**new\_text **%** **(**event\_class**,** img\_src**,** status\_class**,** status\_text**,** type\_text**,** num**,** outof**))**

f**.**close**()**

*Code 4.2.2:*

Saves JSON data of routes to a file, as well as the default Google Maps route. Similar code exists for HTML format to be displayed directly. This is supposed to be used in scripts.

**def** **JSON(**path**,** routes**,** recom**):**

'''Save JSON data derived from routes to file "path"'''

#prepare object

out **=** **{**

'routes'**:[],**

'status'**:**'OK'**,**

'recommended'**:{},**

'savings'**:[]**

**}**

**for** route **in** routes**:**

steps **=** **[]**

**for** ins **in** route**.**instructions**:**

steps**.**append**({**

'feature'**:**ins**.**feature**,**

'command'**:**ins**.**command**,**

'polyline'**:**ins**.**polyline**,**

'instruction'**:**ins**.**string**,**

'energy\_used'**:**ins**.**energy\_used\_string**,**

'energy'**:**int**(**ins**.**energy**),**

'distance'**:**int**(**ins**.**distance**),**

'start'**:{**

'lat'**:**ins**.**start**.**pt**.**lat**,**

'lng'**:**ins**.**start**.**pt**.**lng

**},**

'end'**:{**

'lat'**:**ins**.**end**.**pt**.**lat**,**

'lng'**:**ins**.**end**.**pt**.**lng

**}**

**})**

robj **=** **{**

'steps'**:**steps**,**

'distance'**:{**

'meters'**:**int**(**route**.**distance**),**

'miles'**:**round**(**route**.**distance**/**METERSPERMILE**,**1**)**

**},**

'energy'**:{**

'joules'**:**int**(**route**.**energy**),**

'gasoline'**:{**

'liters'**:**round**(**route**.**energy**/**JPERLITERGASOLINE**\***4**,**2**),**

'gallons'**:**round**(**route**.**energy**/**JPERGALLONGASOLINE**\***4**,**2**)**

**}**

**},**

'time'**:{**

'seconds'**:**int**(**route**.**time**),**

'minutes'**:**round**(**route**.**time**/**60**)**

**},**

'polyline'**:**route**.**polyline

**}**

out**[**'routes'**].**append**(**robj**)**

out**[**'recom'**]** **=** **{**

'distance'**:{**

'meters'**:**int**(**recom**.**distance**),**

'miles'**:**round**(**recom**.**distance**/**METERSPERMILE**,**1**)**

**},**

'energy'**:{**

'joules'**:**int**(**recom**.**energy**),**

'gasoline'**:{**

'liters'**:**round**(**recom**.**energy**/**JPERLITERGASOLINE**\***4**,**2**),**

'gallons'**:**round**(**recom**.**energy**/**JPERGALLONGASOLINE**\***4**,**2**)**

**}**

**},**

'time'**:{**

'seconds'**:**int**(**recom**.**time**),**

'minutes'**:**round**(**recom**.**time**/**60**)**

**},**

'polyline'**:**recom**.**polyline

**}**

recomJ **=** int**(**recom**.**energy**)**

out**[**'savings'**]** **=** **[{**

'route'**:**index**,**

'joules'**:**recomJ**-**int**(**route**.**energy**),**

'gasoline'**:{**

'liters'**:**round**((**recomJ**-**route**.**energy**)/**JPERLITERGASOLINE**\***4**,**2**),**

'gallons'**:**round**((**recomJ**-**route**.**energy**)/**JPERGALLONGASOLINE**\***4**,**2**)**

**}}** **for** index**,** route **in** enumerate**(**routes**)]**

**with** open**(**path**,** 'w'**)** **as** outfile**:**

json**.**dump**(**out**,** outfile**)**

*Code 4.2.3:*

New Overpass function after optimizations made:

* Decreased cyclomatic complexity
* Absolute value instead of comparisons
* Use intersection references
* Split data into chunks, do distributed searching (**def** **chunk**)

**class** **Overpasser(**object**):**

**def** **\_\_init\_\_(**self**):**

**None**

**def** **chunk(**self**,** pts**,** references**):**

t0 **=** time**.**time**()**

chunks **=** **{}**

**for** ref **in** references**:**

chunk **=** str**(**round**(**pts**[**ref**[**0**]].**pt**.**lat**,** 1**))** **+** ',' **+** str**(**round**(**pts**[**ref**[**0**]].**pt**.**lng**,** 1**))**

**if** **not** chunk **in** chunks**:**

chunks**[**chunk**]** **=** **[]**

chunks**[**chunk**].**append**(**ref**)**

**print** time**.**time**()-**t0

**return** chunks

**def** **overpass(**self**,** pts**,** references**,** chunks**,** bounds**,** console**):**

#make url

t0 **=** time**.**time**()**

beginning **=** 'http://overpass-api.de/api/interpreter?data='

query **=** quote**(**'node["highway"="traffic\_signals"]'**+**bounds**.**toOverpassString**()+**';out;'**)**

data **=** **None**

**while** data **==** **None:** #get Overpass API data until successful

**try:**

data **=** urlopen**(**beginning **+** query**).**read**()**

console**.**add**(**'overpass'**,** 1**,** 1**)**

**except:** console**.**add**(**'overpass'**,** 1**,** 1**,** **False,** 'Problem with connection'**)**

data **=** ET**.**fromstring**(**data**)**

t1 **=** time**.**time**()**

#iterate through nodes, and make some temporary data objects for them to speed things up

**for** node **in** data**.**findall**(**'node'**):**

lat**,** lng **=** float**(**node**.**attrib**[**'lat'**]),** float**(**node**.**attrib**[**'lon'**])**

chunk **=** str**(**round**(**lat**,** 1**))** **+** ',' **+** str**(**round**(**lng**,** 1**))**

**if** chunk **in** chunks**:**

pt **=** LatLng**(**lat**,** lng**)**

closest **=** **None**

**for** ref **in** chunks**[**chunk**]:**

**if** 0.0005 **>** abs**(**pts**[**ref**[**0**]].**pt**.**lat **-** lat**):**

**if** 0.0005 **>** abs**(**pts**[**ref**[**0**]].**pt**.**lng **-** lng**):** #in reasonable range

d0 **=** pts**[**ref**[**0**]].**pt**.**distanceTo**(**pt**)**

d1 **=** pts**[**min**(**ref**[**0**]+**1**,**len**(**pts**)-**1**)].**pt**.**distanceTo**(**pt**)**

d2 **=** pts**[**max**(**ref**[**0**]-**1**,**0**)].**pt**.**distanceTo**(**pt**)**

**if** d0 **>** d1 **and** d0 **>** d2**:**

closest **=** ref

**elif** d1 **>** d2**:**

closest **=** pts**[**min**(**ref**[**0**]+**1**,**len**(**pts**)-**1**)].**references

**else:**

closest **=** pts**[**max**(**ref**[**0**]-**1**,**0**)].**references

**break**

**if** closest **!=** **None:** #sometimes pts don't match

**for** ref **in** closest**:** #assign isStoplight to true for all references for closest point

pts**[**ref**].**isStoplight **=** **True**

t2 **=** time**.**time**()**

#update stopsign

**for** ref **in** references**:**

**for** index **in** ref**:**

**if** pts**[**index**].**isStopsign**:**

**for** index **in** ref**:**

pts**[**index**].**isStopsign **=** **True**

**break**

t3 **=** time**.**time**()**

console**.**add**(**'Overpass (query)'**,** error**=**': '**+**str**(**t1**-**t0**))**

console**.**add**(**'Overpass (add to points)'**,** error**=**': '**+**str**(**t2**-**t1**))**

console**.**add**(**'Overpass (add to references)'**,** error**=**': '**+**str**(**t3**-**t2**))**

**return** pts

*Code 4.2.4:*

This is the code to get generate interpolations. It makes a set of used connections, and sorts alternating point references to two other sets.

**def** **getInterpolations(**self**,** routes**,** intersections**,** console**):**

t0 **=** time**.**time**()**

pairs **=** set**()**

elevation\_out **=** set**([**tuple**(**inter**.**references**[**0**].**references**)** **for** inter **in** intersections**.**values**()])** #references to get elevation for

interpolate\_out **=** set**()** #references to interpolate for. Format: (references, left\_references, right\_references)

**for** route**,** dist **in** routes**:**

**for** int1**,** int2 **in** zip**(**route**[:-**1**],** route**[**1**:]):**

pairs**.**add**((**int1**,** int2**))**

**for** index**,** con **in** list**(**pairs**):**

path **=** intersections**[**index**][**con**]**

path\_len **=** len**(**path**)**

**for** pt\_index**,** pt **in** enumerate**(**path**[**1**:-**1**]):**

**if** **(**pt\_index **%** 2**):**

elevation\_out**.**add**(**tuple**(**pt**.**references**))**

**else:**

interpolate\_out**.**add**((**tuple**(**pt**.**references**),** tuple**(**path**[**pt\_index**].**references**),** tuple**(**path**[**pt\_index**+**2**].**references**)))**

console**.**add**(**'Interpolations'**,** error**=**': '**+**str**(**time**.**time**()-**t0**))**

**return** list**(**elevation\_out**),** list**(**interpolate\_out**)**

*Code 4.2.5:*

This code in main.py enabled the program to be run from either command line/PHP or directly from IDLE.

**if** len**(**sys**.**argv**)** **<** 2**:**

args **=** **{**

'start'**:**'6936 Millbridge Road, Clemmons, NC'**,**

'end'**:**'330 Knollwood Street, NC'**,**

'mass'**:**1500**,**

'cd'**:**0.35**,**

'area'**:**3.0**,**

'disp'**:**0.002**,**

'routes'**:**10**,**

'id'**:**'../routes/'**+**input**(**'path (remember: string!): '**)**

**}**

routes**,** recom **=** EcoCartographer**(**args**)**

**else:** #called by command line or PHP

args **=** **{**

'start'**:**sys**.**argv**[**1**].**replace**(**'+'**,**' '**),**

'end'**:**sys**.**argv**[**2**].**replace**(**'+'**,**' '**),**

'mass'**:**int**(**sys**.**argv**[**3**]),**

'cd'**:**float**(**sys**.**argv**[**4**]),**

'area'**:**float**(**sys**.**argv**[**5**]),**

'disp'**:**float**(**sys**.**argv**[**6**])/**1000000**,** #input is in cm^3, not m^3

'routes'**:**int**(**sys**.**argv**[**7**]),**

'id'**:**'routes/'**+**sys**.**argv**[**8**]**

**}**

**print** args

routes **=** EcoCartographer**(**args**)**

Website (HTML/CSS)

**Summary:**

Initial pages placed by Akshat, along with some CSS styling stuff, from a template. Daniel made edits to this. Ali entered content. Collin gathered data for the drop-down menu in the input section of the main application page. Daniel re-formatted the CSS for the website to make it more readable and easy to edit. Ali and Daniel wrote a new version of the End-User Documentation. Initially, the pages were in HTML, and had copies of the same headers. They were changed by Daniel to PHP, and they loaded a header and footer from separate master files to increase compatibility and make editing a less strenuous process. Daniel wrote the main application page, with the JavaScript, from scratch. Daniel also spent a lot of time looking for a way to connect HTML to Python, and ended up finding and using PHP.

All file transfers were made to the remote server over FTP. Daniel kept a mirror of the appropriate directory tree on his flash drive.

**Notes:**

*main.php*

So as said before, the first and most important task was to create an interactive interface for the software to be put on. The webpage now makes it easier to input your data into *EcoCartographer.* You can see the vehicle selection interface below, in output 4.3.1.

With each vehicle is a different set of values that are needed to calculate the fuel-efficient routes. Some examples of the different values that are hidden beneath the hood in the code are shown in the image below, in output 4.3.2.

With the old software webpage, when the software was done with its calculations, a popup webpage was brought up with directions from your starting point to your destination. Now, with the new and improved software page, you can specify the number of routes you would like as well as have all of the directions on the sidebar to the right. The new setup can be seen in output 4.3.3.

And with the newer version of the software, as mentioned above, you can have multiple routes to get to where you want so that you have the choice to weigh your options. There is now also the additional feature that allows you to take the route that Google Maps plots.

In addition to this new feature, there is also the new “Query ID”. You receive an automatically generated ID that is refreshed every time the webpage is. With this ID, you can find your search results without actually having to input all the data again. There is a link provided where you can input your “Query ID” and put it into your web browser and receive your results.

There was an attempt made to further compartmentalize the div that was the sidebar. Initially, the plan was to have only the console and output have scroll, but there was never a solution to making them shrink to fill the remaining space in the div. That meant the solution was to make the entire sidebar scrollable.

*Other pages*

The first additional webpage was just to help you use the software. But from that, the website expanded into a webpage where you can access the software page from, a quick reference guide for how to use EcoCartographer, an End-User Documentation, a statistics page, and lastly a contact page.

Additionally, another major change was made in the usage of PHP for headers and footers. Instead of having copies of the same header/footer on each page, and needing to update each independently, a header.php and footer.php file was created, and these were then imported into the other PHP files that made up the content of the website. Initially, the plan was to do a similar thing with iframes, but PHP was deemed a more acceptable and elegant solution.

There was difficulty with getting the images to display properly in the End-User documentation. The issue was that really tall images would take up more vertical space than the corresponding context, and images would bunch up. A question about this was asked on Stack Overflow, which was called [**Align image while keeping height**](http://stackoverflow.com/questions/30982827/align-image-while-keeping-height). The answer turned up was to use a CSS clear tag. Daniel even got 25 reputation points from five up votes.

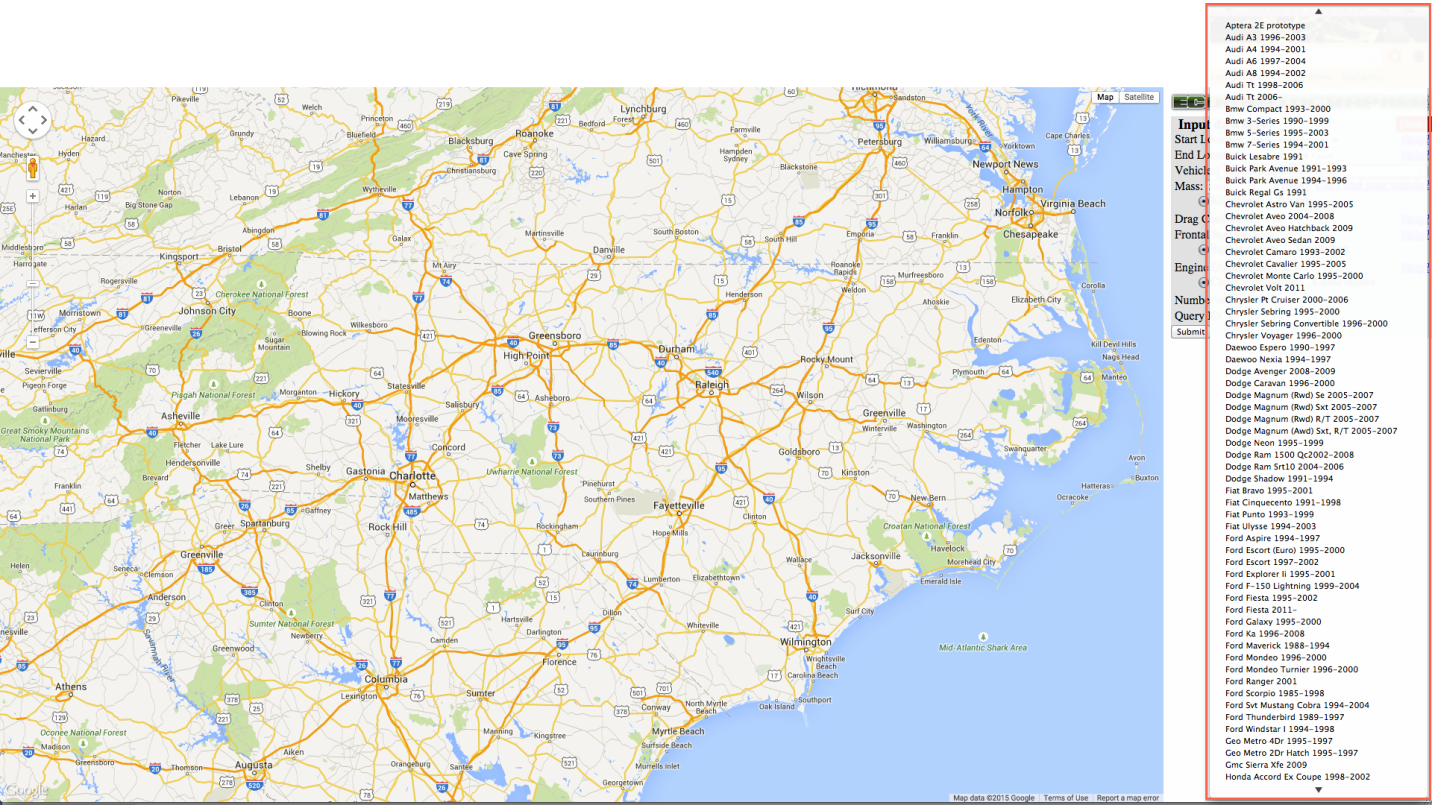
For this webpage, the CSS was not just written on the same page. The CSS was written into the code as a link so as to avoid cluttering up of the page source. This also enabled us to be more organized. The End-User Documentation was transformed from the earlier version into a sleeker design that was more pleasing to the eye. The final version is longer than the previous version of itself. If compared side by side the similarities can be very easily noted and pointed out.

There are significant contrasts between the old End-User Documentation and the new website style. A major difference is that there is a significantly larger amount of information in the final version than the initial page. The color schemes have also changed with the more colorful one from the beginning, and the sleeker and more professional looking design in the final and finished webpage by using more generalized styles.

In the new webpage, there was the introduction of credits that were on the bottom of the page and were omnipresent in every page. There are also statistics that are a new addition to the webpage. Some example statistics are the average money spent on gasoline and carbon dioxide released into the environment.

**Outputs:**

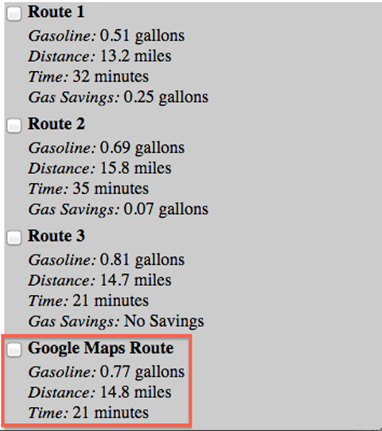
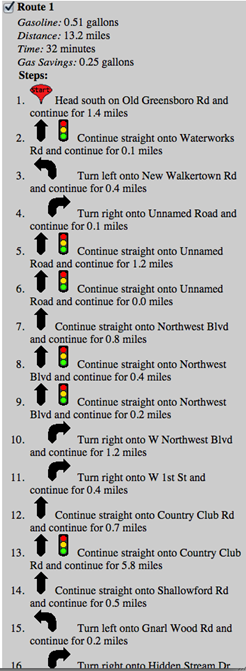
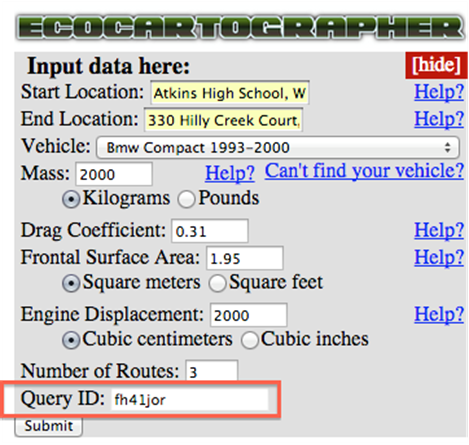
*Output 4.3.1:*



*Output 4.3.2:*

**

*Output 4.3.3:*

**

*Output 4.3.4:*

**

Website (JavaScript/PHP)

**Summary:**

JavaScript was written for main.php in order to load a Google Map (initialize.js), make form elements interact correctly (unit-widget.js), and execute and display queries (main.js). jQuery was used in all three of these.

**Notes:**

The initialize.js was the same old formula as used before to initialize a Google Map.

*Input Form*

The first step to making the new version of the application shell was to make the input boxes, which was done on 5/23/15. The original code can be seen in code 4.4.1 for JavaScript, 4.4.2 for HTML, and 4.4.3 for CSS. This unit widget was designed to incorporate various unit systems, so that the user does not have to know all the specifications in metric. This feature was designed to be independent of the styles of other web pages, and function as a separate entity, though not existing in an iframe.

The JavaScript in the unit-widget was very simplistic, in that it could a) only handle toggles between two predefined unit types, b) not keep a specific value, i.e. 2000cm^3 🡪 122in^3 🡪 1999cm^3 🡪 121in^3 🡪 1982cm^3 🡪 120in^3 🡪 1966cm^3 etc., and c) could not distinguish between unit types upon form submission, so a value representing pounds could be submitted as kilograms.

This was fixed on 6/21/15, taking about 3.5 hours. The new system used jQuery to save a metric value to each input element. Each time the value for that element was updated, the coefficient of the selected unit was used to convert the current input into a metric, which would be stored in the element. A similar process would take place when units changed. This not only solved the problem of lost data, but it also allowed for theoretical extra unit types (like cubic meters, should one find it convenient to do so). Finally, upon form submission, the values and radio buttons changed to metric, so that they were submitted as metric and nothing else.

*PHP*

On Tuesday, 6/2/15, I was looking on Stack Overflow to find ways to execute Python from the browser, not specifically JavaScript. I found a post that included PHP: <http://stackoverflow.com/questions/19735250/running-a-python-script-from-php>.

I looked up examples of PHP and stumbled across popen, which I have used in Python before to get a command line output. I ultimately found this PHP documentation, which was ultimately what was helpful: <http://php.net/manual/en/function.popen.php>.

Finally, I needed to find a way to input arguments to the Python script via the command line. This I found: <http://www.tutorialspoint.com/python/python_command_line_arguments.htm>.

The final PHP code (saved as shell.php):

<html>

<head>

<title>PHP Test</title>

</head>

<body>

<?php

$handle = popen('python test.py hello lorem world ipsum', 'r');

$read = fread($handle, 2096);

echo $read;

pclose($handle);

?>

<div style="height:400px; width:700px; background-color:#ffff00">

Extra Content

</div>

</body>

</html>

The final Python code (saved as test.py):

import sys

variables = sys.argv

for i in variables:

print i

This output exactly what was expected: test.py hello lorem world ipsum. PHP could not be run from my laptop, but it was run from devtano.com after I uploaded the needed files.

I next experimented with adding time.sleep functions to the Python. It did not do what I was expecting, which was to gradually display the outputs. Instead, the page “loaded” until the program finished executing. The idea I had to work around this would be to write the console outputs to a file, and have an Ajax request being constantly fired and constantly updating the screen that way; regardless of extra time to execute file writes, it is ultimately important that the user know the progress rather than get impatient. The final output will be printed to the screen.

On 6/3/2015, I made an output stream to an HTML file from the Python, and integrated it into the script. This included an HTML template for the output.

On 6/4/2015, I started integrating existing documentation into the input-help.html. I plan to make a single documentation page. I also began work on the HTML shell for the application. I asked a question on Stack Overflow about changing the titles on markers. Built-in to the shell is a geocoder that updates a marker position on the map for start and end points, as well as the map orientation. I also have geocoding exception catching. I still need to work on the layout of the page. I plan to have the input box docked on the right side, along with some content beneath that, and have the map be the entire page.

*Shell action*

A desired feature of the application was to show the user where the start and end locations were before a query. The initial solution to this problem was to make a function that retrieved the coordinates of the address(es) entered, and then display them on the map, when the onchange event of either text input was fired. However, this required

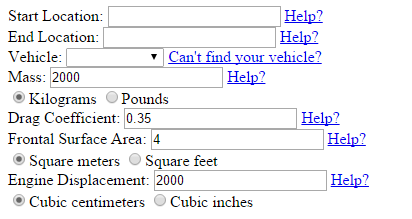
Ajax was used to constantly update a div of the sidebar to the console output from the Python script. When the file output showed up, this cycle stops, and the output is displayed on the map after some JavaScript manipulation and the HTML output is assigned to another div in the sidebar. After some experimentation, it was discovered that the Ajax query needed some time to actually get the data. The final value for the timeout was a second; 300 milliseconds was too little.

One of the design goals for the map output was the ability to show/hide different routes, since there would now be multiple routes in addition to the default Google Maps route. This was done by making a global object containing all the routes, which were objects with prototype show/hide functions that would be triggered by the toggling of a checkmark next to the corresponding directions. This was straight-forward code to right, and it was right in half an hour.

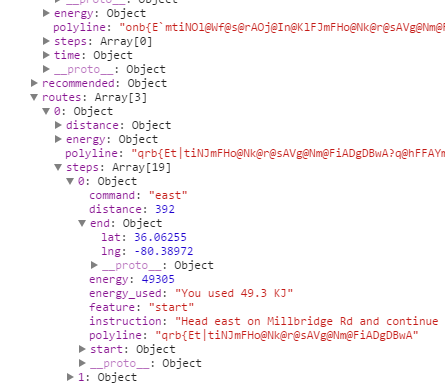
**Outputs:**

*Output 4.4.1:*

Unit-widget, to be used in the input section of the main application page.



*Output 4.4.2:*

**

**Code:**

*Code 4.4.1:*

***function*** massConvert**(**to**){**

***var*** mass **=** document**.**getElementById**(**'mass-input'**);**

***if*(**to**===**'kg'**){**

mass**.**value **=** parseInt**(**mass**.**value **/** 2.20462**);**

**}*else*{**

mass**.**value **=** parseInt**(**mass**.**value **\*** 2.20462**);**

**}**

**}**

***function*** areaConvert**(**to**){**

***var*** area **=** document**.**getElementById**(**'area-input'**);**

***if*(**to**===**'m^2'**){**

area**.**value **=** **(**area**.**value **/** 10.7639**).**toFixed**(**2**);**

**}*else*{**

area**.**value **=** **(**area**.**value **\*** 10.7639**).**toFixed**(**2**);**

**}**

**}**

***function*** dispConvert**(**to**){**

***var*** disp **=** document**.**getElementById**(**'disp-input'**);**

***if*(**to**===**'cm^3'**){**

disp**.**value **=** parseInt**(**disp**.**value **\*** 16.387064**);**

**}*else*{**

disp**.**value **=** parseInt**(**disp**.**value **/** 16.387064**);**

**}**

**}**

*Code 4.4.2:*

<html>

<head>

<script src=*"unit-widget.js"*></script>

<link rel=*"stylesheet"* type=*"text/css"* href=*"unit-widget.css"*>

<body>

<div id=*"unit-widget"*>

<form id=*"main-input"*>

<div id=*"start-div"* class=*"item"*>

**Start Location:**

<input type=*"text"* name=*"start"* id=*"start-input"*>

<span id=*"start-help"* class=*"help"*>

<a href=*"input-help.html#start-input"*>**Help?**</a>

</span>

</div>

<div id=*"end-div"* class=*"item"*>

**End Location:**

<input type=*"text"* name=*"end"* id=*"end-input"*>

<span id=*"end-help"* class=*"help"*>

<a href=*"input-help.html#end-input"*>**Help?**</a>

</span>

</div>

<div id=*"vehicle-div"* class=*"item"*>

**Vehicle:**

<select id=*"vehicle-input"* name=*"vehicle"*>

<option value=*""* id=*"\_\_blank\_\_"*></option>

<option value=*"crv"* id=*"crv"*>**Honda CRV**</option>

</select>

<span id=*"vehicle-help"* class=*"help"*>

<a href=*"input-help.html#vehicle-input"*>**Can't find your vehicle?**</a>

</span>

</div>

<div id=*"mass-div"* class=*"item"*>

**Mass:**

<input type=*"number"* class=*"number"* name=*"mass"* id=*"mass-input"* min=*"50"* step=*"50"* size=*"5"* value=*"2000"*>

<span id=*"mass-help"* class=*"help"*>

<a href=*"input-help.html#mass-input"*>**Help?**</a>

</span>

<div class=*"radio-container"*>

<input type=*"radio"* name=*"mass-unit"* value=*"kg"* onchange=*"massConvert(this.value)"* checked>**Kilograms**

<input type=*"radio"* name=*"mass-unit"* value=*"lb"* onchange=*"massConvert(this.value)"*>**Pounds**

</div>

</div>

<div id=*"drag-div"* class=*"item"*>

**Drag Coefficient:**

<input type=*"number"* class=*"number"* name=*"drag"* id=*"drag-input"* min=*"0"* step=*"0.01"* value=*"0.35"*>

<span id=*"drag-help"* class=*"help"*>

<a href=*"input-help.html#drag-input"*>**Help?**</a>

</span>

</div>

<div id=*"area-div"* class=*"item"*>

**Frontal Surface Area:**

<input type=*"number"* class=*"number"* name=*"area"* id=*"area-input"* min=*"0"* step=*"0.5"* size=*"5"* value=*"4"*>

<span id=*"area-help"* class=*"help"*>

<a href=*"input-help.html#area-input"*>**Help?**</a>

</span>

<div class=*"radio-container"*>

<input type=*"radio"* name=*"area-unit"* value=*"m^2"* onchange=*"areaConvert(this.value)"* checked>**Square meters**

<input type=*"radio"* name=*"area-unit"* value=*"ft^2"* onchange=*"areaConvert(this.value)"*>**Square feet**

</div>

</div>

<div id=*"disp-div"* class=*"item"*>

**Engine Displacement:**

<input type=*"number"* class=*"number"* name=*"disp"* id=*"disp-input"* min=*"0"* step=*"100"* size=*"5"* value=*"2000"*>

<span id=*"disp-help"* class=*"help"*>

<a href=*"input-help.html#disp-input"*>**Help?**</a>

</span>

<div class=*"radio-container"*>

<input type=*"radio"* name=*"disp-unit"* value=*"cm^3"* onchange=*"dispConvert(this.value)"* checked>**Cubic centimeters**

<input type=*"radio"* name=*"disp-unit"* value=*"in^3"* onchange=*"dispConvert(this.value)"*>**Cubic inches**

</div>

</div>

</form>

</div>

</body>

</html>

*Code 4.4.3:*

**#unit-widget****{**

**}**

**#unit-widget****.**item**{**

**margin-bottom:2px;**

**margin-left:5px;**

**}**

**#unit-widget****.**help**{**

**float:right;**

**margin-left:8px;**

**}**

**#unit-widget****.**radio-container**{**

**margin-left:30px;**

**margin-bottom:6px;**

**}**

**#unit-widget****.**number**{**

**width:60px;**

**}**

*Code 4.4.4:*

PHP to execute Python. The arguments are submitted from the form.

<html>

<head>

<title>**Do Not Close - PHP Shell - EcoCartographer**</title>

</head>

<body>

<?php

$handle = popen('python python/main.py' . ' ' . **str\_replace**(' ','+',$\_POST['start']) . ' ' . **str\_replace**(' ','+',$\_POST['end']) . ' ' . $\_POST['mass'] . ' ' . ' ' . $\_POST['drag'] . ' ' . $\_POST['area'] . ' ' . $\_POST['disp'] . ' ' . $\_POST['routes'] . ' ' . $\_POST['id'], 'r');

$read = **fread**($handle, 2096);

pclose($handle);

**echo** "<h1 style='color:#f00; text-align:center'> This query is over, and this tab may be closed now. </h1>";

?>

</body>

</html>

*Code 4.4.5:*

JavaScript to handle the importing of outputs and console.

***function*** cont**(){**

***var*** id **=** $**(**'#id-input'**).**val**();**

***function*** get**(){**

***try*{**

console**.**log**(**'ajax request'**);**

$**.**ajax**({**

url**:**'routes/'**+**id**+**'/console.html'**,**

dataType**:**'text'**,**

success**:**consoleAjaxSuccess**,**

error**:**\_\_blank\_\_**,**

**});**

$**.**ajax**({**

url**:**'routes/'**+**id**+**'/output.json'**,**

dataType**:**'json'**,**

success**:*function*(**data**){**

console**.**log**(**data**);**

resultsAjaxSuccess**(**data**.**routes**,** data**.**recom**);**

setTimeout**(*function*(){**

$**.**ajax**({**

url**:**'routes/'**+**id**+**'/output.html'**,**

dataType**:**'text'**,**

success**:*function*(**output**){**

$**(**'#output'**).**html**(**output**);**

**}**

**});**

**},**2000**);**

**},**

error**:**get**,**

timeout**:**1000

**});**

**}*catch*(**e**){}**

**}**

get**();**

**}**

***function*** \_\_blank\_\_**(){};**

***function*** consoleAjaxSuccess**(**data**){**

$**(**"#console"**).**html**(**data**);**

**}**

*Code 4.4.6:*

This JavaScript writes instructions to the map.

/\*\*\*Class for routes\*\*\*/

***var*** colors **=** **[**'#0f0'**,**'#0ff'**,**'#00f'**,**'#f0f'**,**'#f00'**,**'#ff0'**,**'#0f8'**,**'#08f'**,**'#80f'**,**'#880'**];**

***var*** Route **=** ***function*(**data**,** index**){**

***var*** polyline **=** ***new*** google**.**maps**.**Polyline**({**

path**:**google**.**maps**.**geometry**.**encoding**.**decodePath**(**data**.**polyline**),**

geodesic**:*true*,**

strokeColor**:**colors**[**index**%**10**],**

strokeWeight**:**5

**});**

polyline**.**setMap**(**map**);**

polyline**.**popup **=** ***new*** google**.**maps**.**InfoWindow**({**content**:**"<b>Summary:</b><br><i>Route Number: </i>"**+(**index**+**1**)+**"<br><i>Distance: </i>"**+**data**.**distance**.**miles**+**" miles<br><i>Gasoline: </i>"**+**data**.**energy**.**gasoline**.**gallons**+**" gallons<br><i>Time: </i>"**+**data**.**time**.**minutes**+**" minutes"**});**

google**.**maps**.**event**.**addListener**(**polyline**,** 'click'**,** ***function*(**e**){*this*.**popup**.**setPosition**(**e**.**latLng**);** ***this*.**popup**.**open**(**map**)});**

***var*** markers **=** **[];**

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

***var*** step **=** data**.**steps**[**j**];**

***var*** icon**;**

***if*(**step**.**feature**===**"stoplight"**){**

icon **=** "images/stoplight.png"**;**

**}*else*** ***if*(**step**.**feature**===**"stopsign"**){**

icon **=** "images/stopsign.png"**;**

**}*else*{**

***try*{**

icon **=** "images/"**+**step**.**command**+**".png"**;**

**}*catch*(**e**){**

**}**

**}**

markers**.**push**(*new*** google**.**maps**.**Marker**({**

position**:*new*** google**.**maps**.**LatLng**(**step**.**start**.**lat**,** step**.**start**.**lng**),**

title**:**step**.**instruction**,**

icon**:**icon**,**

map**:**map

**}));**

**}**

***this*.**polyline **=** polyline**;**

***this*.**markers **=** markers**;**

**}**

Route**.*prototype*.**hide **=** ***function*(){**

***this*.**polyline**.**setMap**(**null**);**

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

***this*.**markers**[**i**].**setMap**(**null**);**

**}**

**}**

Route**.*prototype*.**show **=** ***function*(){**

***this*.**polyline**.**setMap**(**map**);**

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

***this*.**markers**[**i**].**setMap**(**map**);**

**}**

**}**

//everything will be contained within routes

//route will be set to routes returned by JSON

//google maps elements will be added on top

***var*** routes**;**

***var*** markers **=** **{};**

***var*** polylines **=** **{};**

***function*** resultsAjaxSuccess**(**data**,** recom**){**

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

polylines**[**i**]** **=** ***new*** Route**(**data**[**i**],** i**);**

**}**

recom**.**steps **=** **[];**

polylines**[-**1**]** **=** ***new*** Route**(**recom**,** 'Google Route'**);**

**}**

*Code 4.4.7:*

The older version of 4.4.6, without the object and prototype functions.

***function*** resultsAjaxSuccess**(**data**,** recom**){**

routes **=** data**;**

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

//set to visible by default

routes**[**i**].**visible **=** ***true*;**

//build polyline

***var*** polyline **=** ***new*** google**.**maps**.**Polyline**({**

path**:**google**.**maps**.**geometry**.**encoding**.**decodePath**(**data**[**i**].**polyline**),**

geodesic**:*true*,**

strokeColor**:**colors**[**i**%**10**],**

strokeWeight**:**5

**});**

polyline**.**setMap**(**map**);**

polyline**.**popup **=** ***new*** google**.**maps**.**InfoWindow**({**content**:**"<b>Summary:</b><br><i>Route Number: </i>"**+(**i**+**1**)+**"<br><i>Distance: </i>"**+**routes**[**i**].**distance**.**miles**+**" miles<br><i>Gasoline: </i>"**+**routes**[**i**].**energy**.**gasoline**.**gallons**+**" gallons<br><i>Time: </i>"**+**routes**[**i**].**time**.**minutes**+**" minutes"**});**

google**.**maps**.**event**.**addListener**(**polyline**,** 'click'**,** ***function*(**e**){*this*.**popup**.**setPosition**(**e**.**latLng**);** ***this*.**popup**.**open**(**map**)});**

//google.maps.event.addListener(polyline, 'mouseout', function(e){this.popup.close()});

//build markers

***var*** markers **=** **[];**

***for*(*var*** j**=**0**;** j**<**data**[**i**].**steps**.**length**;** j**++){**

***var*** step **=** data**[**i**].**steps**[**j**];**

***var*** icon**;**

***if*(**step**.**feature**===**"stoplight"**){**

icon **=** "images/stoplight.png"**;**

**}*else*** ***if*(**step**.**feature**===**"stopsign"**){**

icon **=** "images/stopsign.png"**;**

**}*else*{**

***try*{**

icon **=** "images/"**+**step**.**command**+**".png"**;**

**}*catch*(**e**){**

**}**

**}**

markers**.**push**(*new*** google**.**maps**.**Marker**({**

position**:*new*** google**.**maps**.**LatLng**(**step**.**start**.**lat**,** step**.**start**.**lng**),**

title**:**step**.**instruction**,**

icon**:**icon**,**

map**:**map

**}));**

**}**

//assign data to route

routes**[**i**].**mapPolyline **=** polyline**;**

routes**[**i**].**mapMarkers **=** markers**;**

//route show/hide functions

routes**[**i**].**hide **=** ***function*(){**

***this*.**mapPolyline**.**setMap**(**null**);**

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

***this*.**mapMarkers**[**i**].**setMap**(**null**);**

**}**

**};**

routes**[**i**].**show **=** ***function*(){**

***this*.**mapPolyline**.**setMap**(**map**);**

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

***this*.**mapMarkers**[**i**].**setMap**(**map**);**

**}**

**};**

**}**

**}**