GPS Data Analyzer
Brendan Young, Louden Yandow

Team composition:

What role(s) did each team member have. How did you balance the load across members?

Brendan Young - primary coder (building KML file, stops, turns, errand locations)

Louden Yandow - writeup writer, some coding (path, turns, reading in GPS data to a list)

We balanced the load by discussing what we needed to work on and deciding who would work on what. Brendan found the actual data analysis to be the most interesting so he chose to work primarily on that. Brendan also worked out how to build a KML file from the GPS data using the “simplekml” Python package, and worked on the writeups as well. Louden picked up whatever else there was left to do, which was mostly some of the writeups as well as reading in the GPS data and storing it into an easily accessible list for later analysis. Louden also built the KML path using the same method as Brendan from the simplekml package.

Abstract:

For this project, we created two Python files: GPS_to_KML.py and GPS_to_PlanningMap.py. GPS_to_KML.py simply takes in GPS data, and builds a path in an output KML file using <linestring> elements. The output KML, when loaded into Google Earth, will appear as a white path tracing the route of the vehicle. The path will extrude from the ground proportionally to the vehicle’s speed. GPS_to_PlanningMap.py will take in GPS data, and output a KML file that places colored pins for left and right turns, stops, and common errand locations.

Overview:

What were the problems you faced? What design decisions did you make? How did you solve them?

We did not encounter too many problems throughout this project. The project goals were pretty straightforward, and relatively easy to pursue.

One thing that was troublesome (but not necessarily a problem) is that some of the output KML files would be so large that they make Google Earth lag when a user zooms in or pans the planet. To resolve this, we merely ignore any GPS data that was a “GPGGA” sentence. This cut down the data we manipulate (and thus, output) by half, and drastically reduced the amount of lag created by a large KML file. We also created bin sizes of 5 to further reduce the number of data points we were working with.

Project decomposition:

How did you decompose, or break down the project? How did you design it to work? How did you test it and develop it?

We decomposed the project into many distinct, separate problems. We need to build a path from the GPS data. We need to determine where stops occur. We need to determine where turns (left and right) occur and differentiate them from roads that curve but aren't an actual turn onto a new road. And lastly, we need to determine common errand locations.

Separating the project into these distinct problems allowed us to easily understand it better, as well as develop solutions to each problem one by one. This made our workload much smaller, and less confusing.

Converting GPS to KML:

What issues did you have converting the files? How did you do data cleaning and anomaly detection? What issues were in the files?

One issue we encountered, as previously mentioned, was that we just had too much data in the output KML file that made Google Earth perform slower. We fixed this issue by ignoring GPGGA GPS data sentences, and only recording GPRMC. This was also useful for us because GPRMC contains the vehicle's speed, which we are able to use to raise the path from the ground to visually show when the vehicle was moving faster or slower. Overall, ignoring the GPGGA sentences halved the data we were working with, without losing too much accuracy.

Constructing <linestring> elements was made very easy by the use of the "simplekml" Python package. This allowed us to build a linestring object, and fill in various fields with the GPS data as needed. This package also allowed us to build the actual KML file very easily.

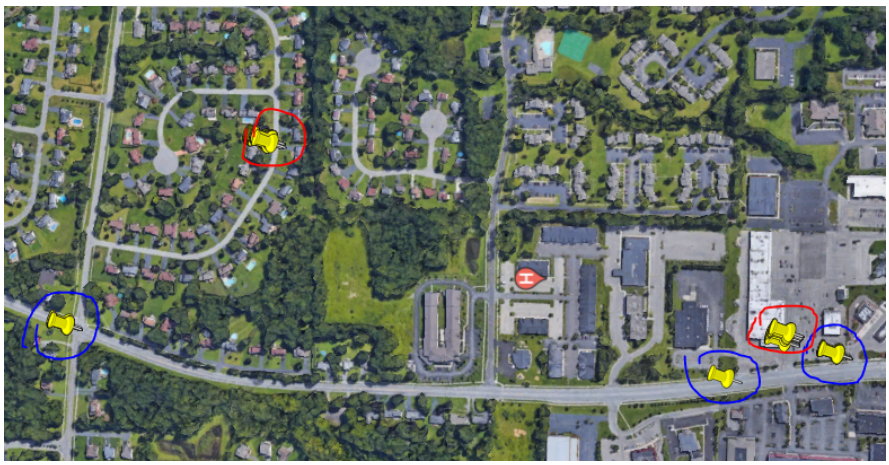
Stop Detection:

What classifier, and what features did you use to detect a stop or stop light? What issues were there with it? How did you ignore the case when the vehicle was stopped to run an errand such as drop off a book at the library or go shopping?

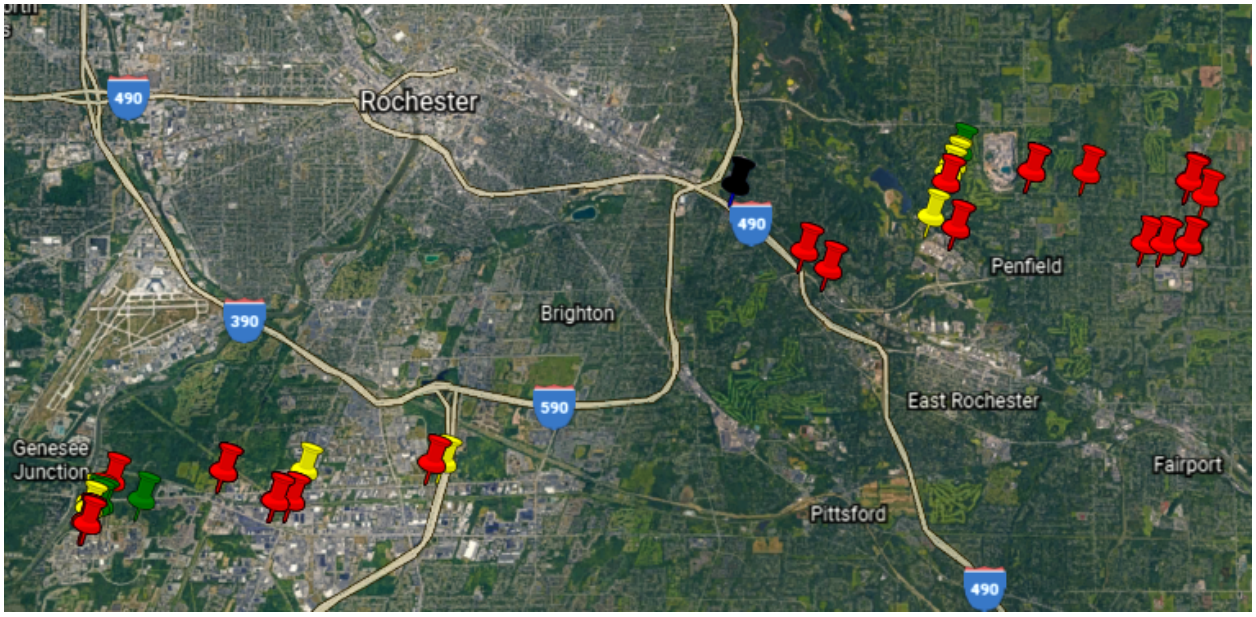
We used the vehicle's speed (as reported by the GPRMC data) to determine when the car was stopped. If the speed was 0, we determined that this record indicates an errand stop (or the start/stop of the journey). For stop signs and stop lights, we used 0.3 knots as a lower bound and 4 knots as an upper bound to determine if the vehicle would be stopped, but not necessarily stopped for a long time.

Our reasoning for this threshold is because vehicles need to slow down for stops, and since we know that the GPRMC data is recorded every quarter second or so, we knew that we could take some liberties with the speed to differentiate the stops from errand locations.

The following screenshot shows the results of our stop detection. Pins circled in blue indicate stop signs/stop lights (we have not yet colored the pins in code) while pins circled in red indicate errand stops (and the beginning/end of the trip):



The screenshot below shows the final outcome of the program with the same gps data used from above where now we added the threshold classifiers and clustering of data points together, although it is not 100% accurate, it is close enough to reduce the amount of points and see the data in a meaningful way.



Within this file, it seems that there is a loss of data points from penfield to RIT. (The blue datapoints did not show on this screenshot, but when the file is generate they are there).

Turn Detection:

What classifier, and what features did you use to detect a left turn? How about a right turn? Were there any issues?

For detecting turns, we first implement a threshold of speed. Generally, a good driver will slow down their speed when they make a turn onto a new road. Knowing this allowed us to isolate many turns.

To determine whether a turn was left or right, we attempted to use the latitude and longitude coordinates of the beginning, middle point, and end of the turn, to figure out if it was a left or right turn. This proved a little problematic because many GPS coordinates were not far enough apart to have an accurate difference in coordinates - that is, sometimes even if the vehicle traveled 20+ feet, the latitude and longitude did not change, resulting in faulty math. You will notice the program outputs some turns as incorrectly left or right.

We also attempted to implement a threshold classifier of speed based on the vector angle between a center point and 2 points after and if that angle is within a certain range then we know that the car is currently turning. The logic behind this method was solid, but was a bit difficult to implement so we provided the code that we did have for this version of the program within the submission just because it was something that we also worked on. A better implementation would be some kind of weighted cost function between the angle threshold, speed, and direction to get close to the best possible turn detection algorithm rather than just speed or angle. The angle

would have a weight based on how close it is to 90. Speed would have some weight associated with it based on previous and future speeds. The direction weighted being an exponential weight because direction would only work for north, south, east west and not if a turn was on an angle or small and only works for some cases not all. All these different types of ways would only get close to the actual turn, and most likely have some cases where it is not 100%. For example, the angle algorithm wouldn't work as well as a speed algorithm or direction if a road was very windy which is why we didn't use it in the final implementation, but it is better at detecting smaller turn changes. Speed is just an overall the best way to determine if you are turning because you usually will slow down to make a turn then change direction.

Assimilating across tracks:

Describe how you designed and wrote your program. What design pattern did you use? What intermediate data structures did you use? Did you build intermediate databases? Did you hold all the information in memory at once? Did you parse all of the files at once? How did you handle assimilation across different tracks? What parameters did you use?

Our code reads in GPS data using pynmea2, and stores it into a list of "NMEASentence" objects. These objects are parsed collections of a GPS sentence, from which we can grab useful information like latitude and longitude, speed, etc.

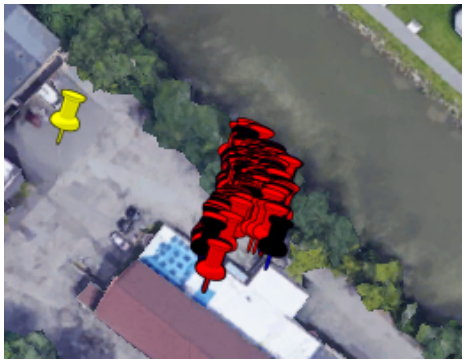
With this list, we can easily build a path using the "simplekml" Python library by filling linestring objects with the relevant information from the NMEASentence objects we stored. This remained true for figuring out stops, turns, and errand locations as well.

We tested each file one by one manually instead of combining our gps data together. We didn't feel the need to use data structures, although we did attempt to create our own data structure in another program that did not work as intended so we scrapped the idea, but kept the code within our final solution because we felt that the code was worth mentioning. This idea was scrapped, because there was a much easier way that was accurate enough to calculate points given the way the data is entered.This was assuming that the gps data we received is entered in order as a person starts their car and drives from and to each location. This assumption allowed us to condense our algorithms and create more efficient ones that will work when GPS data we were given. It allowed us to put opposite turns (to a location and back to the location you came from) on the same road instead of an agglomeration algorithm that would combine turns on the same intersection together would not work for turns.

One thing that we also do, is assume that the data was in order. This allows for us to cluster data together without having to make more complex calculations because the data is used in a linear time manner meaning that we can calculate the points as we go and not have to perform intensive agglomeration calculations. The one problem with this method is that if a person stays at a location for a super long time with the gps data

running, then there will still be many gps points at a single location (as seen in the image below from gps file 2021_09_BB_GPS_CHECK.txt). The same algorithm getThresholdCluster() was used for stops and common places for errands because it calculated the same thing, but with a different threshold. It takes the current point and appends a cluster as long as the speed of each consecutive datapoint is within the threshold range and then appends the data points together.

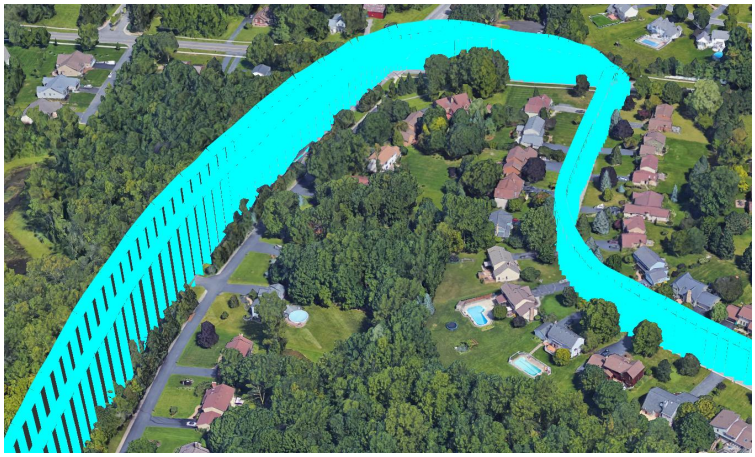
Although the algorithm we implemented for stops and errands works, it does come with some faults and could definitely use some improvement by agglomerating the points. Agglomeration would work better for stops and errand locations, but we got swamped with work and exams between checkpoint 3 and now and did not have enough time to implement this, but we recognize that it is the better way to go for stops and errands and the way we started would just be a way to reduce the amount of stops and errand locations to make the agglomeration faster.



Results of a path file:

Show an example screenshot of your path KML file.

The following screenshots show our result path from a given GPS data file. The path is cyan and extends vertically from the ground in accordance with the vehicle's speed. Higher paths indicate higher speeds from the vehicle.



Discussion:

What problems did you find with the approach? Did you have to do any noise removal or signal processing?

Overall, we had no major issues with our approach for the problems in this project. Reading in the GPS data and converting it to KML was almost trivially easy with the use of pynmea2 and simplekml Python packages. From there, we were able to play with the data to determine good thresholds/methods for determining stops, turns, and locations.

We did end up doing some noise removal by ignoring all GPGGA data. This reduced the data we worked with by half, without losing too much accuracy. With only GPRMC data, each data point still occurs roughly a quarter second after the last, which is pretty accurate for our purposes. Reducing the amount of data we read also reduced how much we output by half. This made loading the KML path into Google Earth quicker and did not impact the performance of Google Earth as much as it did when we included the GPGGA data.

Conclusion:

A summary conclusion of what you learned overall, and how this might be useful for some commercial applications.

Overall, we learned that there are myriad ways to approach analyzing data. From concrete algorithms to just “playing” with the data, one can find many different ways to learn from the data. For example, by playing with the speed thresholds and validating them on Google Earth, we were able to figure out good thresholds for indicating stops and errand locations. Although not the most accurate way to calculate the threshold, we knew nothing about this data and had no training data to start with to tell us what is and isn’t a stop so we had to find the threshold manually so we could train the data using our program.

There are also many ways to reduce the amount of work one has to do, such as “cleaning” via ignoring/removing data you do not want or need. For us, this involved ignoring the GPGGA data, which reduced the data we read/analyzed by half, and also creating bins for all the data points. We also ignored data points within our reading file if the datapoint was entered incorrectly or corruptly.

As for practical applications, projects such as this can be used to analyze travel patterns of people. By looking at common errand locations, one can figure out what ads might be better to serve a given person. Or, insurance companies may analyze a person’s speed and driving habits around turns, stop signs, and stop lights, to assess the risk of insuring a said person. Lastly, a given company may use this type of project to determine where, if any, competing businesses may be located, so they can make informed development decisions on new products, services, foods, or whatever the company may produce.