Localization Overview

Guy Stoppi

April 2019

0 Contents

1 Explanation of Localization's Tasks

In Winter of 2019, the Localization subteam was expected to 1) estimate the car's location and generate a costmap of its surroundings and 2) interface with the HD map of the competition track.

Estimating the car's location and costmap generation was mainly done with Google Cartographer (which was decided to be used last semester) with the LIDAR, IMU, and GPS. It was also suggested to use the HD map and the perception lane line detection to improve localization, which would require using a second sensor fusion algorithm `robot_localization`. Unfortunately, due to overlap with how Cartographer and `robot_localization` publish their output, we ended up only using Cartographer. Cartographer also generates a costmap of the surroundings.

Localization's task with the HD map has changed quite a few times over the semester and, as of April 25, we're working on publishing all of the lane lines near the car and describing if they're turning left or turning right. This is quite a complicated task; it includes parsing the protobuf files into usable text files and then, in real time, fetching and tagging lane lines near the car.

2 HD Map Parsing

The HD maps are in HERE format. Here's the format specification:
https://developer.here.com/olp/documentation/hd-live-map/topics/hdlm2-chapter-format.html

We use the lane model. Refer to "lane_parse.py" in the "hdmap_processing/scripts" folder of the localization repository. The lane file consists of a series of lane geometries, all defined by HERE's method of storing a polyline (i.e. a line defined by an array of points). Each of these lane lines makes up either an actual lane line on the road or an artificial lane line in an intersection that describes the ideal trajectory of the car when "using" that lane.

HERE's method of storing a polyline is very obtuse. Read the following links:
https://developer.here.com/olp/documentation/hd-live-map/topics/hd-map-coordinate-encoding.html
https://developer.here.com/olp/documentation/hd-live-map/topics/hd-map-coordinate-array-offset-encoding.html

Essentially, each lat-long coordinate is stored in a single number. To convert from that number X you do the following:

```
B = to_binary(X)

# every other bit starting with the second bit
lon = to_decimal(B[1::2]) * (360.0 / 2^32)
```

```
# every other bit starting with the third bit
lat = to_decimal(B[2::2]) * (180.0 / 2^31)
```

Now, when storing polylines, HERE map stores every point as an offset from the previous point using the XOR operation. So to decode:

```
prevB = ... # binary value of previous coordinate
B = XOR(prevB, to_binary(X))
lon = to_decimal(B[1::2]) * (360.0 / 2^32)
lat = to_decimal(B[2::2]) * (180.0 / 2^31)
```

Using these decode methods, we can generate a series of lane lines which can be used by our nodes running on the car. Note that these lane lines must be separated into "intersection lines" (artificial lane lines in an intersection) and "road lines" (actual lane lines) because of lane_publisher.

3 lane_publisher node - HD Map Interfacer

The lane_publisher node loads the lane lines (parsed using above) and searches through them to find the closest ones (all lanes within a certain radius). If these lane lines are artifical lane lines from an intersection, it will tag them as "turning left", "straight", or "turning right".

Searching through the lane lines is pretty simple: find the lane line's closest point to the car and, if the point is within a certain radius, return the lane line.

Tagging them takes more work. Although it's possible this methodology will change, the current method requires the lane lines to be translated into the car's local coordinate system.

The math behind this uses matrix multiplication and 2D rotational matrices. Suppose we have two 2D coordinate systems: the global one, G, and the local one, C. In our HD map case, G is the coordinate system for the HD map and C is the car's coordinate system. For now, we assume that C's origin is always at G's origin and so C is only different than G by its rotation θ. So we have that, for a point $L = (Lx, Ly)$ defined in C and is equivalent to the point $P = (Px, Py)$ in G:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} Lx \\ Ly \end{bmatrix} = \begin{bmatrix} Px \\ Py \end{bmatrix} \Rightarrow \begin{bmatrix} Lx \\ Ly \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} Px \\ Py \end{bmatrix}$$

The first matrix was obtained by the definition of a 2D rotational matrix and the second was obtained by taking the inverse of the first. So this allows us to translate a point in a lane line to the car's coordinate system IF the car is at (0,0). To get around this condition, we simply subtract the car's map coordinates from the lane point. This will translate the lane point into a coordinate system that isn't rotated relative to the global map's coordinate system but has the car at (0,0). We then apply the above transformation to get the lane point in the car's coordinate system.

Once we have all of the lane points in the car's coordinate system, we can observe how the lane is defined relative to the car to describe it as "turning left", "straight", or "turning right". We do this by taking the closest lane point and the furthest lane point and comparing how far to the left/right they are. If the furthest lane point is much further to the left than the closest lane point, then we say the lane "turns left". If the furthest lane point is much further to the right than the closest lane point, then we say the lane "turns right". Otherwise, we say the lane is "straight".

We collect all the lanes (which are tagged if they're in an intersection) which are within a certain radius of the car and publish them.