Model Documentation

The purpose of this document is to provide support documentation about the performed steps to implement Project 7 – Highway Driving from Udacity's Self Driving Car Nanodegree.

Summary

In this project, a simulator is provided. The goal of the project is to successfully make the car drive at least one entire lap around the highway without colliding with other cars, whilst also performing a smooth drive, never exceeding reasonable amounts of jerk.

The simulator already provides us with very essential information that is crucial to have when executing path planning of a self driving car. Among such data, the simulator provides:

- Car location (x,y)
- Car frenet location (s,d)
- Car velocity
- Yaw rate
- Sensor fusion data – in this case, sensor fusion is all the above information but regarding the other cars.

This information is going to be used in every iteration of the simulator with the program to generate a new finite number of points of where the car should be located in the future.

This path is planned is built in such a way that the car is able to drive around the highway the fastest way possible without breaking the speed limit, which in this case is fixed to 50 mph, and without exceeding the jerk accepted value for the project.

Disclaimer: During Udacity's lessons for the project, what the Mercedes-Benz team explains as a theoretical way to minimize jerk, is to use a fifth order polynomial equation. However, due to the simplicity of usage of the spline library as guided in the Project Q&A lesson, this library was preferred instead of using the fifth order polynomial. This spine library is used to solve the jerk minimization issue as it is given a set of points and a smooth transition between points is applied.

Behaviour Planning – Logic

For each update cycle of our path planning implementation, the car takes usage of sensor fusion data to verify what is happening with the other car.

A pseudo code algorithm of what is happening is described as below:

- The car checks the next 30 meters ahead of the lane where it currently is.
- If the car senses that another car is present in it's next 30 meters of the lane and it's slower than our car, it prepares to change lane.
    - If the car is on the most right lane, it only tries to verify if the middle lane is free. It does this by also evaluating the sensor fusion data. If it is free on the next 30 meters and also on the 15 meters behind, it effectively changes the lane, otherwise, it slows down and stays on the lane.

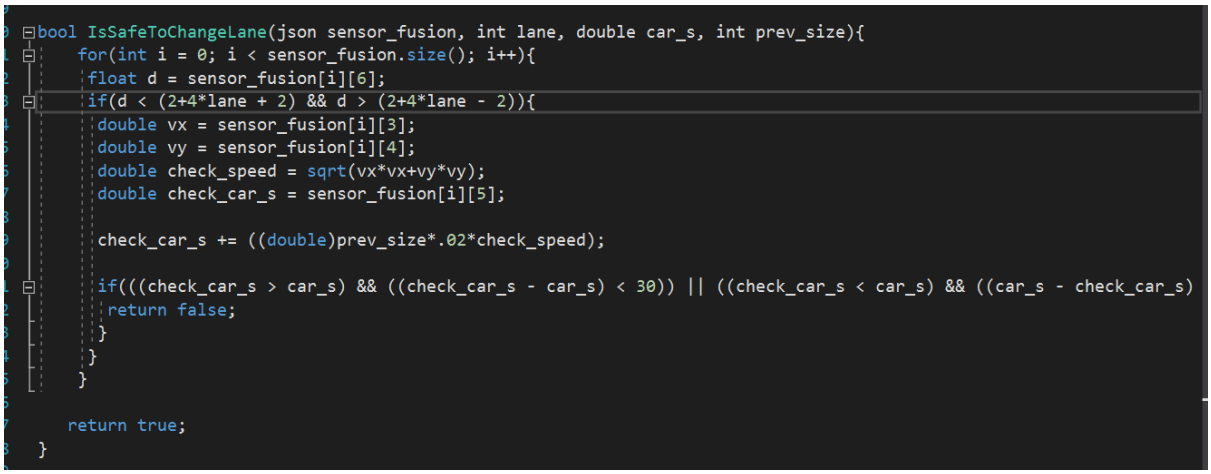
- If the car is on the middle lane, if first evaluates if the right lane is free in a similar process and explained above. If it is free, it moves to the right, otherwise, the same process is done to change lane to the left. If both lanes are busy, the car slows down and stays on the middle lane.
- If the car is on the most left lane, if tries to move with safety to the right as explained on the steps above.

Code reference

The picture below represents a custom implemented helper functions which act almost as a “cost function”.

This piece of code is what allow us to understand if it’s safe to change lanes or not in the code.

As previously mentioned, this check is possible to the usage of sensor fusion data that is already provided for us by the Udacity Simulator.



```
bool IsSafeToChangeLane(json sensor_fusion, int lane, double car_s, int prev_size){
    for(int i = 0; i < sensor_fusion.size(); i++){
        float d = sensor_fusion[i][6];
        if(d < (2+4*lane + 2) && d > (2+4*lane - 2)){
            double vx = sensor_fusion[i][3];
            double vy = sensor_fusion[i][4];
            double check_speed = sqrt(vx*vx+vy*vy);
            double check_car_s = sensor_fusion[i][5];

            check_car_s += ((double)prev_size*.02*check_speed);

            if(((check_car_s > car_s) && ((check_car_s - car_s) < 30)) || ((check_car_s < car_s) && ((car_s - check_car_s)
                return false;
            }
        }
    }

    return true;
}
```