SDCND Term 3 – Project 1: Path Planning

Introduction

The main goal of this project is to build a path planner for Udacity's term 3 simulator in order to navigate around a virtual highway with other traffic. According to the project rubric, the car must be able to drive for at least 4.32 miles: not crossing the 50 mph speed limit, not exceeding the comfort limits for the car's jerk and acceleration, not coliding with another vehicle, being able to change lanes consistenly and without spending too much time in between lanes.

Code Development

To start the development of the path planner, Udacity provided a starter *main.cpp* file with all the boilerplate code from the socket library necessary to comunicate with the simulator and a few extra helper functions unrelated to the planner. Udacity also suggested the *spline.h* library to help in the generation of smooth lane changes without crossing the jerk and acceleration maximums. This library aproaches the problem of trajectory generation in a different manner than the polynomial approach described during the lessons, but since it's fast and optimized it's a great fit for the project.

After a lot of trial and error and with the help of the Udacity Project Walkthrough video, I was able to get the car moving and changing lanes smoothly. I used a target maximum speed of 22 m/s (~49.21 mph) and an acceleration of 7.5 m/s^2. I also calculated a path 50 points long (at a speed of 22 m/s, this represents a horizon of 22 m). For the spline, I used 4 points: the first two were the last two points remaining from the previously calculated trajectory, and the last two were calculated at 30 m and 60 m ahead of the first points, but considering possible lane changes. These points were decided after some trial and error, until the requirements for maximum speed, acceleration and jerk were met.

After the trajectory generation problem was solved, it was time to develop a behavior planner. The first task to be developed was reading the output from the sensor fusion to detect the state of the traffic around the car. I decided to detect vehicles in a horizon of 30 m ahead on the same lane as my car, and 60 m on the lanes beside my car. This allowed me to monitor the gap between cars in lanes next to each other, to decide whether it would be best to change lanes or not. I also decided to detect vehicles that were either right next to or behind me on lanes next to mine, since depending on their distance to me, it could cause a colision if I decided to change lanes.

The last thing was to develop the Finite State Machine (FSM) responsible to control the car's behavior, given the sensed state of the traffic. The FSM developed has 2 state variables (*prep_change_lanes* and *changing_lanes*) and 2 variables (*target_lane* and *target_speed*). It doesn't rely on cost functions as instructed in the lessons, but it consists of several *if/else* statements precisely calculating likely future positions for the other vehicles and acting accordingly. This would probably perform poorly on a real life environment (due to several kinds of uncertainties), but it worked quite well on the simulator.

The FSM checks if there's traffic ahead first and, if so, it checks if it's too close. Then it looks for traffic on a lane (or lanes) next to it. First, if possible, it checks its left lane and afterwards, also if possible, its right one. During the side checks, the FSM looks for two things: there's no traffic neither right next to nor behind its car, and if one of the following three conditions are met. The conditions are: the lane is clear; or the car on the side lane is further than the car ahead; or the car on the side lane is faster than the car ahead. If these two things are met, the “prepare to change lanes” boolean flag goes true and, after the FSM finishes checking all the possibilities, it changes its state to changing lanes.

If there's no traffic ahead, the FSM checks if the middle lane is clear. If it is and the car is not in it, it sets the FSM state and variables to change lanes and go to the middle lane, since it's more likely to end in a better decision in the case of future traffic. While changing lanes, the FSM only checks for traffic ahead of its car. This prevents the FSM to change its mind about a decision mid lane change just because a new possibility came up. This behavior, if allowed, could make the car zigzag around the track in a dangerous manner.

Conclusion

The car behaves optimally most of the time, but it still fails to identify some optimal moves and gets stuck behind traffic. As mentioned before, in a real life scenario this would probably perform worse than a planner based on a fine tuned cost function, but according to the simulator, it performs quite well, so maybe it would be worth to try it on a real car.