Udacity Self Driving Car Term 3
Path planning

Hiu Chan

Design:

Waypoints creation:
We initially have no path plan, and we construct it from scratch. Once we submit our path plan (the path plan for the one second), we start collecting information of the previously submitted path plan from the sensor fusion. As soon as we have this information for more than 0.1 second, we preserve the first 0.1 seconds of the path plan and recalculate the next 0.9 second of path plan. This way the car can become more responsive if the overall environment changes or our previous prediction is inaccurate. For example, if the car at front suddenly slow down, my car can response more quickly. Since the maximum speed is 50mph, therefore in the plan planning program, I try to look ahead of the next 25 meter for my one second path planning. That work because 50 mph is about 22.352m/s and looking ahead 25 meter is more than enough.

Spline function smoothing, acceleration, and jerk smoothing:
To drive the car smoothly, there is a helper function call `getSmoothCurvePathPlan.` This function use the functions defined on spline.h that define the associated polynomial smoothing function so that the car can drive smoothly in the Frenet coordinate system. While in Frenet system, we rotate the coordinate so that the current yaw angle is 0. That make the thinking and implementation easier to understand while we are thinking about the logic.
Of course when we do the planning we need to think about the acceleration, jerk. The maximum acceleration is 10 m/s^2 and we define the path plan for the next one second, therefore as long as the acceleration does not exceed 1 meter per $1/10^{th}$ of seconds, the acceleration cannot exceed 10 m/s^2.

State machines:
There are two state machines being used in this design, one determines how fast/slow the car need to drive. The second state machine determine rather it makes or not to change lane. These two state machine work independently and the state transition for each of them does not depended on each other. However, they both share some of the same information to determine rather the state transition take place.

First state machine: Speed control

The car is designed to drive as fast as possible but does not exceed the maximum speed, which is 50 mph. There is a variable call smooth_factor which has a maximum value of 1. And that mean we can add at most 1 meter per 0.1 seconds or speed to make sure it does not exceed the acceleration of 10m/s^2. As soon as the sensor fusion detect that there is car in front of my car and is within my look ahead distance (25 meter in this case) and the car is driving slower

than my current speed, then I will set the variable speed_reduce to 0.9 and set the smooth_factor to 0.1. In the code, I have the following logic:

```
if (too_close){
    ref_vel = car_speed_ms – speed_reduce;
}
else if (car_speed_ms < topspeed-1.5){ // Trying to accelerate at
1m/100ms, which is 10m/s
    ref_vel = car_speed_ms + smooth_factor; // smooth_factor is 1 most
of the time. we update once every 100ms. therefore the acceleration is
10m/s^2
    if (smooth_factor < 1.0)
        smooth_factor += 0.1; // too smooth out the speed of the car
in case it is following another driving car.
}
else if(car_speed < topspeed – 0.5) // If I am close to top speed I
can allow (50mph), I want to slow down my acceleration.
    ref_vel = car_speed_ms + 0.3;
else if (car_speed_ms < topspeed)
    ref_vel = car_speed_ms + 0.07;
```

This logic basically said that if I am too close to the car in front of my car, I will have to slow down my car, otherwise, as I approach to topspeed, my acceleration will slow down until I reach the top speed. That I will maintain the same speed until the car in front of my car is too close. The very first “else if” also slowly increase the smooth_facter 0.1 at a time until it reaches 1. This is to prevent the fact that if the car in front of me is driving at 30 mph, I should not try to drive faster than this speed. I should preserve my current speed. This is achieved by cycling between the first “if” statement and the first “else if” statement. That mean in reality, we may see the transition between too close to the car in front and slowly accelerate by adding smooth_factor can be cycling until something else happen (such as lane changing, which is control by the other state machine, or the front of the car drive faster). The smooth_factor is always reset to 0.1 if we are very close to the front of the car. This is achieved by the following code:

```
// If the car is really close and slower than my current speed, then I
really need to slow down
if (check_speed < car_speed_ms && ((check_car_s-car_s) <
lookahead_distance + additional_dist)){
    speed_reduce = 0.9;
    // Smooth is used in case my car is following another car and I
start to accelerate again
    // I want to accelerate slowly in case I get too close and need to
slow down again, this help made the speed more stable
    smooth_factor = 0.1;
}
```

Second state machine: lane change decision

First, there is assumption made in this case, we always have three lanes (We know this is not true in reality but it works in this project). We first determine the best lane and then we try to figure rather it make sense or not to switch to the best possible lane. Our goal is to try to change lane as few times as possible. So best lane is defined as if there is no car detected in front of the current car frenet position. If there is more than one lane that have no cars in front, we try to pick the current lane if it is also the lane that has no car. Otherwise, we will try to pick the middle lane since the middle lane can later let the car switch to either left or right, that create more chance for the car to make smarter path planning later. If all three lanes have cars in front of our current frenet position, then we pick the lane with the car that has the furthest distance.

Now, having the best lane does not mean that we have to immediately switch to that lane. There are other factors to consider. If there is a car in front of me and I am not in the best lane, that is the time I should consider changing lane. If I am in lane 0 or lane 2 and the best lane is one, as long as there is no car behind lane one at my current frenet position, I will definitely change to lane one. If I am in lane 0 (or lane 2) and the best lane is lane 2 (or lane 0), I should try to proceed to lane 1 only if no car behind lane 1 and lane 2 (or lane 1 and lane 0) and is safe to proceed to lane 1. If I am currently at lane 1 and both lane 0 or 2 are better that my current lane 1 (either no car of car in front has further distance), I will select the one where no car behind. If both lanes are good, then I will just pick whatever the one being assigned as best lane. There are helper functions that defined noCar BehindThisLane and SafeToProceedLane. Both of these function help determines rather or not changing lane make sense.

Weakness:

There is a weakness for the design that I have. Suppose I am in lane 0 and there is a car in lane 1 beside me at constant speed and lane two has no car. My implementation will not be able to slow down the car, change to lane one and then lane two to speed up since two of the state machines are independent. But I think for a safe self driving car, this may not be very necessary because this may not make passenger feel very comfortable and safe, it can definitely make the car reach the destination but we may need more data to see rather this make sense or not from passenger point of view.