Path Planning

Andreas Gotterba

1. Edited Files

Model_Documentation.pdf	**This Document**
src/	
main.cpp	**main program**

2. Code Description

2.0 Structure

All my code is in main.cpp (unfortunately). I'll describe each of these sections:

1. State variables, and other important values

2. Predict location of other cars

3. Check each lane for nearby cars; set a lane speed; check if it's safe to change into lane

4. Set target speed for our car

5. Decide whether to change lanes, and if so, plan a jerk minimizing trajectory, and detect when lane change is complete

6. Compute a path in global xy coordinates for our car to follow, and send this path to the simulator.

2.1 State Variables

My state variables are declared just before the onMessage, so they persist across multiple calls. They are:

ref_vel: The velocity at the reference point (the last point that was previously sent to the simulator)

old_lane: If changing lanes, the previous lane I'm leaving (numbered 0,1,2). If not changing lanes, this is the current lane.

target_lane: If changing lanes, the new lane I'm entering. If not changing lanes, this is the current lane. Together old_lane and target_lane are used for the state machine of whether I'm keeping in the current lane, changing lanes left, or changing lanes right.

d_path: this is a list of d coordinates that should be used for the upcoming path. When keeping our current lane, it simply reflects the d_coordinate of this lane. But when changing lanes, it stores that jerk minimizing trajectory to be followed.

Additionally, these values are declared inside onMessage (I should have made many of them constants):

num_points: The number of points to plan forward of the car's current location. Since I keep all points from the previous path that weren't used (as shown in the walk-through), I want this to be small so that the car can react to changing conditions sooner. It only needs to be large enough to ensure the simulator doesn’t run out of points before it receives the next update from my code. 25 is still probably much bigger than needed.

lane_change_time: The amount of time to spend changing lanes. This is for the JMT planner; note that the car doesn’t actually spend this amount of time outside a lane.

2.2 Predict Location of Other Cars

Starting at line 325, I step through the sensor_fusion data and compute the speed of each car. I assume that they are all staying in their lane, so that this speed is all in the s direction. I then predict what their s locations will be when my car is at the reference point, as well as 1 lane change time after the reference point, and 2 lane change times after the reference point. I’ll use these to check if it’s safe to change into each lane. I should have created an object for every other car, but after starting with the walk-through, I didn’t want to rip my existing code apart.

2.3 Check for Nearby Cars; Set a Lane Speed; Mark if Safe to Enter Lane

Starting at line 359, I step through each other car again, looking at its location and speed (this is a separate loop since I originally intended to make it a separate function).

I set other_lane to the lane a car is in (assuming it’s not changing lanes, and taking the one that it’s closest to the center of).

‘lane_speed’ is the speed I use to decide which lane I want to be in, and is set to the speed of the slowest car within 80m in front of my car. I vacillated over whether this distance should be larger or not. Too large, and it will pick a slow lane even though it has enough time to change to a faster lane, pass a car, and then change back before reaching the distant slow car. Too small, and it won’t look at an entire clump of cars until it’s too late to change to the fastest lane, and may pick one that’s slower, getting boxed in.

‘lane_cur_speed’ is the speed I need to target if I’m currently in (or entering) a particular lane, and is set to the speed of the slowest car within 30m in front of my car.

If there’s a car within 15m in front of my car, I set way_too_close for that lane to true, so that I can take more aggressive action (maximum de-acceleration) to avoid it.

If there’s a car within 15m of my s location in a lane (front or back), or if there will be within the time it takes to change lanes, I mark the lane as not being clear. Compared to other cars in the simulator, this is more space than they leave. But compared to real driving, it’s quite small. Staying 2-3 seconds behind the car in front of you, as I was told in driver’s ed, would mean staying 44-67m away from other cars. And I refuse to cut another car off by moving in front of it without leaving as much room as

I would want in front of me. When my car is painfully stuck behind a slower car and isn't changing lanes, it's usually because there isn't this much space.

2.4 Set Target Speed

At line 400, I set target_vel to the 'lane_cur_speed' for the lane I'm currently in. If I'm entering a new lane and it has a lower lane_cur_speed, I use that instead. This only happens if I decided to enter a lane that was faster than the old one, but that lane slows down before I complete the lane change. Nonetheless, I complete the lane change, rather than trying to aborting, and risking being outside a lane for too long.

Note that I don't target a faster speed until a lane change is complete. Just because I'm changing lanes doesn't mean I can justify reducing my following distance.

Also, this isn't the speed that I'm going to use immediately. This is the speed I'll be accelerating to when I create points to send to the simulator, subject to my acceleration limit (max_accl_step).

2.5 Decide Whether to Change Lanes

At line 407, I decide whether to change lanes. I'll start a lane change if:

1. I'm not currently changing lanes (change_dir == 0)
2. The new lane is clear
3. The new lane is faster than the current lane
    a. I require it to be more than 1 m/s faster than the current lane so that I don't jump back and forth just because of noise (2 lanes that are about the same speed), except for the center lane, which I'll change to if it's practically the same speed as the current lane. I do this because the center lane has access to all the lanes, so I'm less likely to get 'boxed in' if I'm in the center lane.
4. The new lane is next to my current lane.

At line 423, I consider a double lane change. I'm looking 80 meters down the road to decide which lane to be in, so if I'm approaching a clump of cars, I have time to decide the best lane. If I'm flying along in lane 0 at 22 m/s, while there's a car 50m ahead in lane 1 at 15 m/s and a car in lane 2 60m ahead at 20 m/s, then I want to stay in lane 0. But if I suddenly see a car in lane 0 going 17 m/s, I want to switch to lane 2. The code at line 407 won't want to switch to lane 1 because it's slower, so this check sees if:

1. I'm not currently changing lanes
2. I'm in lane 0 or lane 2
3. lane 1 is clear
4. lane 2 or 0 is clear 1 lane change time in the future
5. lane 2 or 0 is more than 2 m/s faster than my current lane. Since this is a more elaborate operation that will take longer, I want the benefit to be higher than a single lane change.

You can think of lane_speed as my (simplistic) cost function, with some binary checks of whether a lane change is safe, and some strategy to look far ahead, and prefer the center lane (all else being equal). This is a greedy strategy- I never slow down so that I can fall back to a gap to change to a faster lane, but that would have been much more complex (I'd have to look for gaps behind, and make some cost judgments of the likelihood of reaching the gap without it closing, etc).

At line 435, I check if I'm starting a lane change, and if so, throw away everything in d_path after the reference point (which would have just been set to the d coordinate of the current lane, and call the compute_lane_change function to calculate a jerk-minimizing trajectory from the old lane to the new lane over lane_change_time.

compute_lane_change calculates the starting and ending d coordinates, and calls the jmt function to compute the trajectory (the code for this is what I wrote for the classroom quiz). It then computes the d coordinate for each point over the duration of the lane change.

This path gets concatenated to the old d_path so that the points I've already committed to reflect the values that will be used. I then fill d_path up with the d coordinate of the new lane. d_path has to be long enough to return a value for the farthest waypoint I use for the spline function; at 1000 points, it has plenty of space.

At line 448, If I'm currently changing lanes, I check to see if the maneuver is complete. Rather than relying on the d_coordinate I asked for, I check the car's actual d coordinate, and see if it's within 0.1m of the new lane's center. If so, I set old_lane to the new lane.

2.6 Compute a path in global xy coordinates

The structure of this follows the walk-through, but I'll point out some interesting notes. As was done before, I use the last point from the previous path that was sent as the reference point, and keep all the unused points before that without modification (this helps the car drive smoothly, though I considered using some kind of averaging with the new spline so that I could react more quickly). If starting from scratch, I initialize these points based on the car's current location and angle.

I calculate a spline based on points 40,80, and 120m ahead of the reference point, using the d coordinates from d_path (which has the jerk-minimizing trajectory if we're changing lanes). I tried using the jmt directly, but since it's based on the raw d coordinates, it was too coarse to prevent sudden large accelerations, so I still use a spline. I convert these points from Frenet to Cartesian (in the car's from of reference) and calculate the spline.

At line 520, I decide what speed to apply, which I'll use to set the distance between my points. Even though I set a target speed (target_vel) when looking at the other cars, that isn't the speed to be used. ref_vel is the speed that was last sent to the simulator (it's a state variable). If I was way_too_close to a car, then I de-accelerate at the maximum rate I allow. Otherwise, I asymptotically approach target_vel, based on the difference between ref_vel and target_vel. This keeps jerk quite low. From this, I set ref_accl_step, which is the amount I want to change my speed in the current step.

I use this to increase the spacing between every new point I'm adding at a constant rate- this is smoother than the walk-through, which would have increased the velocity for the first new point, but kept the points after that at the same speed.

target_dist is the straight distance from the x=ref_point to x=ref_point+30m (which is >= 30m since there's a change in y too), and I use this to compute target_fact, which modifies the x spacing of the points I'm adding.

Then once I have the points, I convert them from the car's xy to global xy, add them to message, and tell the simulator where to go next!