Path Planning Project David Simon Udacity

The path planning trajectory was detailed in the lecture using a quintile polynomial with a matrix solution procedure. I was going to try this approach; however, looking through the forum it appeared the few who attempted it had trouble getting the path to stabilize. Most forum students used the walkthrough video code from Aaron which uses a spline to establish the car's path. Aaron is also the one who created the simulator driving program. This code starts after lines 1113 through 1120 as he sets up ptsx and ptsy vectors and the reference state, car_x, car_y, car_yaw from the sensor output. If there is no path (prev_size < 2, lines 1122 - 1130), for example, at the start, the code uses the car location and its tangent back to get a previous point, giving two starting points for the path. Otherwise the code (line 1132 - 1144) uses the last two points from the existing path to start the next path (previous_path_x, previous_path_y). Then the code (lines 1146 - 1157) uses the function getXY to convert Frenet s, d at 30m, 60m, 90m to x,y coordinates to set up spline calculations for every 30 meters on the planned path within the lane of the car. To make the calculations easier, the code (lines 1159 - 1166) uses trigonometry to change from global (map) coordinates to a car reference with x=0, y=0, and heading of 0 in direction of the car path. The spline is declared with tk::spline s (line 1168); and then set up (line 1170) to handle the points s.set_points(ptsx,ptsy). The two starting points and the points at 30m, 60m, and 90m give five anchor points for the spline. Then under next_x_vals, next_y_vals (lines 1175 through 1210) within each 30 meters the upcoming points are calculated using the spline and added to any leftover points not used in the previous trajectory for a 50-point total. The actual distance x for each added point is determined by dividing the linear target distance over the 30 meters by the time period 0.02 sec and a conversion of 2.24 for m/s. double N = (target_dist/(0.02 * ref_vel/2.24)); double x_point = x_add_on + (target_x)/N The spline is called to calculate y, given x (double y_point = s(x_point)). A transformation back to global (map) coordinates is then done. x_point = (x_ref * cos(ref_yaw) - y_ref * sin(ref_yaw)); //back to map reference and y_point = (x_ref * sin(ref_yaw) + y_ref * cos(ref_yaw)); //back to map reference

The path is pushed back to next_x_vals and next_y_vals for trajectory generation on the simulator.

The Boolean 'too close' decision making code to trigger lane changes was also taken from the walkthrough video. This code is from lines 361 through 389 with the for loop that runs through the sensor output for all the cars and contains the 'too close' if statement. Variables d, vx, vy, and check_car_s are defined for the sensor output. If the leading car is ahead by 30 to 40 meters, the 'too close' Boolean is triggered to be true, and a lane change could be initiated. 30 meters worked ok, but 35 meters seemed better as the decision to change lanes came a little further back by 5 meters, providing a better angle for the trajectory and a quicker time for the decision to do a lane change. While 40 meters should have been better, some impact crashes from the rear made me change this; perhaps the sensor was not picking up the cars behind as well, but it's also true that our car is 5 meters closer to those rear cars as well.

The remaining code for lane changes is mine and is essentially a complicated decision-making procedure using if statements to determine whether the potential passing lanes are clear enough from ahead and behind to enable a lane change to pass the slow car ahead. There are this many if statements to cover all possible lane scenarios. Initially, after the car sensor data is given, the code determines the car lane using a sequential if statement block over the range of d for each lane (lines 279 – 292).   Originally, the main code to check for lane clearance was just abs(check_car_s – car_s) > 55 to 75 meters. If the car was in lane 0 on the left or lane 2 on the right, the middle lane, lane 1, ahead and behind had to be clear. If the car was in lane 1, the middle lane, the left and/or right lanes must be clear ahead and behind to allow a pass. This search was again done using a range of d for each lane. Round(d) also generally worked, though I thought every now and then it would fail as a cout of the sensor output seemed to indicate a little variation there. Couts of all the sensor output gave some information on the d and s for each car, but seemed to heavily disrupt the simulator, leading to judder(shaking) of our car. That was unfortunate, as not using this output made debugging more difficult. While simple, this abs distance code surprisingly worked, with a maximum run of 80 miles without any incident. However, the car was certainly not aggressive in its passing, as this 80-mile run was perhaps at most only 15 to 20 passes or one every 4 to 5 miles.

I started a new file with a more detailed code, looking at the front and rear separately for each lane, for example, cost_benefit_left_ahead and cost_benefit_left_behind. This, while more accurate, was not better, and perhaps somewhat worse. After some crashes I advanced this with a more complicated Boolean trigger scheme, for example, front0clear=true, rear0clear = true to pass in left lane. However, the Boolean seemed complicated, and did not seem to work well. In addition, the car would sometimes just stop, and the error would say Boolean, so perhaps the logic was incorrect somewhere. After removing the Boolean switches, the code worked well at times, but there would still be some mysterious collisions. I finally realized the sensors were picking up multiple cars in the same passing lane, and if the last update was a higher distance car, the lane change could be triggered, causing an accident as a nearby car with a lower distance would be ignored by the if statement lane change decision. To combat that, I created another variable cost_ahead_left, cost_behind_left, for example, that was temporary, but could be used to get the minimum of the cost_benefit_left_ahead and cost_benefit_left_behind. This way, the car would be looking at the updated nearest cars for a potential lane change, and then ignoring cars further away in the same passing lane. However, to do this, the minimum would have to be erased each sensor cycle to account for new distance numbers from the ever-changing locations of the nearby cars as the minimum calculation procedure would start anew. Currently I am using 7000 meters which is just above the max s value of 6945.554 meters; however, 1000 also worked fine, as I never have seen over 300 for the distance between the car and any other cars. Lower numbers below 300 did not seem to work at all. One of the greatest difficulties in the program has been to account for no cars either ahead or behind in a lane. I have used sort of a reverse procedure that generally seems to work. With the high reset, if there is no detection, that 7000 value will still be there and will indicate an open lane. I am also using a no_cars_left_ahead = 0 and no_cars_left_behind = 0, for example, which is reset after each sensor cycle. These variables = 1 if another car is detected within the if statement. Due to the interpretation by the program of the 7000 value being no cars ahead or behind, these variables may be redundant. Note that using too much of these decisions in combination did work to stop accidents where a car was there on the side yet still not detected, but this also returned our car to a passive state with minimal passes. An irritating problem still sometimes occurred at the beginning of the run. As our car starts in lane 1, the middle lane, if a nearby car is on the left or the right, but these same lanes ahead are generally clear, the car can then lane change into that side car. The sensors have not updated every location at that early point so the initialization 7000 value indicates the adjacent passing lane is clear. Though it seems odd, this appears to occur even with about 20 cycles at the beginning. As a result, for the first 30 cycles using a 'cycle' counter variable within an if statement, the system costs are set at 0. (lines 313 – 336) This 0 value causes the car to interpret that there are nearby cars, and then no passes are done. After the first 30 cycles, the regular reset after each sensor sweep begins, with the costs set at 7000 prior to the minimization distance procedure. (lines 338 – 357) This reset is contained within a for loop and so can be reset after more sensor sweep cycles if desired. It is set at just one full sensor sweep as that has appeared to work well. The distance from the nearest cars required for a lane change was set at least 70 meters ahead and behind. This distance was tested from 60 meters to 75 meters. 75 meters also worked well, but 70 meters means a sooner lane change. 75 meters meant more cars would pass our car, making the car more passive with less lane changes. Distance at 60 meters or 65 meters proved to be a problem as some lane changes, being closer to nearby cars, caused an outside lane violation. This was likely due to the difficulty in applying the trajectory given the speed and the closer distance for the lane change.

(lines 1093 – 1111) Reference speed was also tested, and the 49.5 mph reference speed from Aaron's code worked well; higher than that caused occasional acceleration violations. The 'too close' velocity changes at the end of the if statement lane change block were derived from the walkthrough video. Velocity was changed from the 0.224 in the video. For slowing down, -= 0.27 was used, as sometimes a lower number would allow our car to rear end the slow car if the slow car braked hard. An aggressive += 0.3 was used to push our car for faster lane changes as well as full speed in clear lanes. A yo-yo speed effect is inherent in the simulator; this is not realistic for actual driving. Cars ahead are speeding up and slowing down repeatedly when usually each car would try to go at a relatively constant speed, matching the slower car speed ahead. In the simulator a following car cannot match the slower cars speed ahead and maintain a constant following distance while waiting for the passing lane to clear. The coding for car_speed <= slow_car ref_vel+=0.15 is an attempt to smooth out that yo-yo speed effect. The code below that line is an attempt to reduce the failures that occur during a very hard deceleration of the slow car in front, causing our car to surprisingly just stop. The code with car_speed <= 3.0 and ref_vel+=0.05 is an attempt to keep the car going forward.

I also tried to address a rare collision that occurs from a side car lane changing into the car. At the end of each lane's if statement passing block I used if statements for any nearby side cars having a d range near the lane edges which triggers a hard braking of ref_vel -=0.27 for our car in an attempt to avoid this imminent side collision. Lastly at the end of each lane's if statement passing block I tried to address rear end collisions with an if statement for our lane that monitored the location of the nearby trailing car. If that car got under 5 meters away and our car_speed was under 5 mph, an increase of ref_vel +=0.20 would attempt to avoid the rear end collision. I do not know if these if statements for side lane change collision and rear end collision will work as previous attempts within the if statement block to increase the velocity by 0.05 to 0.1 mph for smoother lane changes seemed to cause acceleration violations. Note that an average speed, variable ave_car_speed is calculated early in the program (lines 267 – 277) and is used in the lane changes with a requirement of minimum average speed of 30 mph. This was an attempt to block slow lane changes that were difficult due to oncoming traffic from the rear, especially if those cars had a higher speed. Also, the slow speed in the lane changes seemed to cause outside lane violations as decision making is not as good, with some hesitation from the slower lane change interacting with the changing distance of the oncoming cars from behind. A more complicated approach for a lane change that I tried used different distances to account for more distance needed with faster oncoming cars behind and less distance for the slower ones behind. This did not seem that effective, and was complicated, so the longer 70-meter distance was used instead to have enough distance from cars behind to enable a clear lane change, regardless of the trailing car speed. Note that I initially did try using our car_speed for the lane change, but problems occurred as this car_speed is going up and down behind the slow car. While a minimum over 30 mph seemed to work ok, a trigger under 40 mph caused hesitation. Average speed overall seemed to work better, so that was used instead of current car speed. I also tried to implement the 'too close' or car_speed minimum or car_speed maximum to enter the if statement lane change block, but this again produced hesitancy in the decision process for lane changes.

(Lines 392 – 1091) The large if statement block for lane passing generally goes through logic scenarios, triggered by the 'too close' = true for the slow car in front of our car. If our car is in lane 0, the middle lane must have nearby cars at least 70 meters away in front and behind. If no car is in front and behind, the initialization at 7000 will be above the 70 meters. Also, the no_cars_left_ahead = 0 and no_cars_left_behind = 0 statements are also set. These last two options are likely redundant to the 7000 initialization value. If the middle front or middle behind is less than 70 meters, our car stays in lane 0. Any cars detected at all in the middle lane alters the 7000-cost benefit number and this also changes the no_cars_left_ahead = 1 and the no_cars_left_behind = 1. This exact logic scenario is repeated if our car is in the right lane, lane 2. The middle lane is more complicated, as clearance checks need to be done for both the left lane and the right lane. The left lane and the right lane may have cars within the 70-meter distance both ahead and behind, causing our car to stay in the middle lane, lane 1. Or perhaps, a car is either ahead or behind within the 70 meter zone on the left, so that lane is not clear, but maybe the right side is clear with either no cars ahead and behind or perhaps cars are at least 70 meters ahead and behind or it is a combination of clear ahead or behind with cars ahead or behind at least 70 meters away, all allowing a right lane change. This complicated situation can be repeated for the left lane, with the right lane now not clear, allowing passing in the left lane. Lastly, both lanes could be open due to a combination of cars at least 70 meters away ahead or behind or the lanes are clear, allowing passing either within the left or the right lanes. Here if the cost_benefit_left_ahead is greater than the cost_benefit_right_ahead, indicating more room for the car to pass and drive at full speed, the car chooses the left lane. If this is reversed, and the cost benefit for the right side ahead is higher, the car chooses the right lane. If both lanes are clear with no cars ahead, and clear behind due to no cars or cars at least 70 meters away, our car will take the left lane, lane 0, as that is usually the faster lane. My code does not look at two lanes away which could be advantageous if that lane is clear. Certainly, simulator cars were changing lanes two lanes away to take clear lanes. That would be a further improvement to the program, instead of waiting for the adjacent lane(s) to open. I have, however, seen our car do two sequential quick lane changes to the open lane.

Lastly couts are used throughout the if statement block to keep track of the minimum cost benefits for the nearby cars as the trigger is the 70 meters minimum clearance needed for passing in the adjacent lanes. The cout for the cost benefit behind the car is also used to keep track of potential situations for a rear end collision. Upon each lane change there is a cout for the average car velocity, and a cout for each lane change which is labeled for the if statement location in the code. (H Lane 1 to 2, for example).

All these couts are useful for monitoring the proper decision making for the lane changes and debugging the code if any violations occur.