Path Planning Project

Hamed Saeidi

November 29, 2017

Abstract

This reports presents the details of the implementation of the mathematical models in the path planning project for the self-driving car nanodegree.

Path Planning Implementation

1) Preventing maximum jerk, maximum acceleration, and speed limit violations.

I used a simple model of a real car $m\dot{v}(t)+bv(t) = f(t)$ where m is the car mass and b is a viscous damping for the effects of air drag. Using this model, I first produce a controlled force for speed tracking (either the maximum speed or the speed of front car in lane). Two Proportional Integral controllers take care of producing this force input depending on the car following or free driving speeds. The force is then passed through the dynamic car model to produce the car velocity. The car velocity produced at the output of this model includes a virtual car mass (here 800kg) which prevents sudden motions of the car and hence violating maximum jerk, acceleration and speed limits by preventing the car to teleport. The function "double control_speed(double ref_speed)" in the main.cpp code implements this dynamic model. The resulting car velocity is used to calculate how much the car can move in 20 milliseconds.

```
// control the speed for tracking a reference speed and also avoiding
 collision with the front car
double control_speed(double ref_speed){
double dist = 0;
double kp = 0.0; // proportional control gain
double ki = 0.0; // integral control gain
double k_safe = 0.0; // a proportional controller for keeping the safe
 distance with the front car

double e = ref_speed - vc; // current tracking error
e_integral += e*dT;
// parameters of a car model with mass m and viscous damping of
b: m*v_dot + b*v = f (control force)
double m = 800;
double b = 10;
// choose these gains when driving at max speed
if (ref_speed == MAX_SPEED){
kp = 300;
ki = 50;
// and switch to these gains when a car is in front
}else{
kp = 1000;
ki = 50;
k_safe = 100;
}
// produce the control force to the car model
double force = kp*e+ki*e_integral + k_safe*(gap - gap_desired);

// apply the dynamic model of the car to prevent sudden jerky motions
 and car teleport
double vn = (m*vc + dT*force )/(b*dT+m);
// saturate the velocity output
vn = saturate(vn, MAX_SPEED);

// calculate how much the car should move on the road (distance to travel
 in this step)
dist = vn*dT;
std::cout << std::fixed << std::setprecision(2);
// keep the last value of vn for the next control loop.
vc = vn;
return dist;
}
```

2) Path generation.

At each control iteration, a smooth spline is created between the last two points of the previous path and 3 distant (30 meter apart) points in front of the car in the target lane. Using the traveled distance calculated in the previous section (i.e. via the virtual car model), some new points on the spline are selected and added to the previous trajectory of the car to obtain a total of 50 points for a 1 second (50*20ms) trajectory similar to the following code snippet

```
int N = 50;
// produce the remaing parts of the trajectory to get 50 points
for (int i = 0; i < N-prev_length; i++){
// find how much car should move using the controller and car model
dist_inc = control_speed(ref_speed);
// find the corresponding next point on the trajectory (using the traveled distance)
// updates xc, and yc in the car frame to find the next points
vector<double> next_tmp = find_dxdy(dist_inc,xc,yc,spl);
xc = next_tmp[0];
yc = next_tmp[1];
// update the next way points in the world frame
next_tmp = in_world_frame(car_x, car_y,car_yaw, xc,yc);
next_x_vals.push_back(next_tmp[0]);
next_y_vals.push_back(next_tmp[1]);
}
```

For the first time the control loop runs, since the previous trajectory is empty, we use the current position of the car and 1 meter behind it in addition to the three distant points to create the spline.

```
//initialize the spline points
if ( prev_length < 2){
double car_x_prev = car_x - cos(car_yaw);
double car_y_prev = car_y - sin(car_yaw);
curve_x.push_back(car_x_prev);
curve_x.push_back(car_x);
curve_y.push_back(car_y_prev);
curve_y.push_back(car_y);
start_x = car_x;
start_y = car_y;
start_s = car_s;
vc = 0;
//if the previouse path exists use the remaining parts of it
}
.
.
.
//produce three distant waypoints
for(int i = 1; i < 4; i++){
vector<double> next_curve = getXY(start_s+i*target_move,target_d,...
... map_waypoints_s,map_waypoints_x,map_waypoints_y);
curve_x.push_back(next_curve[0]);
curve_y.push_back(next_curve[1]);
}
// transfrom the points to the car frame
for (int i = 0; i < curve_x.size();i++){
vector<double> transformed = in_car_frame(car_x,car_y,car_yaw,curve_x[i],...
...curve_y[i]);
curve_x[i] = transformed[0];
curve_y[i] = transformed[1];
}
```

3) Lane changing policy. In the "calc_cost" function, the costs of keeping lane (KL), changing lane to left (LCL) and changing lane to right (LCR) are calculated according to the following. The cost of keeping lane is calculated according to the following cost function which penalizes the differences between the car speed and maximum speed.

$$cost_{max}(1 - e^{-\frac{|\mathrm{speed}_{\mathrm{car}} - \mathrm{speed}_{\mathrm{max}}|}{\sigma}}) \tag{1}$$

If the car gets stuck behind another car this cost goes up. In this equation σ is an adjustable constant. Increasing σ decreases the cost of driving at low speeds in the cost function (1).

For checking the left and right lanes (if they are not the road shoulder or the opposing lane), the following cost functions are used.

$$cost_{max} e^{-\frac{|s_{\text{car}} - s_{\text{car in the other lane}}|}{\sigma}} \tag{2}$$

Cost function (2) checks if there is a car nearby in the chosen lane. The maximum value is obtained when a car is exactly to the right or left. σ is a constant in this equation. Increasing σ increases the range around the car in which the presences of other cars are felt and added to the cost function. In order to predict if a lane change results in a collision, the following cost function is also considered for the left/right lanes.

$$cost_{max}(e^{-\frac{\text{speed}_{\text{car}} - \text{speed}_{\text{car in the other lane}}}{s_{\text{car}} - s_{\text{car in the other lane}}}} - 1) \tag{3}$$

Using the cost function (3), if a car is behind and is driving faster or if a car is leading but driving slower, there is a chance of collision and the cost of changing lane goes up. The cost functions (2) and (3) are calculated and summed for the nearby cars in the right/left lanes. The three cost functions are implemented in code as exp_cost1 to exp_cost3. The minimum cost is checked at 1 second intervals and a target lane is chosen and passed to the trajectory generator.

```
ctr ++;
// every second after the 5th second make decisions about changing or keeping lanes
 ( initially the car has not come to speed on road to make stable decisions)
if (ctr % 50 == 0 && ctr > 250){
// calculate the costs of keeping lane, changing lane to left, and right
vector<double> all_costs = calc_cost(car_s, car_d, car_speed, sensor_fusion);
double min_cost = MAX_COST;
int min_ind = 0;
cout << "from: ";
// find the minimum cost
for (int i = 0 ; i < all_costs.size(); i++){
cout << all_costs[i]<< " ";
if (all_costs[i] < min_cost){
min_cost = all_costs[i];
min_ind = i;
}
}
// change lane if necessary
cout << "chose: ";
switch(min_ind){
case 0:
cout << "KL" << endl;
break;
case 1:
cout << "Left" << endl;
target_lane -= 1;
target_lane = max(target_lane , 0.0);
break;
case 2:
cout << "Right" << endl;
target_lane += 1;
target_lane = min(target_lane , 2.0);
break;
}
}
```