MPC for SDC nano-degree for Udacity, by Hiu Chan

This is the documentation describe the implementation detail of the MPC.

The implementation is exactly identical to the way how the class material described. Here are the steps

1. Set N and dt.
2. Fit the polynomial to the waypoints
3. Calculate initial cross track error and orientation error
4. Define the components of the cost function (state, actuator, etc).
5. Define the model constraints

Here, N * dt is the time we look ahead and I want to look ahead at least one second. If "N" is too large, we will require more computational time. And I want to design something as responsive as I can. I choose a small value of N, which is 10. So, my dt is 0.1 seconds, end up that I am looing ahead of what is happening in the next one second. Since my car will be very responsive, there is no need to look to far beyond one second. And I do think this is the right balance. Especially with 0.1 delay of response time to be is reasonable since I am not designing a super-fast self-driving car that try to drive at 200mph.

Next, fiting the polynomial to waypoint is easy. But before I go any further. The simulator give me the global coordinate system waypoint. I convert them into a coordinate system based on the current position of my car.

```
double shift_x = ptsx[i]-car_position_x;

double shift_y = ptsy[i]-car_position_y;

ptsx_transform[i] = shift_x * cos(0-psi) - shift_y * sin(0-psi);

ptsy_transform[i] = shift_x * sin(0-psi) + shift_y * cos(0-psi);
```

After transform the waypoints into a new coordinate system with my car current position as a center point, I used the provided function polyfit to fit those points into third order polynomial function.

Now, I have my third order polynomial function, time to calculate CTE and orientation error. We simple fit the polynomial function at position 0 for CTE since we created the function based on the current location of the car. The x value "0" represent the CTE for this third order polynomial function. The orientation error is simply the following

```
psi - atan(coeffs[1] + 2 * car_x * coeffs[2] + 3 * coeffs[3] * car_x * car_x)
```

Since we use the car current position as center of the new coordinated system, car_x become zero and initial psi is zero, and it simplify to the following:

```
-atan(coeffs[1]);
```

For the cost function, I have CTE, current velocity of the car, the car orientation, the steering, the actuator, and the difference of both the steering and actuator. Now, I want my car to be as closed to the center of the road as I can all time, so, I put a large constant of 1800 and 1700 for the weighting of CTE and the car orientation respectively so that we try to reduce as much those error as we can when doing optimization. In my view, speed is not the objective and therefore I put a very small weighting for velocity.

For the model constraints, they are simply copied from the quiz in the class material. All these implementations are located in the MPC class. The constraints and update equation are located in the class FG_eval, which Is implemented in the MPC.cpp file. The MPC solve function is mainly for book keeping and preparation purpose before the FG_eval instance is used for optimization.

Now, we need to handle latency since we cannot expect the car will response immediately base on what we want the car do. For this project, I assume that the latency is 100ms. And they are defined in the main.cpp with the following:

```
double dt = 0.1; // set latency to be 0.1 seconds

// Predicted what is going to happen 0.1 seconds later and use these values for our calculation.

double predicted_x = v*dt + a*dt*dt/2;  // distance = v*t + 1/2 * a * t^2

double predicted_y = 0; // 0 since we want the car follow the track.

double predicted_psi = v*-s/Lf*dt;

double predicted_v = v + a*dt;

double predicted_cte = cte + v*sin(epsi)*dt;

double predicted_epsi = epsi + v*-s/Lf*dt;
```

Here, predicted_x is the current x plus the amount of distance the car will travel. Current x is zero, we can just ignore it. According to calculus or physics, distance = velocity * t + ½ a * t * t. For predicted_y, it will be zero in my case. Although I can make some prediction base on the current car steering wheel position and the car orientation. But I tried it and it did not make too much different. The rest of the predicted values are coming from the class material. One thing that we need to pay attention, that is the values “a” and “s” are coming from the simulator as steering angle and throttle. The issue is the throttle does not exactly mean acceleration or deceleration for the car, it only means how hard the gas pedal or the brake is being stepped on. We may have to fine tune this by applying a constant to it. But since this is only a prediction and the time gap is only 100ms, the error generally do not matter until I want to design a super fact car.