Udacity CarND Term2 Project5 MPC Writeup

Joe Zhou

"Describe the model in details."

In this project we use a simple kinematic bicycle model for the vehicle's first order dynamic parameters such as location, heading, steering and throttle. And for vehicle control technique we use Model Predictive Control (MPC), which is a good fit due to its superior MIMO capability [Kong-Pfeiffer-Schildbach-Borrelli 2015]. The vehicle state is defined by six variables: vehicle *x* and *y* positions, heading(ψ), speed(v), cross-track error(cte), and heading error($e\psi$). And the two actuators are steering angle and throttle. So the model has two inputs and six outputs. And their update equations are:

$$x_{t+1} = x_t + v_t * cos(\psi_t) * dt$$

$$y_{t+1} = y_t + v_t * sin(\psi_t) * dt$$

$$\psi_{t+1} = \psi_t + \frac{v_t}{L_f} * \delta_t * dt$$

$$v_{t+1} = v_t + a_t * dt$$

$$cte_{t+1} = f(x_t) - y_t + (v_t * sin(e\psi_t) * dt)$$

$$e\psi_{t+1} = \psi_t - \psi des_t + (\frac{v_t}{L_f} * \delta_t * dt)$$

And actuator constraints are [-1, 1] for throttle value and [-25 °, 25 °] for steering value. And the cost, aka the objective or reward, for the MPC model is composed of seven components, as shown in the codes below.

```
//fg[0] is chosen to store the sum of all costs
//initialize it to 0
fg[0] = 0;

for (unsigned int t = 0; t < N; ++t) {
  //1st cost: location ... represented by cte
  fg[0] += w0_cte * CppAD::pow(vars[cte_start + t] - ref_cte, 2);
  //2nd cost: vehicle heading
  fg[0] += w1_epsi * CppAD::pow(vars[epsi_start + t] - ref_epsi, 2);
}

for (unsigned int t = 0; t < N; ++t) {
  //3rd cost: for stop-and-go situation
  fg[0] += w2_v * CppAD::pow(vars[v_start + t] - ref_v, 2);
}

for (unsigned int t = 0; t < N - 1; ++t) {
  //4th cost: for avoiding sudden jerk in steering
  fg[0] += w3_steer * CppAD::pow(vars[steer_start + t], 2);
  //5th cost: for avoiding sudden jerk in acceleration
  fg[0] += w4_a * CppAD::pow(vars[a_start + t], 2);
}


//6th and 7th costs: for differences of actuators values
//between adjacent look-ahead steps
for (unsigned int t = 0; t < N - 2; ++t) {
  fg[0] += w5_diff_steer * CppAD::pow(vars[steer_start + t + 1] -
           vars[steer_start + t], 2);
  fg[0] += w6_diff_a * CppAD::pow(vars[a_start + t + 1] -
           vars[a_start + t], 2);
}
```

And based on above state update equations, actuator constraints, and cost definition, the MPC solves for an optimal trajectory under a **time limit**.

"Discuss picking timestep length N and elapsed duration between timesteps dt."

The *dt* is the **time limit** stated in prior section. At each *dt*, MPC controller finds an optimal solution for the constrained problem. At each *dt*, MPC provides a trajectory update for the path planning. And more frequent update is better, particularly when the vehicle model is a simple first order approximation. So small *dt* is desired. But small *dt* depends on the computational complexity of the constrained problem and the performance capability of the hardware. If the control problem is too complex or the hardware cannot meet computational demand, small *dt* is not possible. The product of N and dt defines a finite and undiscounted horizon. For most vehicle controls the horizon can be around 1 second. In this project, I settle with N = 10 and dt = 0.1 second to have a 1 second horizon.

"Discuss polynomial fitting for waypoints."

Waypoints are given to provide a reference path, which can be represented by a 3rd degree polynomial. But waypoints are given in global coordinations. So the first step is to transform them into the coordination system in which vehicle always sits in the origin and heads toward the positive x direction. The transformation is a typical 2D translation and rotation, as shown in the codes below.

```
//convert waypoints from global coordination to car's coordination
Eigen::VectorXd waypoint_x(waypoint_global_x.size());
Eigen::VectorXd waypoint_y(waypoint_global_y.size());

for (unsigned int i = 0; i < waypoint_global_x.size(); i++){
  //2D translation
  double translation_x = waypoint_global_x[i] - vehicle_global_x;
  double translation_y = waypoint_global_y[i] - vehicle_global_y;
  //2D rotation
  waypoint_x[i] = translation_x * cos(psi) + translation_y * sin(psi);
  waypoint_y[i] = translation_y * cos(psi) - translation_x * sin(psi);
}

//x, y are the waypoints' x and y positions in vehicle coordinates
auto coeffs = polyfit(waypoint_x, waypoint_y, 3);
```

"Discuss latency incorporation."

Latency is common in a complex control system. It can be the delay caused by the actuators or the time taken by the computer in solving the constrained problem. MPC can handle latency. MPC reformulates its initial state by assuming the inputs and latency are applied to the original initial state then feeds the new initial state into the solver. Its project codes are shown below.

```
//case: with latency

//in the vehicle coordination system
//assume vehicle starts at the origin and pointing at x direction
double vehicle_x0 = 0.0;
double vehicle_y0 = 0.0;
double psi0 = 0.0;

//double epsi0 = psi0 - atan(coeffs[1]+2*vehicle_x0*coeffs[2]+
//                           3*vehicle_x0*coeffs[3]*pow(vehicle_x0,2));
double epsi0 = -atan(coeffs[1]);

//update state after certain latency

//for vehicle position after latency, use initial heading psi0

//double vehicle_x1 = vehicle_x0 + (v_mps * latency) * cos(psi0);
double vehicle_x1 = v_mps * latency;  //shorthand equation
//double vehicle_y1 = vehicle_y0 + (v_mps * latency) * sin(psi0);
double vehicle_y1 = 0.0;  //shorthand equation

//sect06 lesson19
//double psi1 = psi0 - (v_mps * latency) * steer / Lf;
double psi1 = -(v_mps * latency) * steer / Lf;  //shorthand equation
double v1 = v_mps + a* latency;


//cross-track error
//double cte1 = polyeval(coeffs,vehicle_x0) - vehicle_y0 +
//              (v_mps * latency) * sin(epsi0);
double cte1 = polyeval(coeffs,vehicle_x0) +
              (v_mps * latency) * sin(epsi0); //shorthand equation
//vehicle orientation error
double epsi1 = psi1 - atan(coeffs[1]);

//create the state vector
Eigen::VectorXd state(6);

state << vehicle_x1, vehicle_y1, psi1, v1, cte1, epsi1;
```