P5 Model Predictive Controller (MPC) Project

The goals of this project are the following:

- Implement an MPC in C++ to control the steering angle and throttle of the car in the simulator.
- The vehicle must successfully drive a lap around the track in autonomous mode.
- The MPC should minimize the cross track error (cte) and orientation error (epsi) to stay in the center of the track.
- Drive as fast as possible while staying on the drivable surface at all times.
- Optionally, visualize the waypoints and the predicted car trajectory in the simulator.

The Model

1. the State

The state consists of 6 variables, px, py, psi, v, cte and epsi.

- px: x coordinate of the vehicle
- py: y coordinate of the vehicle

- psi: the orientation of the vehicle
- v: the velocity of the vehicle
- cte: cross track error is the distance of the vehicle from the reference trajectory (i.e the waypoints)
- epsi: orientation error is the difference of the vehicle orientation and the trajectory orientation.

In this project, I chose to convert the waypoints from map coordinate to car coordinate, therefore, the state variables are expressed in car coordinate. For the time being, we simplify the state to the following.

- px = 0 because the car is at the origin
- py = 0 because the car is at the origin
- psi = 0 because positive x is always the car heading
- v = v
- cte: evaluate the y value using the fitted polynomials with x set to 0, subtracted by py (= 0)
- epsi: evaluate the difference between psi (= 0) and the tangential angle (=arctan(f'(px))) of the fitted polynomials at px , where px = 0

Later in the section "Model Predictive Control with Latency", I talk about how I make adjustment to the state variables by considering the effect of the latency.

2. the Actuators

There are 2 actuators in the MPC model, the steering angle and

throttle. Using the coefficients of the polynomial, the state as the input applied with constraints, we can solve for the optimal actuator values. (see `auto actuators = mpc.Solve(state, coeffs)` in `main.cpp`) The actuator values are sent back to the simulator via a json object. The steering angle has the constraint of +/- 25 degrees; the throttle can be set within the range of +/- 1, but I chose a slightly more conservative upper bound at 0.8.

3. the Update Equations

We applied the kinematic model in the MPC. The kinematic equations are implemented in the class function `operator()` of the class `FG_eval`.

Here are the equations for the model.

```
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] + v[t] / Lf * delta[t] * dt
```

```
v_[t+1] = v[t] + a[t] * dt
```

```
cte[t+1] = f(x[t]) - y[t] + v[t] * sin(epsi[t]) * dt
```

```
epsi[t+1] = psi[t] - psides[t] + v[t] * delta[t] / Lf * dt
```

The above equation are implemented as constraints with vector `fg`.

See `line 134 to 141` in the class function `operator()` of the class `FG_eval`.

```
fg[2 + x_start + i] = x1 - (x0 + v0 * CppAD::cos(psi0) * dt)
```

```
fg[2 + y_start + i] = y1 - (y0 + v0 * CppAD::sin(psi0) * dt)
```

```
fg[2 + psi_start + i] = psi1 - (psi0 + v0 * delta0 / Lf * dt)
```

```
fg[2 + v_start + i] = v1 - (v0 + a0 * dt)
```

```
fg[2 + cte_start + i] = cte1 - ((f0 - y0) + (v0 * CppAD::sin(epsi0) *
```

```
fg[2 + epsi_start + i] = epsi1 - ((psi0 - psides0) + v0 * delta0 / Lf
```

Since I chose to use a 3rd order polynomial to fit the waypoints, I make sure this is reflected in the model's f0 and desired psi. (see `line 121 and 122` in `operator()` of `MPC.cpp`)

Timestep Length and Frequency

Timestep length is represented by N and timestep frequency is represented by dt. The product of N and dt is referred to as the horizon T. In general, T is short (a few seconds at most) as the environment will change enough that it won't make sense to predict any further into the future. In this project, after some trial and error, I chose N = 7 and dt = 0.08. The larger the N, the longer it takes to obtain a solution with

`MPC::Solve()`. So choosing the smaller value of N that allows the car to drive around the track makes sense.

Besides longer latency, the effect of larger N sometimes could cause instability in the control. When N is too small, the car veers off the track quickly.

Smaller dt could result in over sensitive control. As a result, the car may overshoot more frequently and the trajectory is more wobbly. On the other hand, larger dt could result in less fitted polynomial and the car could go out of control at tight turns.

Polynomial Fitting and MPC Preprocessing

In this project, I chose to fit a 3rd order polynomial to the given waypoints. (see `coeffs = polyfit(ptsx_e, ptsy_e, 3)` in `MPC.cpp`) Since waypoints are given in the map coordinate, I need to transform (see `line 112 to 121` in `main.cpp`) them to the car coordinate first before curve fitting. In addition, since the data type of the waypoints are given in `vector<double>`, and the data types of the parameters in `polyfit()` function are in `Eigen::VectorXd`, type casting were performed before curve fitting. (see `line 123 to 127` in `main.cpp`)

Model Predictive Control with Latency

In reality, the car does not actuate the commands (steering angle and

throttle) instantly. Before the new commands take effect, the car continues to travel. To provide better prediction, we need to consider the effect of the latency. In this project, we consider 2 types of latency.

- latency 1: 100 ms which is defined for the project
- latency 2: the latency created by the Solver function to find a solution

By using `chrono::high_resolution_clock`, I measured the actual latency 1 of the thread in sleep (see `t_sleep_time_ms` in `main.cpp`). Similarly, latency 2 (see `t_Solving_ms` in `MPC.cpp`) can be measured by instrumenting `MPC::Solve` in `MPC.cpp`. Combining latency 1 and 2 (=dt) which is a small time difference, we can apply the kinematic model to predict the change in position px, psi, v and fine tune cte and epsi.

- difference in v: dv = acceleration * dt where acceleration is the last throttle input to the simulator
- difference in px: dx = (v + dv) * dt
- difference psi: d_psi = (v+dv)/Lf * delta * dt, where delta is the last steering input to the simulator, Lf is defined as 2.67
- cte: evaluate the y value using the fitted polynomials with x set to dx, subtracted by py (= 0)
- epsi: d_psi - atan(coeffs[1] + 2 * coeffs[2] * dx + 3 * coeffs[3] * dx * dx). d_psi replaces the (psi=0) , and dx replaces the (px=0), both mentioned in the above State.

Video: MPC in action

Video: MPC in action