Model Predictive Control (MPC)

Hamed Saeidi

June 22, 2017

Abstract

This reports presents the details of the implementation of the mathematical models in the model predictive control project for the self-driving car nanodegree.

Implementation Rubrics

1) Student describes their model in detail. This includes the state, actuators and update equations.

I used the following model of the dynamics of the car and errors (similar to what was included in the lectures). This model uses a right-hand coordinate for the steering angle δ_t. However, since the simulator uses the left-hand coordinates for δ_t we need to negate the outputs of the model prediction controller before sending it to the simulator.

$$
\begin{aligned}
x_{t+1} &= x_t + v_t cos(\psi_t) dt && (1)\\
y_{t+1} &= y_t + v_t sin(\psi_t) dt && (2)\\
\psi_{t+1} &= \psi_t + \frac{v_t \delta_t}{L_f} dt && (3)\\
v_{t+1} &= v_t + a_t dt && (4)\\
\text{cte}_{t+1} &= f(x_t) - y_t + v_t sin(\psi_t) dt && (5)\\
\text{e}\psi_t &= \psi_t - \psi_{d_t} + \frac{v_t \delta_t}{L_f} dt && (6)
\end{aligned}
$$

Since the optimization software requires the parameters to be in a single vector, the dynamics of the states in the N predcition steps are concatenated in the "fg" vector. Moreover, the optimization function accepts the dynamic equations in the form of $\mathbf{x}_{t+1} - \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t) = 0$ instead of the form $\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t, \mathbf{u}_t)$ (where $\mathbf{u}_t$ is the input vector and $\mathbf{x}_t$ is the states vector). The implementation is included in the FG_eval class in the "MPC.cpp" file according to the following.

```
// apply the dynamic equations of motion (here I assumed a right handed
// coordinate for delta and after finding the optimal value,
// I will negate it and apply it to the simulator
fg[1 + x_start + t] = x1 - (x0 + v0 * CppAD::cos(psi0) * dt);
```

```
fg[1 + y_start + t] = y1 - (y0 + v0 * CppAD::sin(psi0) * dt);
fg[1 + psi_start + t] = psi1 - (psi0 + v0 * delta0 / Lf * dt);
fg[1 + v_start + t] = v1 - (v0 + a0 * dt);
fg[1 + cte_start + t] = cte1 - ((f0 - y0) + (v0 * CppAD::sin(epsi0) * dt));
fg[1 + epsi_start + t] = epsi1 - ((psi0 - psides0) + v0 * delta0 / Lf * dt);
```

2) A polynomial is fitted to waypoints. If the student preprocesses waypoints, the vehicle state, and/or actuators prior to the MPC procedure it is described.

I realized that changing the coordinates from the world frame to the body frame makes the implementation of the MPC much easier tractable. Therefore, I used the following simple transformation to convert the world frame coordinates to the body frame.

$$\begin{aligned} x_b &= x_c cos(\psi) + y_c sin(\psi) \\ y_b &= -x_c sin(\psi) + y_c cos(\psi) \end{aligned} \tag{7}$$

where x_c and y_c are world frame coordinates and x_b and y_b. Here, I used the relative displacements of the waypoints compared to the car in the transformation (i.e. dx and dy in the following code from the main.cpp file.)

```
// compensating for the delay (predict 100 milliseconds into the future
// and use the results as initial values for the MPC
double delay = 0.15; // 150 ms delay
double dt = 0.05;
for (double k = 0; k < int(delay/dt); ++k){
px += v*cos(psi)*dt;
py += v*sin(psi)*dt;
psi -= v*delta*deg2rad(25)/2.67*dt; // steering_angle = 1 means 25 degrees
//(0.44 radians)
v += acc*dt*5.0; // 5.0 is a multiplier (max acceleration) explained in
//details in the MPC.cpp file
}
// preparing the cure fit points in the vector format (I transform all of
//the readings to the car's body frame for easier calculations)
for (unsigned int k = 0; k < ptsx.size(); ++k){
double dx = ptsx[k] - px;
double dy = ptsy[k] - py;
ptsx_vec[k] = dx * cos(psi) + dy * sin(psi);
ptsy_vec[k] = - dx * sin(psi) + dy * cos(psi);
}
```

In this code it can be seen that I apply an estimation of the states into the future (150 ms from the current moment) so that I can use them as the initial values of the MPC states to handle the latency.

3) The student implements Model Predictive Control that handles a 100 millisecond latency. Student provides details on how they deal with latency.

Here, I am repeating the code snippet from the main.cpp file. In this code, I use a multi-step method with the dynamic equations of motion (1)-(4) to predict the states into the future and use them as the initial value for the MPC to bypass the latency.

```
// compensating for the delay (predict 100 milliseconds into the future
// and use the results as initial values for the MPC
double delay = 0.15; // 150 ms delay
double dt = 0.05;
for (double k = 0; k < int(delay/dt); ++k){
px += v*cos(psi)*dt;
py += v*sin(psi)*dt;
psi -= v*delta*deg2rad(25)/2.67*dt; // steering_angle = 1 means 25 degrees
//(0.44 radians)
v += acc*dt*5.0; // 5.0 is a multiplier (max acceleration) explained in
//details in the MPC.cpp file
}
```