Model Predictive Controller (MPC)

Brent Wylie, September 8th, 2017

The goals / steps of this project are the following:

- Create a Model Predictive Controller in C++ using class structures
- Make a successful lap around the track using the MPC controlled steering & throttle, without leaving the track surface
- Understand and explain how the model based controller works and how latency is accounted for

The model

My model based controller uses a simplistic kinematic model, which ignores forces such as tire forces, gravity, mass, and the like. It solely uses 2D position, velocity, orientation (psi) to describe the vehicle state. To track and minimize error, I added cross-track error (cte) and orientation error (epsi). A more informed model would be more accurate, especially at higher speeds, but because the simulator and car model within are simplified, a simple model such as this made the most sense.

The model also uses two actuation variables, delta, and a, which correspond to steering angle and throttle value respectively. Delta operates in a range [-25,25] with -25 meaning full left, +25 full right, and 0 meaning no turn. ‘a’ operates in a range [-1,1], with negative values signifying braking and positive values signifying acceleration.

The equations used to predict the state on the next time step in this model are the following:

$$x[t+1] = x[t] + v[t] * \cos(psi[t]) * dt$$

$$y[t+1] = y[t] + v[t] * \sin(psi[t]) * dt$$

$$psi[t+1] = psi[t] + v[t] / Lf * delta[t] * dt$$

$$v[t+1] = v[t] + a[t] * dt$$

Lf is the distance between the front of the vehicle and it’s center of gravity, which was provided by udacity.

Dt is the time between timesteps, and was to be decided by us in this project.

The cte and epsi were used to build part of the cost function for this MPC. This function was used to optimize behavior and actuators to accomplish the goals of staying on the track in a safe manner.

The cte and epsi are updated with the following equations:

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

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

Timestep length and elapsed duration

I chose a timestep length of 2 seconds and an elapsed duration, dt, of 0.1. This translates to an N of 20. At first, I chose lower values of N, as low as 10 but found higher speeds needed more forecasting. I fixed dt at 0.1 (100ms) to be equal to the specified latency.

Polynomial fitting & MPC Preprocessing

Prior to fitting the waypoint polynomial, I performed some preprocessing. Namely, I transformed the waypoints from a global coordinate system to the vehicle’s coordinate system. This made it much easier for the solver, as in the vehicle coordinate system, the state variables x, y, and psi are 0 at any given time. This is done in lines 100-107 of main.cpp.

After preprocessing, I convert the waypoint vectors to Eigen compatible vectors and fit the third order polynomial.

Model predictive control and latency

One requirement for this controller, was that it be able to account for actuator latency. I handle this in my MPC class by delaying the actual activation by one timestep. Since my dt is equal to the specified latency of 100ms, this is all that is needed to delay the activation. Note that because this and the simulator are very computationally heavy programs, excessive CPU load on the host could result in additional latency which could throw off operations. For this reason I ran at a lower resolution and the lowest graphical quality. To account for any additional latency, I set the speed to 50mph for safe operation.