MPC Controller

Andreas Gotterba

1. Edited Files

agotterba_mpc_writeup.pdf	**This Document**
src/	
main.cpp	**main program**
MPC.h	**MPC controller header file**
MPC.cpp	**MPC controller implementation**
Videos	
50 mph lap:	**https://youtu.be/q0zoL7J5ZWo**

2. Model Description

2.1 State

The state variables are:

x:	The x coordinate of the car
y:	The y coordinate of the car
psi:	The angle the car is pointing
v:	The speed of the car
cte:	The cross track error of the car, relative to the computed (polynomial) centerline
epsi:	The error in the angle of the car, relative to the computed angle of the centerline

Note that all measurements are made relative to the car's current position. So in the car's current state, x, y, and psi are all 0. These state variables take on different values for future time-steps, as the model predicts what each of these values will be *relative to the car's current position*.

2.2 Actuators

The actuators are:

sangle:	The steering angle of the car (processed in radians, and then converted to '25 degrees = 1.0' at the end
acc:	The acceleration (throttle) of the car. The model is ideal (so it really considers this acceleration), but there is resistance/friction/drag in the simulator. Thus, when the model targets 50mph, it ends up running at around 48.5 mph.

2.3 Update Equations

For each time-step, the model predicts the values of the state variables, based on the previous state and the values of the actuators that were applied. The fundamental 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] - (v[t]/Lf) * sangle[t] * dt
v[t+1]	=	v[t] + acc[t] * dt
cte[t+1]	=	f(x[t]) - y(t) + v[t] * sin(epsi[t]) * dt
epsi[t+1]	=	psi[t] - psides(x[t]) + (v0/Lf) * sangle[t] * dt

Most of the operands are the state and actuators described above; the additional parameters are:

Lf: The length from the front axle of the car to its center of gravity. Supplied as part of the starter code, from the result of a controlled simulation.

f(x[t]): this function returns the y coordinate for the centerline of the road at a given x coordinate (and I use the x coordinate for time t). This could be any means of figuring out the y coordinate; in this implementation, I'm using the polynomial model from the waypoints received from the simulator, so I simply evaluate the polynomial at x[t].

psides(x[t]): this function returns the angle of the road's centerline at a given x coordinate (and I pass in the x coordinate for time t). This could be any means of finding the angle; in this implementation, I'm using the polynomial model from the waypoints received from the simulator; I compute the derivative of the polynomial, then take the arctangent of that to find the angle.

The actual equations I'm using are in lines 180 through 189 of MPC.cpp. Note that the code is a bit more involved than what I've shown here. Instead of the raw update equations, these are implemented as constraints. On its own, the model would think it could pick any value for x[t+1] that it likes. But I've already constrained fg[1+param_start+t] to be zero. Therefore, these equations tell it that x[t+1] minus (the equation I wrote above) should be zero, thus constraining the value of x[t+1] that it picks to be equal to the update equation written above.

3. Timestep Length and Elapsed Duration

I selected my values for elapsed duration (or rather, number of timesteps N) and the timestep length (dt) iteratively, but with a few guiding ideas:

1. I watched the distance that was being predicted ahead by plotting all the predicted values using the mpc_x and mpc_y keys in the json message, and compared it to the range of the waypoints supplied by the simulator. Predicting beyond the waypoints isn't useful, since the polynomial it's trying to fit to is no longer matching the road. This distance varies around the track (becoming quite short in the tight turns), but as N and dt are constant, I chose a distance I thought struck a good compromise around the track.

2. I actually set T (the amount of time to look ahead) to a fixed time and compute N from T and dt. That way, if I decide to change dt but want the modeled amount of time to stay the same, N automatically adjusts. I found that looking half a second ahead wasn't enough, but a full second is fine.

3. I want dt to be small enough to maintain accuracy, but not so small as it has a noticeable impact on performance. When I set T to 1 second and dt to 20ms (N was therefore 50), I did see worse performance. Things are much better with T = 1 second and dt = 50ms (N = 20).

4. As I'll discuss later, I'd like the latency to be a multiple of dt, since that simplifies how I'm handling latency (means I don't need any interpolation).

4. Polynomial Fitting and MPC Preprocessing

When I started the project, I tried keeping my calculations in absolute coordinates, imagining that if I was getting the waypoints from localization, that would be the frame of reference I was last in. I soon realized this was problematic: since the path is being approximated as a polynomial, what would happen when the car is headed in the y direction, and the correct line is no longer a function. The path would go crazy, as it tries to fit in the wrong direction. Once I switched to working in the car's coordinates, everything became easier. I read the waypoints into raw_ptsx and raw_ptsy, then transform them to the car's coordinates (using the reported x,y, and psi values). Then I know that the car is at 0,0 and psi is 0 (in the car's reference frame). I can call the supplied polyfit function on the transformed waypoints, and know that cte is the constant coefficient and epsi is -1 * the arctangent of the linear coefficient.

From there, I know the current state is (x,y,psi,v,cte,epsi) = (0,0,0,v,cte,epsi) where v is reported from the simulator while cte and epsi are calculated from the polynomial as I described, and I can call Solve with that state and the polynomial's coefficients as inputs.

Once the steer_value and throttle_value are returned, I scale the steer_value (which was calculated to be between +/- 0.436 radians) to range between +/- 1. The throttle value was already in its +/- 1 range, so it doesn't need to be scaled. The steering and throttle values are then ready to be sent to the simulator.

5. Latency

There was a lot of talk on the forums about predicting where the car will be in 100ms, by just assuming it continues with the steering angle and throttle that it currently has. That seemed to me like a poor man's model predictor- why should I use that when I can remember what values of acc and sangle were sent in previous timesteps, and have already implemented a heavy-duty model predictor that gets run every iteration?

So instead, I compute n_latency: the number of timesteps that are consumed by the latency; in this case n_latency is 2. I've added n_latency as a variable of the MPC object, as well as sangle_hist and acc_hist, which store the previous values of steering angle and throttle; they get initialized to zeros in the init function.

When the model makes its prediction, it constrains the values of sangle and throttle for the first two timesteps to be those that have already been sent (at lines 289 and 306 of MPC.cpp). This is similar to how it normally constrains these to +/- 0.436 and +/- 1; just for these time steps, they are constrained to be exactly the values that were sent.

Then when reporting the values to use next, instead of sending the values from the first timestep of the model (sangle[0] and acc[0], I send sangle[n_latency] and acc[n_latency] (in this case sangle[2] and acc[2]), and push those values into the sangle_hist and acc_hist queues, to be ready for the next iteration.

I found this system to work extremely well- the car is very stable, and stays centered in the lane. I confirmed that when I enabled the latency but did not correct for it, it would oscillate across the road.

I tried adding even more latency to the computation (the effective_latency) while the applied_latency remained 100ms, to account for the additional time to compute the new values, and get a new request from the simulator. But I found that while I could measure this loop delay with system_clock commands, I couldn't tell when the new values were applied within the simulator. I could probably nail it down by trial and error, but found that performance was fine when I simply set effective_latency equal to the applied_latency.

Note that because I use the latency to compute n_latency, I set it as a variable (applied_latency) at line 15 of main.cpp, instead of hard coding it at the this_thread::sleep_for command.