Model Predictive Control (MPC) Project

Introduction

This project implements a model predictive controller to drive a car around a track. The controller handles a >100 ms latency in sensor readings and actuator inputs in addition to the latency introduced by the time taken by the controller itself to compute the actuator inputs.

The Model

The controller uses a kinematic model which ignores tire forces, gravity and mass. While this simplification reduces the accuracy of the model, it makes it much more tractable. For the vehicle speed of 30-40 miles per hour, which is what this controller is designed for, this model approximates the vehicle dynamics pretty well. Note that though this controller is designed for a reference speed of 30-40 mph, it performs well for speeds up to 50 mph.

The State

The model state consists of the following variables:

1. x[t] - the x-coordinate of the vehicle at time t.
2. y[t] - the y-coordinate of the vehicle at time t.
3. psi[t] - the heading angle of the vehicle at time t.
4. v[t] - the speed of the vehicle at time t.
5. cte[t] - the cross-track error of the vehicle at time t.
6. epsi[t] - the error in the heading angle of the vehicle at time t.

Inputs

The model has two inputs:

1. delta[t] - the steering angle at time t
2. a[t] - the acceleration (or thrust) at time t

Model Equations

The following equations can be used to obtain the new state at time t + dt from an existing state and inputs at time t:

1. x[t + dt] = x[t] + v[t] * cos(psi[t]) * dt
2. y[t + dt] = y[t] + v[t] * sin(psi[t]) * dt
3. psi[t + dt] = psi[t] + v[t] * (delta[t] / Lf) * dt
4. v[t + dt] = v[t] + a[t] * dt
5. cte[t + dt] = cte[t] + v[t] * sin(epsi[t]) * dt
6. epsi[t + dt] = epsi[t] + (v[t] / Lf) * delta[t] * dt

Here, Lf is the distance between the front of the vehicle and its center of gravity, and is set to 2.67 meters for this controller.

Timestep Length and Elapsed Duration (N & dt)

After trying with many variations, I eventually ended up using N = 20 and dt = 0.1 second. For speeds of under 40 miles per hour, this seemed to provide the best result.

In order to find the optimal values of N and dt, I first started with N = 10 and dt = 0.5 and used trial-and-error to arrive at the final result. My main motivations in this tuning process were:

1. Accuracy - In order to reduce the discretization error as much as possible, I first reduced dt to as low as I could with N = 10.
2. Tractability - I then tried to increase my timestep horizon by increasing N as far as possible without significantly impacting latency.

Using the above motivations, I ended up with N = 20 and dt = 0.1 second as the optimal values for speeds below 40 mph. At higher speeds, a lower value of N (maybe 10) coupled with a higher value of dt (maybe 0.2 s) might perform better since the model is more susceptible to latency at higher speeds.

Polynomial Fitting and MPC Preprocessing

While position (x, y and psi) and speed (v) measurements are available from vehicle sensors, the errors (cte and epsi) must be computed. The track is full of waypoints at the center of the track which can be observed by the vehicle as it gets near them. The errors are computed by fitting a curve that describes the track around the location of the vehicle and computing the position of the vehicle relative to the fitted curve.

The designed controller fits a 3^{rd} order polynomial to the waypoints and uses the following equations to compute the cte and epsi:

1. cte[t] = y[t] - f(x[t])
2. epsi[t] = psi[t] = psi_des[t]

Here, f is the fitted polynomial and psi_des[t] can be calculated as the tangential angle of the polynomial f evaluated at x[t], i.e., psi_des[t] = arctan(f'(x[t])).

The vehicle sensors sense position (x, y, psi) in the global coordinate system. However, for the purposes of the controller's internal computations, it makes sense to convert these coordinates to the vehicle coordinate system before passing them to the controller. This is because it is much easier to make the above computations of error in the vehicle coordinate system. Since x is always 0, cte[t] = y[t], and psi_des[t] = arctan(f'(0)). This conversion also makes it easier to visualize the waypoints and the predicted trajectory.

The controller implementation uses the following equations to transform the waypoints into the vehicle coordinate system:

1. x’ = r * cos(theta)
2. y’ = r * sin(theta)

Here, r and theta are calculated as below:

1. r = sqrt((x - x_v) ^ 2 + (y - y_v) ^ 2
2. theta = arctan((y - y_v) / (x - x_v)) - psi

Here, x’ and y’ are the vehicle coordinates being computed, x, and y are the coordinates of the point being converted in the map coordinate system, and x_v and y_v are the coordinates of the vehicle in the map coordinate system.

Handling Latency

In a real car, there is a time lag between the actuator command from the controller and the point of time when the command actually gets applied to the vehicle. In this implementation, a thread sleep of 100 ms is used to artificially introduce latency between the controller command and actual vehicle actuation. There is additional latency of about 20 ms introduced by the controller code itself. These latencies need to be handled for the controller to work properly.

The principle of how to handle latency is simple: since we know that whatever actuation commands the controller provides will be applied about 120 ms later, we use the predicted state 120 ms later than the current state to compute the actuations. This is done using the following equations:

1. x’ = x + v * cos(psi) * latency
2. y’ = y + v * sin(psi) * latency
3. psi’ = psi + (v * tan(delta) / Lf) * latency + ((a * tan(delta) / (2 * Lf)) * pow(latency, 2))
4. v’ = v + a * latency

This controller implementation actually measures its own time of computation, which includes the artificial delay of 100 introduced due to the thread sleep, averages it with the latency used in the previous controller run, and uses the average value in the above equations to compute the delayed state. This approach effectively compensates for the latency.