MPC Project Reflections

Project Objective

The purpose of this project is to develop a nonlinear model predictive controller (NMPC) to steer a car around a track in a simulator. The simulator provides a feed of values containing the position of the car, its speed and heading direction. Additionally it provides the coordinates of waypoints along a reference trajectory that the car is to follow. All coordinates are provided in a global coordinate system.

Principle:

Model predictive control (MPC) is an advanced method of process control that has since the 1980s. The following picture shows the main steps to conduct the model predictive control.

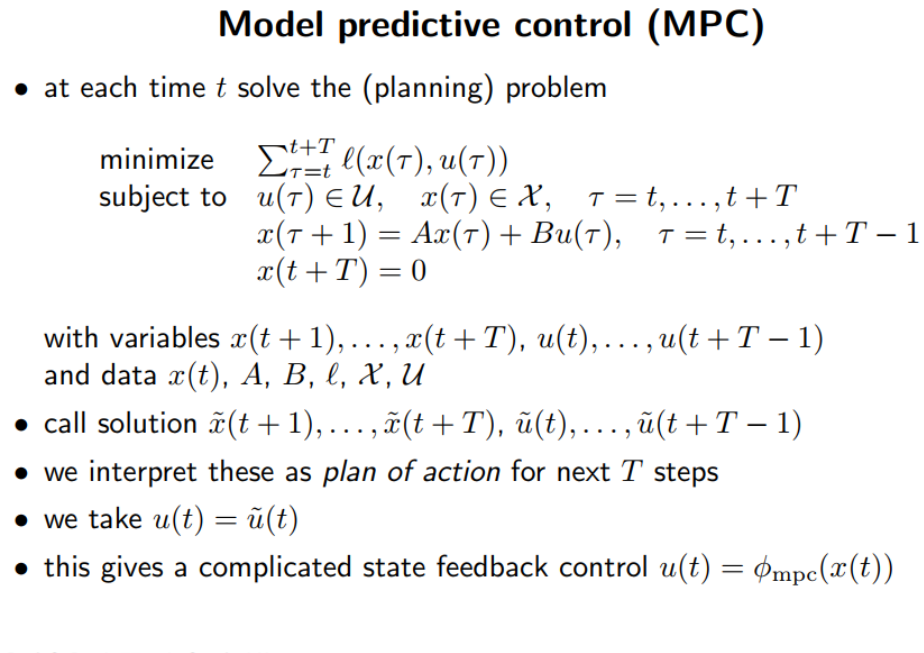

Figure 1. MPC Steps

Implementation:

- Set up:
    1. Define the length of the trajectory, N, and duration of each time step, dt.
    2. Define vehicle dynamics and actuator limitations along with other constraints.
    3. Define the cost function.
- Loops:
    1. Pass the current state as the initial state to the model predictive controller.
    2. Call the optimization solver. Given the initial state, the solver will return the vector of control inputs that minimizes the cost function. The solver used is Ipopt.
    3. Apply the first control input to the vehicle.
    4. Go back to Loop step 1.

In my own words, the MPC method can anticipate future events because we have an idea of what is probably going to happen if we do something. This is because we have a model of how things work in our world (like physics for example). We can anticipate future events based on our current plan of action and also anticipate our next plan of action based on the result of the current plan.

- Kenematic Model
    - So based on *physics,* here is a simplified version of how the world (with our vehicle in it) works. How the state variables get updated based elapse time dt, the current state, and our actuations delta and a.
    - px` = px + v * cos(psi) * dt
    - py` = py + v * sin(psi) (dt)

- psi` = psi + v / Lf * (-delta) * dt (Lf - this is the length from front of vehicle to its Center-of-Gravity)
- v` = v + a * dt
- We can also predict the next cte, and epsi based on our actuations.
- cte` = cte - v * sin(epsi) * dt
- epsi` = epsi + v / Lf * (-delta) * dt

- Cost Function and Penalty Weights
  - Minimize the cross track error cte, we want to be in our desired position
  - Minimize our heaidng error epsi, we want to be oriented to our desired heading
  - If possible, we want to go as fast as we can. I set this to v = 100 but you can play around with this
  - We don't want to steer if we don't really need to
  - We don't want to accelerate or brake if we don't really need to
  - We don't want consecutive steering angles to be too different
  - We don't want consecutive accelerations to be too different
  - cost = A * cte^2 + B * epsi^2 + C * (v - vmax)^2 + D * delta^2 + E * a^2 + F * (a` - a)^2 + G * (delta` - delta)^2
  - A = 1.0, B = 1.0, C = 1.0, D = 1.0, E = 25.0, F = 1000, G = 1200
  - The penalty weights are again determined through trial-and-error, taking into consideration that our primary objective is to drive safely and smoothly.

- Time Step Length and Frequency

- The time step length N is how many states we "look ahead" in the future and the time step frequency dt is how much time we expect environment changes. I chose a dt = 0.05 seconds because the that's the half of the latency between actuation commands so it seemed like a good ballpark (nyquist sampling theorem). If the N is too small, this makes us too short-sighted which defeat the purpose of planning for the future. If the N is too small we might not be able to take advantage of looking ahead to plan for curves that we might not be able to do with simpler and less sophisticated control methods like PID. It doesn't make sense to *look too far to the future* because that future might not be as we expect it, so we must not calculate too much before getting feedback from the environment. I started with N = 6 because that's the number of waypoints given to us but looking at the displayed green line at the simulator makes too short to plan for curves. At N = 15, I noticed that given the time limit of around 0.5 seconds, it seems that the computer was running out of time to find the best variables that have minimized the cost well. With trial-and-error I found that N = 12 was good.

- Model Predictive Control with Latency
  - Note we have to take the 100ms latency into account, so instead of using the state as churned out to as, we compute the state with the delay factored in using our kinematic model before feeding it to the object that will solve for what we should do next.

```
//calculate delayed state
// current state must be in vehicle coordinates with the delay factored in
// kinematic model is at play here
// note that at current state at vehicle coordinates:
// px, py, psi = 0.0, 0.0, 0.0
// note that in vehicle coordinates it is going straight ahead the x-axis
// which means position in vehicle's y-axis does not change
// the steering angle is negative the given value as we have
// as recall that during transformation we rotated all waypoints by -psi
const double current_px = 0.0 + v * dt;
const double current_py = 0.0;
const double current_psi = 0.0 + v * (-delta) / Lf * dt;
const double current_v = v + a * dt;
const double current_cte = cte + v * sin(epsi) * dt;
const double current_epsi = epsi + v * (-delta) / Lf * dt;

const int NUMBER_OF_STATES = 6;
Eigen::VectorXd state(NUMBER_OF_STATES);
state << current_px, current_py, current_psi, current_v, current_cte, current_epsi;

// compute the optimal trajectory
Solution sol = mpc.Solve(state, coeffs);
```