Model Predictive Control

Fabrizio Frigeni, 2017.12.03

Specifications

The goal of this project is to design an MPC controller to drive a car around a track.

The observed variables are current position, orientation and speed of the vehicle. We are also provided with the waypoints that build the ideal path along the middle of the track.

We first need to calculate the actual error, and then generate the optimal trajectory that allows the car to best follow the target path.

The controlled output variables are steering value and throttle value.

Implementation

The controller is implemented in C++. All the project files are saved in the *src* folder.

The important files are:

- MPC.cpp: this is where the core Model Predictive Control class is implemented.
- main.cpp: this is where the observed variables are read, the controller is initialized and called cyclically, and the output variables are sent to the simulator.

Code explanation

The Model Predictive Control is a technique that improves on basic PIDs by injecting knowledge into the controller by means of a model of the process we want to control.

Assuming we provide a good model, the controller will be able to predict the outcome of the chosen sequence of actions and can therefore optimize the selection process.

A PID has no knowledge of the environment and cannot predict future states: it calculates its output action simply based on the present state. On the other hand, an MPC can base its action calculations also on future states, albeit approximated.

Kinematic model

We first build a kinematic model of our vehicle: that is we predict the future state of the car based on its current state and the actions we take.

The state space of our choice includes position, orientation and speed: $[p_x, p_y, \varphi, v]$.

The actions are steering and throttle: [δ, a].

The distance between the center of the car and the front axle is L and it influences the turning rate.

The updated state after a time step t is:

$$\begin{cases} x' = x + vt\cos\varphi \\ y' = y + vt\sin\varphi \\ \varphi' = \varphi + vt\delta/L \\ v' = v + at \end{cases}$$

Similarly, given the current cross-track error (cte) and angular error (eφ) we can also predict their future values:

$$\begin{cases} cte = f(x) - y_0 \\ e\varphi' = \varphi - \varphi_0 \end{cases}$$

$$\begin{cases} cte' = cte + vt\sin(e\varphi) \\ e\varphi' = e\varphi + v\delta t/L \end{cases}$$

Keep in mind that the generated actions are subject to physical constraints:

$$\begin{cases} \delta \in [-25°, 25°] \\ a\epsilon[-1, 1] \end{cases}$$

Polynomial fit

The actual cte and eφ can be calculated as difference between the actual and target values. However, we do not have the complete target path, only selected waypoints.

We use a cubic polynomial to approximately fit a few target points ahead of the car and generate an approximate target path:

```
//fit a cubic polynomial through the path points
Eigen::VectorXd coeffs = polyfit(pointsX, pointsY, 3);
```

Note that the waypoints were initially provided in the global coordinate system and we had to convert them into the vehicle's local coordinate system:

```
//convert path points into car coordinate system
Eigen::VectorXd pointsX(int(ptsx.size())), pointsY(int(ptsy.size()));
for (unsigned int i=0;i<ptsx.size();++i) {
	double x_loc = ptsx[i] - px;
	double y_loc = ptsy[i] - py;
	ptsx[i] = x_loc * cos(psi) + y_loc* sin(psi);
	ptsy[i] = - x_loc * sin(psi) + y_loc * cos(psi);
	pointsX[i] = ptsx[i];
	pointsY[i] = ptsy[i];
}
```

Optimizing the cost function

The MPC selects the best output actions solving an optimization problem: different actions generate different future states, which correspond to different trajectories for our vehicle, and we want to pick the one that minimizes a general error.

The error (or cost) is a function of several parameters. The previously described cte and eφ clearly need to be minimized. But we also want the car to keep moving at a high enough speed, and we might even want to enjoy a smooth ride by minimizing higher derivatives of our actions.

$$J = \sum_{N} w_{cte}(cte - cte_0)^2 + w_{epsi}(e\varphi - e\varphi_0)^2 + \cdots$$

Note that although we optimize the trajectory for a large temporal horizon, we only execute the first step of the new curve, and then we repeat the whole prediction procedure again after we receive the next sensors measurements.

Tuning the weights w_i of the cost function modifies the influence that each parameter has on the planned trajectory. We should ideally give all parameters similar importance, and not let one of them overwhelm the others. Note that what is important is the relative size of the weights, not their absolute values.

We first analyze the range of the parameters around the target state so that we find out the order of magnitude of their attached weights. Then we can fine tune to improve controller behavior.

> cte: 0 .. 100; epsi: 10e-5 .. 10e-3; speed: 10e2 .. 10e4

In other words, the speed will need a much smaller weight than cte, while the epsi will need the highest weights of all.

Based on this observation we select the following weights for good control:

> w_{cte}=10; w_{epsi}= 1000; w_{speed}= 0.01

Using only these values the car swings around considerably trying to follow the best trajectory. A good way to fix this issue is add more constraints to the cost function, limiting the size and the changes (first and second derivative) of the steering action:

> w_{steer}= 10; $w_{steer'}$= 100; $w_{steer''}$= 1000

The driving is now much smoother along the track. Note that by keeping the speed weight relatively small we force the car to slow down during turns so to keep cte and epsi as small as possible.

Prediction horizon

Selecting the length and density of the predicted trajectory is critical.

Our kinematic model is a mere approximation of the real vehicle's behavior, so it makes no sense to predict the trajectory too far ahead in the future.

However, increasing the sampling frequency is very important, because it generates more accurate predictions at the cost of computational expense.

We ended up with a sample time of 0.1s and a total number of 10 points, giving an horizon of 1 second.

Latency

In physical systems the loop action-process-measurement does not happen instantaneously. The action is sent to the actuators, which have their own dynamics and take some time to respond. The process will react accordingly and the measurements are taken thereafter with a specific cycle time. The consequence is that it will take a non-zero time between setting the action and receiving the corresponding feedback. The longer the latency time, the harder the process is to control.

A PID controller bases its output calculation on the current feedback it receives from the sensors. However, if the feedback is slow to come, or analogously, the actions are slow to execute, the PID won't work correctly anymore. Imagine the PID trying to correct an error but not seeing any immediate effect: the result would be a dangerous increase in the output action until saturation, likely causing instability in the system.

On the other hand, a time-latency can be easily modeled and therefore predicted by the MPC. The resulting actions will be chosen with that characteristic already accounted for and the controller will not go crazy for not seeing immediate effects.

A simple way to add latency to the model is simply delaying the starting state we provide to the optimizer:

```
//add latency to actual state
//assuming car moves at constant speed along x axis
double latency = 0.1; //in seconds
px = v * latency;
py = 0;
psi = -v * steering_act / Lf * latency;
v += throttle_act * latency;
cte += v * sin(epsi) * latency;
epsi += psi;
```

Comments

The results using MPC are much better than those we obtained with the simple PID. The kinematic model adds much needed information to the controller, which is now able to predict the outcome of its own actions thereby optimizing them in order to achieve the best approximation of the target trajectory.