CarND-Controls-MPC

Self-Driving Car Engineer Nanodegree Program

Dependencies

- cmake >= 3.5
- All OSes: click here for installation instructions
- make >= 4.1(mac, linux), 3.81(Windows)
  - Linux: make is installed by default on most Linux distros
  - Mac: install Xcode command line tools to get make
  - Windows: Click here for installation instructions
- gcc/g++ >= 5.4
  - Linux: gcc / g++ is installed by default on most Linux distros
  - Mac: same deal as make - [install Xcode command line tools]((https://developer.apple.com/xcode/features/)
  - Windows: recommend using MinGW
- uWebSockets
  - Run either `install-mac.sh` or `install-ubuntu.sh`.
  - If you install from source, checkout to commit `e94b6e1`, i.e.

    ```
    git clone https://github.com/uWebSockets/uWebSockets
    cd uWebSockets
    git checkout e94b6e1
    ```

    Some function signatures have changed in v0.14.x. See this PR for more details.
- **Ipopt and CppAD:** Please refer to this document for installation instructions.
- Eigen. This is already part of the repo so you shouldn't have to worry about it.
- Simulator. You can download these from the releases tab.
- Not a dependency but read the DATA.md for a description of the data sent back from the simulator.

Basic Build Instructions

1. Clone this repo.
2. Make a build directory: `mkdir build && cd build`
3. Compile: `cmake .. && make`
4. Run it: `./mpc`.

Model predictive control

The dynamics of the car are modelled by the state vector (x_t, y_t, v_t, ψ_t), that denote x and y position, velocity and heading.

The state is extended by two variables:

- cte_t: cross-track error: error in between the current y position and the ideal y position (both in the car's coordinate system)
- and $e\psi_t$: error in between the current heading and the ideal heading

The ideal y position is determined by fitting a third order polynomial through the waypoints provided by the path planning algorithm. A least-squares solution (through QR decomposition) is used to calculate this. The ideal y position can then be found by using the current x position in the third order polynomial. The results of this polynomial fit is displayed by the yellow line in the video (using some x points ahead of the car as input to the polynomial).

The ideal heading is determined by the direction coefficient of the tangent of the same third order polynomial at the current x position. The heading can be calculated by $arctan(f'(x))$ where $f(x)$ is the third order polynomial.

The state of the model in the next step is described by the following formula's:

$$
\begin{aligned}
x_{t+1} &= x_t + v_t * cos(\psi_t) * dt \\
y_{t+1} &= y_t + v_t * sin(\psi_t) * dt \\
\psi_{t+1} &= x_t - (v_t / L_f) * \delta_t * dt \\
v_{t+1} &= v_t + \alpha_t * dt \\
cte_{t+1} &= a_0 + a_1 * x_t + a_2 * x_t^2 + a_3 * x_t^3 - y_t + v_t * sin(e\psi_t) * dt \\
e\psi_{t+1} &= \psi_t - arctan(a_1 + 2 * a_2 * x_t + 3 * a_3 * x_t^2) - (v_t / L_f) * \delta_t * dt
\end{aligned}
$$

Note that the sign of the terms containing δ_t have been inverted (equations 3 and 6) due to the fact that a positive value denotes counter-clockwise rotation (turning to the left) in the car's coordinate system, while this denotes a right turn in the simulator.

Since, we want the heading to be in the car's coordinate system, we convert all waypoints and states to the car's coordinate system. This means that each waypoint (w_x, w_y) is transformed from global coordinates $(w_{x,g}, w_{y,g})$ to car coordinates $(w_{x,c}, w_{y,c})$ like this:

$$
\begin{aligned}
w_{x,c} &= (w_{x,g} - c_{x,g}) * cos(-\psi) + (w_{y,g} - c_{y,g}) * sin(-\psi) \\
w_{x,c} &= (w_{x,g} - c_{x,g}) * sin(-\psi) + (w_{y,g} - c_{y,g}) * cos(-\psi)
\end{aligned}
$$

or:

$$
\begin{aligned}
w_{x,c} &= (w_{x,g} - c_{x,g}) * cos(\psi) - (w_{y,g} - c_{y,g}) * sin(\psi) \\
w_{x,c} &= -(w_{x,g} - c_{x,g}) * sin(\psi) + (w_{y,g} - c_{y,g}) * cos(\psi)
\end{aligned}
$$

The (x, y) position of course becomes (0,0) and the heading of the car becomes 0 radians.

The outcome of the algorithm is to find good candidates for δ_t and α_t. Non-linear programming using the Ipopt optimizer) tries to find good candidates based on the aforementioned formula's (which can be converted to constraints by itself, with 0 as lower and upper boundaries), some extra constraints and a cost function that denotes how good the candidates are.

The extra constraints come from physical / technical limitations of the actuators:

- $-0.436332 * L_f <= \delta <= 0.436332 * L_f$ (-0.436332 radians = 25°)
- $-1 <= \alpha <= 1$

The cost function that is used is:

$$J = \sum_t (\lambda_1 cte_t^2 + \lambda_2 e\psi_t^2 + \lambda_3 (v_t - v_{ref})^2 + \lambda_4 \delta_t^2 + \lambda_5 \alpha^2 + \lambda_6 (\delta_{t+1} - \delta_t)^2 + \lambda_7 (\alpha_{t+1} - \alpha_t)^2)$$

The lambda's determine the importance we attach to each part of the cast. The first three lambda's try to minimize the errors with the ideal position, heading and reference velocity. The fourth and fifth lambda's try to minimize the occurrence of large steering angles and accelerations. The last two try to minimize the occurrence of large differences in between subsequent steering angles and accelerations.

The green line that is displayed is based on the predicted (x,y) positions (N in total) that were calculated from the Ipopt optimization. These positions are part of the result vector (stitched after the new model's state) from *MPC::Solve()*.

Time step length

I tried to look one second ahead with ten intervals of 100 ms where the model is evaluated.

Thus: N = 10 and dt = 0.1

The fact that a single step is 100 ms proved also convenient when taking latency into account.

Latency

Since a delay of 100 ms is applied in the program to simulate the latency in the actuator and the latency is of equal to a single time step, I adapted the series of constraints at time steps > 1 :

$$\begin{gathered}
x_{t+1} - (x_t + v_t * cos(\psi_t) * dt) = 0 \\
y_{t+1} - (y_t + v_t * sin(\psi_t) * dt) = 0 \\
\psi_{t+1} - (x_t - (v_t / L_f) * \delta_{t-1} * dt) = 0 \\
v_{t+1} - (v_t + \alpha_{t-1} * dt) = 0 \\
cte_{t+1} - (a_0 + a_1 * x_t + a_2 * x_t^2 + a_3 * x_t^3 - y_t + v_t * sin(e\psi_t) * dt) = 0 \\
e\psi_{t+1} - (\psi_t - arctan(a_1 + 2 * a_2 * x_t + 3 * a_3 * x_t^2) - (v_t / L_f) * \delta_{t-1} * dt) = 0
\end{gathered}$$

To be able to apply the same tactic for time step 1, we retrieve the current steering angle δ_c and throttle α_c from the websocket's JSON stream and provide it to the MPC algorithm. The equations for time step 1 then become:

$$\begin{gathered}
x_1 - (x_0 + v_0 * cos(\psi_0) * dt) = 0 \\
y_1 - (y_0 + v_0 * sin(\psi_0) * dt) = 0 \\
\psi_1 - (x_0 - (v_0 / L_f) * \delta_c * dt) = 0 \\
v_1 - (v_0 + \alpha_c * dt) = 0 \\
cte_1 - (a_0 + a_1 * x_t + a_2 * x_t^2 + a_3 * x_t^3 - y_0 + v_0 * sin(e\psi_t) * dt) = 0 \\
e\psi_1 - (\psi_0 - arctan(a_1 + 2 * a_2 * x_t + 3 * a_3 * x_t^2) - (v_0 / L_f) * \delta_c * dt) = 0
\end{gathered}$$

It would be a possibility to extend the state from the model from 6 variables to 8 variables (including the previous actuator values), but I added these values as parameters to the *MPC::Solve()* function for simplicity.

In case the latency would be higher than a single time step, then it would be necessary to keep a history of the previous δ and α values.

Value of the lambda's

The seven λ_i values were determined by trial-and-error:

- They were initialized so that the errors with the ideal position, heading and reference velocity got a high value (with less focus on the velocity, because the most important factor is that the car doesn't leave the track)
- Also the use of the actuators got some good weights, although these factors are not as important as the first three requirements.
- Then the λ_i values were optimized so that the car follows stays relative close to the yellow line.

There is of course no single ideal set of λ_i values: you can opt for fast tracks for example (stay close to the reference velocity) or you can put more focus on minimising the position and heading errors or even focus on an optimal driving experience (more of a cruising scenario).

In the end, I choose this set of λ_i values (but other sets of values I experimented with, resulted in the good tracks as well):

$(100, 100, 5, 10, 10, 1, 10)$

Video

A video that made use of the optimizer and the set of parameters discussed above, can be found here