Udacity Self-Driving Car Nanodegree

Project 5 - MPC Controller

Contents

1 Reference to Project Code and Files

Here is a link to my *project code and files*. My project includes the following files:

- **Main.cpp** (*link*): to communicate with the simulator via uWebSocket and run the MPC pipeline, as described in the next section.
- **MPC.cpp** (*link*): to define the MPC class, to find optimal actuators (steering and throttle).
- **MPC.h** (*link*): header file for the MPC class.
- **Prm.cpp** (*link*): to define the Prm class, to store global parameters.
- **Prm.h** (*link*): header file for the Prm class.
- **CMakeLists.txt** (*link*): list of files to be compiled (slightly modified from initial version, to incorporate the Prm class)
- **Project_5_writeup.pdf** summarizing the results (*this memo*).

As a result of this implementation, the vehicle is able to steer safely around the track.

2 Model Predictive Control

2.1 Overview

In a nutshell, the MPC model consists in:

- Getting the **target trajectory** to be followed by the car, in the form of waypoints.
- Fitting a **mathematical function** to the target trajectory, in our case a 3rd degree polynomial.
- **Predict** where the car should be at the next time step.
- Find **optimal actuators** to drive accordingly, in our case steering angle and throttle value. This last step is at the core of the model: it consists in solving a non-linear optimization problem.

The model is reevaluated each time the simulator sends data, i.e. at each web socket event. To be able to properly predict the next state, the time elapsed during each event is subdivided into **N discreet time steps**, each with a **dt duration**. The last section of this document discusses the choice of those time steps.

In addition, the model simulates the **latency** between the activation of actuators and their actual impact on the car, as it would occur in real life.

<u>Note</u>: to simplify the problem, we are ignoring variables such as gravity forces, slip angle, slip ratio and tire model, which are not captured in the simulator.

2.2 Pipeline

The pipeline takes place in the **main.cpp** file. It is executed at each web socket event, which roughly takes x seconds on our machine (Windows laptop with Linux VM).

2.2.1 Staging Variables

The simulator provides the **current state** via uWebSockets:

- Actual **waypoints**: 7 (x, y) coordinates forming the trajectory to be followed
  - pts_x: vector of x-coordinates
  - pts_y: vector of y-coordinates

- Car **position**: (x, y) coordinates
  - p_x: x-coordinate
  - p_y: y-coordinate

- Car **heading** ψ: angle with respect to x-axis, in radians, as provided by simulator

- Current car **speed** v: car speed as provided by simulator, after conversion from mph into m/s

- Current **steering** angle: current orientation, in radians, as provided by simulator

- Current **throttle** value: as provided by simulator, in [-1, 1] range.

The current state is converted into **car's coordinates system**, using the following formulas.

- Trajectory, for each waypoint i:
  - $pts\_x_{car,i} = (pts_{x,i} - p_x) * \cos(\psi) + (pts_{y,i} - p_y) * \sin(\psi)$
  - $pts\_y_{car,i} = -(pts_{x,i} - p_x) * \sin(\psi) + (pts_{y,i} - p_y) * \cos(\psi)$

- Car position:
  - $p\_x_{car} = 0$
  - $p\_y_{car} = 0$

- Heading: $\psi_{car} = 0$

- Speed: $v_{car} = v$

2.2.2 Fitting a Polynomial to the Trajectory

A **3rd degree polynomial** is fitted to the trajectory, using **QR decomposition** and **least squares** method (*see link for more details*).

The result is stored in the **coeffs** variable (size 3 vector).

2.2.3 Calculating Cross-Track Error and Heading Error

The cross-track error (cte) and the heading error (eψ) are calculated as following:

- cte: difference between p_y_{car} coordinate and value of polynomial at p_x_{car}
  - $cte = (coeffs_0 + coeffs_1 * p\_x_{car} + coeffs_2 * p\_x_{car}^2 + coeffs_3 * p\_x_{car}^3) - p\_y_{car}$
  - i.e. $cte = coeffs_0$

- eψ: difference between actual and desired heading, i.e. between ψ and angle of tangent to polynomial at at p_x_{car}
  - This step requires calculating the polynomial's derivative, then find the corresponding angle using the inverse tangent function (arctan).
  - $e\psi = \psi - \arctan(coeffs_1 + 2 * coeffs_2 * p\_x_{car} + 3 * coeffs_3 * p\_x_{car}^2)$
  - i.e. $e\psi = \arctan(coeffs_1)$

2.2.4 Calling the Solver

The next step is at the core of the model. A **state vector** is made up of the aforementioned variables:

- $state = [p\_x_{car}, p\_y_{car}, \psi_{car}, v_{car}, cte, e\psi]$
- i.e. $state = [0, 0, 0, v, cte, e\psi]$

This state vector is passed to the **Solve** function, which takes two arguments:

- The state vector (state variable)
- The polynomial coefficients (coeffs variable)

2.2.5 Feeding the Solver

This method leverages the **IPOPT library** (*link*) to find the optimal actuators. It is built as part of the MPC class in the MPC.cpp file, which consists of two main sections. The first section implements **inputs** to the Solve function. As required by the IPOPT library, the cost along with all constraints is stored in one single vector, in our case the **fg** vector.

- The first element of fg is reserved for the **cost**, to be minimized.

- Then we have constraints on **6 state variables**: p_x_{car}, p_y_{car}, ψ_{car}, v_{car}, cte, eψ.
  - Since our timeframe is subdivided into **N time steps**, each state variable requires N elements.

- And constraints on **2 actuators**: δ (steering angle) and a (throttle value). They are stored at the end of fg.
  - Each actuator requires N-1 elements (N-1 actuations for N time steps).

As a result, the size of the fg vector is the following: 1 + 6 * N + 2 * (N-1).

The other input is the **vars** vector, which stores the **values** to be associated with each aforementioned variable. Following the same logic, the size of the vars vector is: 6 * N + 2 * (N-1). This time, no placeholder for the cost.

Constraints are initialized with the current state: x_t, y_t, ψ_t, v_t, cte_t, $e\psi_t$.

Then relationships between two consecutive time steps rely on a **kinematic model**:

- $x_{t+1} = x_t + v_t * \cos(\psi_t) * dt$
- $y_{t+1} = y_t + v_t * \sin(\psi_t) * dt$
- $\psi_{t+1} = \psi_t + v_t * \delta_t/L_f * dt$
- $v_{t+1} = v_t + a * dt$
- $cte_{t+1} = f(x_t) - y_t + (v_t * \sin(e\psi_t) * dt)$
- $e\psi_{t+1} = \psi_t - \psi des_t + v_t * \delta_t/L_f * dt$

- In those equations:
    - L_f is a constant that captures the length between front wheels and the car's center of gravity. It is instantiated as part of the Prm class.

    - f is the polynomial, therefore $f(x_t) = coeffs_0 + coeffs_1 * x_t + coeffs_2 * x_t^2 + coeffs_3 * x_t^3$
      That is where the previously passed polynomial coefficients come into play.

    - $\psi des_t = \arctan(coeffs_1 + 2 * coeffs_2 * x_t + 3 * coeffs_3 * x_t^2)$

The **cost** is comprised of references to the cross-track and heading errors, to a reference speed, to actuations and to consistency between sequential actuations. After being initialized at zero, it is incremented at each time step.

- Cross-track and heading errors: the main purpose of this model is to reduce those errors.

    - cost += $\omega_{cte} * (cte_t - cte_{ref})^2$
    - cost += $\omega_{\psi} * (\psi_t - \psi_{ref})^2$

- Reference speed: this helps approaching a reference velocity as often as possible.

    - cost += $\omega_v * (v_t - v_{ref})^2$

- Current actuations: those operations help minimizing actuations, i.e. driving straight as often as possible.

    - cost += $\omega_\delta * \delta_t^2$
    - cost += $\omega_a * a_t^2$
    - cost += $\omega_{\delta a} * \delta_t^2 * a_t^2$

- Consistency between sequential actuations: those terms avoid excessive zig-zagging and sudden throttle changes.

    - cost += $\omega_{d\delta} * (\delta_{t+1} - \delta_t)^2$
    - cost += $\omega_{da} * (a_{t+1} - a_t)^2$

All ω weights mentioned here above are defined as part of the Prm class. Adjusting them is a subtle exercise, which is discussed in the last section of this memo.

2.2.6 Running the Solver

The second section of the MPC.cpp file implements the **Solve** function. The function initializes variables at zero (besides initial state). Then it sets **boundaries** for state variables and actuators:

- Lower and upper limits for state variables are set to a large number (10^{19} in our case).
- The steering angle δ is bounded between -25 and 25 degrees.
- The throttle value a is constrained between -0.2 (to reflect realistic braking) and 1 (full speed).

Those constants are defined as part of the Prm class.

Then initial values and boundaries are injected in the IPOPT **solve** function, which returns the **solution** variable after solving the optimization problem. Finally, our Solve function returns a 4-element vector:

- N-1 values for the δ steering angle
- N-1 values for the a throttle value
- N x-coordinates for predicted waypoints
- N y-coordinates for predicted waypoints

2.2.7 Sending Actuators to the Simulator

Back in the main.cpp file, the solver's output is captured by the **mpc_output** variable. δ and a actuator values are averaged in order to smooth the driving. Our model allows customizing the number of elements included in this average, by setting the **n_smooth** constant, from the Prm class.

Finally, actuators are passed to the simulator. The steering angle is normalized in the range [-1, 1] as requested by the simulator, by dividing it by the value corresponding to 25 degrees in radians.

In addition, actual and predicted waypoints are sent for **debugging** purposes:

- Actual waypoints are stored by the **next_x_vals** and **next_y_vals** variables, forming a **yellow line** on the video stream.
- Predicted waypoints are captured by the **mpc_x_vals** and **mpx_y_vals** variables and form a **green line**.

2.3 Dealing with Latency

As mentioned earlier, our model simulates real-life latency between actuations and their actual effect, by putting the program on hold for 100ms (sleep_for standard function). The latency is defined by the **latency** constant in the Prm class.

To handle this problem, we update the current state before calling the solver, by **anticipating** what the values will be after a time lag corresponding to the latency. Variables are updated as following:

- Car position:
    - p_x_{car} += v * cos(ψ) * latency
    - p_y_{car} += v * sin(ψ) * latency

- Heading: ψ_{car} += v * δ/L_f * latency

- Speed: v_{car} += a * latency

Then cte and eψ are updated accordingly:

- cte += v * sin(ψ) * latency

- eψ += v * δ/L_f * latency

<u>Note</u>: here above, δ and a represent respectively the current steering angle and throttle value, as provided by the simulator.

3 Discussion

The number of time steps N and the elapsed time between two consecutive time steps has been determined empirically. We found that those settings are highly sensitive to the car’s reference speed. We fixed them in the Prm class as **N = 25** and **dt = 0.125** seconds for a reference speed of **45 mph**.

We also tuned the **cost** empirically, by adjusting different weights with regards to each other. Our final settings are:

-	$\omega_{cte} = 15.0$	→	Weight affected to cross-track error
-	$\omega_{\psi} = 2.0$	→	Weight affected to heading error
-	$\omega_{v} = 1.0$	→	To be close to reference velocity as often as possible
-	$\omega_{\delta} = 1.0$	→	To minimize use of steering wheel
-	$\omega_{a} = 1.0$	→	To minimize use of throttle
-	$\omega_{\delta a} = 300.0$	→	To adjust steering depending on acceleration
-	$\omega_{d\delta} = 20.0$	→	To keep sequential actuations consistent from t to t+1 (steering)
-	$\omega_{da} = 5.0$	→	To keep sequential actuations consistent from t to t+1 (throttle)

Those weights are not perfect and should be improved to reach **higher speed**. They could be adjusted on the fly using the same architecture of Twiddle that we implemented for the PID Control project, relying on “Twiddle checkpoints” (see here).