MPC Controller

Self-Driving Car Nanodegree, Term 2, Project 5

Luc Frachon, February 2018

Introduction

This project is the fifth in term 2 of Udacity's Self-Driving Car Nanodegree. It implements a Model Predictive Controller to drive a simulated car around a 3D track (simulator provided by Udacity and built using the Unity game engine).

1. Controller description

1.1. Inputs and outputs

The code is written in C++ and uses uWebSockets to exchange JSON messages with the simulator. These messages contain information on the vehicle's position, orientation, speed and actuator values. It also exchanges waypoint and landmark positions and representations.

- **Inputs:**

  The program receives the following information from the simulator:

  - Position of the vehicle (x_v, y_v or `px, py` in the `main.cpp` code) in a global Cartesian coordinates system, in meters
  - Orientation of the car (ψ / `psi`) in global coordinates, in radians. Note that two values are actually provided, `psi` and `psi_unity`. The latter uses clockwise orientation and is offset by 90^o. We can safely discard it.
  - Speed of the car (v) relative to the global referential, **in mph**
  - Steering input (δ / `steer_value`) in radians. This value is positive for right-hand turns, negative for left-hand.
  - Throttle input (a / `throttle_value`). This is a scalar value from the $[-1, +1]$ range. -1 corresponds to full braking, +1 to full throttle.
  - Waypoint positions: x_{A_i}, y_{A_i} $for\ i = 1, \dots, n_{waypoints}$ with $n_{waypoints} = 6$ the number of waypoints returned by the simulator at any given time. These waypoints span roughly 100m in distance and are presumed to be chosen based on the vehicle's position around the track (or off track). These are stored in two vectors `ptsx` and `ptsy`.

- **Outputs:**

  The program uses the input data to compute a lowest-cost solution (see next section) and sends it back to the simulator, along with additional information that allow a graphical representation of the ideal trajectory vs solution. The messages sent back to the server contain:

- Optimal steering angle δ to minimize the cost function
- Optimal throttle actuation value a
- Coordinates of the predicted positions of the vehicle with the latest calculated actuation values (`mpc_x_vals`, `mpc_y_vals`), used to graphically represent the solution and expressed in the **vehicle's coordinate system**
- Coordinates of a few points of the fitted polynomial (see next section) (`next_x_val`, `next_y_val`), used to graphically represent the ideal trajectory and expressed in the **vehicle's coordinate system**

1.2. Variable conversions

Before explaining the control process, we need to deal with a technicality. Most of the variables received from or sent back to the simulator cannot be used as such and need to be converted in one way or the other:

- The vehicle's speed is converted from mph to SI units ($m.s^{-1}$):

  $v := v \times 1.609/3.6$

- δ (`steer_value`), as provided by the simulator, is positive for right-hand turns. However this is contrary to the trigonometric convention. Thus: $\delta := -\delta$ before computations, then again just before sending the computed value back to the simulator

- a (`throttle_value`) is the throttle/braking actuation value and is bounded by $[-1, +1]$, but is used as a proxy for the vehicle's acceleration. We therefore need to make sure that it represents actual acceleration as closely as possible. To do this, I measured the time taken by the vehicle to reach 50mph under full throttle from stop, then the time it takes to brake from 50mph to full stop. This gave me approximate values for maximum acceleration (a_{max}) and braking (a_{min}). I used these to map `throttle_value` to acceleration, assuming a piecewise linear relationship, as follows:

  if $a \geq 0$:

  $a := a \times a_{max}$

  else:

  $a := a \times |a_{min}|$ before computations, then:

  if $a \geq 0$:

  $a := a/a_{max}$

  else: $a := a/|a_{min}|$

  just before sending the computed value back to the simulator

  with $a_{min} \approx -7.7m.s^{-1}$ and $a_{max} \approx 3.9m.s^{-1}$

- Waypoints are provided in global coordinates (represented as $x_A^{(g)}, y_A^{(g)}$), however further calculations are performed in the vehicle's coordinate system ($x_A^{(v)}, y_A^{(v)}$). The conversion is done through the following equations:

$$x_A^{(v)} = (x_A^{(g)} - x_v)\cos(\psi) + (y_A^{(g)} - y_v)\sin(psi)$$
$$y_A^{(v)} = -(x_A^{(g)} - x_v)\sin(\psi) + (y_A^{(g)} - y_v)\cos(psi)$$

  There is no need to convert these back to global coordinates after computations, as the simulator expects vehicle coordinates for the points used when plotting the ideal and the predicted trajectories.

1.3. Control Process

The aim of the MPC is to find the best actuator inputs given a state vector and a trajectory we would like the vehicle to follow.

The first thing we need to decide is the length of the prediction time steps (dt) and the number of time steps over which to make prediction (N). **These prediction time steps are not to be confused with the simulator's**. Every simulator time step, the algorithm computes N prediction time steps (i.e. looks $N \times dt$ seconds ahead in time). Although we will only keep the first time step solution for actuator inputs and discard the other $N-1$, they are essential in ensuring that our solution is compatible with future desired states of the vehicle.

There are four parts to the control process. First, we determine an approximation of the ideal trajectory that the vehicle should follow. Then we build a cost function that we will need to minimize. A low cost function means that the vehicle is following the "ideal" line at the specified target speed. Thirdly, we define a vehicle motion model using the state vector provided by the simulator and use it to define update rules from one time step to the next. We can then use a non-linear optimization algorithm to compute the lowest-cost solution for this time step. Finally, we feed this solution back to the simulator for execution.

1.3.1. Trajectory approximation

At each time step, the server provides the next 6 waypoints in global coordinates. After conversion (see above), we use the function `polyfit()` (provided by Udacity) to fit a 3rd degree polynomial to these waypoints. 3rd degree polynomials have one inflexion point, which means they are able to handle turns in both directions. They are also have only 4 parameters, so are a good compromise for accuracy v. efficiency. The resulting polynomial has the form:

$$fit(x) = coeff_0 + coeff_1 \times x + coeff_2 \times x^2 + coeff_3 \times x^3$$

1.3.2. Cost function

The cost function quantifies "how well" a computed solution follows the fitted trajectory. We can also use it to describe other aspects of a good solution. Intuitively, we would like a vehicle to 1) be safe, i.e. stay on the track, 2) be efficient, i.e. match a certain target speed and 3) offer a comfortable ride and not exceed the tires' grip limit, i.e. as smooth inputs as possible.

- The vehicle's position error is given by the cross-track error (or CTE) square and is defined as:
  $cte(t)^2 = [fit(x_v^{(v)}(t)) - y_v^{(v)}(t)]^2 = fit(0)^2$ because the vehicle's coordinates in its own coordinate system are (0, 0)
- The vehicle's orientation error is given by:
$$\begin{aligned} e_\psi(t)^2 =& [\psi(t) - \psi_{desired}(t)]^2 \\ =& [\psi(t) - arctan(\frac{\partial fit}{\partial x}(x,t))]^2 \\ =& [\psi(t) - arctan(coeff_1)]^2 \end{aligned}$$
  because $\frac{\partial fit}{\partial x}(x,t)) = coeff_1 + 2\,coeff_2 \times x + 3\,coeff_3 \times x^2 = coeff_1$ at $x = x_v^{(v)} = 0$.
- The error on the vehicle's speed is given by $e_v(t)^2 = (v(t) - v_{ref})^2$ where v_{ref} is the target speed (both in m.s^{-1}).
- The smoothness of inputs is determined by two values for each actuation input:
$$\begin{aligned} Input\,magnitude &: \delta(t)^2,\ a(t)^2 \\ Magnitude\,of\,input's\,derivative &: e_{d\delta}(t+1)^2 = (\delta(t+1) - \delta(t))^2,\ e_{da}(t+1)^2 = (a(t+1) - a(t))^2 \end{aligned}$$

To code these features into the cost function, I used a similar code to what was provided in the MPC lesson quizzes but with weights assigned to each element of the function (as in the walkthrough video). The cost function has therefore the following form, where $k_0, \ldots, k_6$ are tunable parameters and the $\widehat{\ }$ symbol denotes predictions associated with the candidate solution S:

$$
\begin{aligned}
cost(S) = & k_0 \sum_{t=0}^{N} \widehat{cte}(t)^2 + k_1 \sum_{t=0}^{N} \widehat{e_\psi}(t)^2 + k_2 \sum_{t=0}^{N} \widehat{e_v}(t)^2 \\
& + k_3 \sum_{t=0}^{N-1} \hat{\delta}(t)^2 + k_4 \sum_{t=0}^{N-1} \hat{a}(t)^2 \\
& + k_5 \sum_{t=0}^{N-2} \widehat{e_{d\delta}}(t+1)^2 + k_6 \sum_{t=0}^{N-2} \widehat{e_{da}}(t+1)^2
\end{aligned}
$$

There are three things to note here:

- This value can only be computed once all prediction time steps are known for a candidate solution. In practice, the optimizer will run many candidates through this cost function and retain the one that generates the lowest cost. The search for the minimum cost is performed every time the program received new data from the simulator, as long as the computation times make this possible.
- The vehicle's speed v is measured relative to the global referential, but expressed in the vehicle's coordinate system. In other words, the coordinate system is defined with its origin at the initial (i.e. actual) position of the vehicle at the start of each round of computations, and its x-axis parallel to the direction of travel, but it is fixed relative to the global environment. Positions and speeds in predicted time steps are then expressed in relation to the initial point.
- Actuation inputs $\hat{\delta}$ and $\hat{a}$ are only calculated by the optimizer up to $t = N - 1$, hence the different upper bounds for the summations involving these terms in the cost function equation.

1.3.3. Update equations and optimizer constraints

To model the vehicle's behavior and the updates to the state vector, we use the bicycle model: The state vector of the vehicle includes:

- Its position (x_v, y_v) - note that in the vehicle coordinate system, the initial (actual) position is always $(0, 0)$
- Its orientation ψ - again, initially equals 0
- Its speed v, relative to the global referential but expressed in the coordinate system defined by the vehicle's true position

In addition to these physical measures, we also pass the error values to the optimizer, because they are used in the cost function and change over time steps. Predicting their future values is therefore essential.

- Cross-track error cte
- Orientation error e_ψ

All these values depend on the time step t.

From the point of view of the optimization problem, implementing the model involves telling the optimizer how these variables are allowed to vary from one time step to the next. These rules are passed as constraints that specify that subsequent state vectors must follow the update equations below:

$$
\begin{aligned}
x_v(t+1) &= x_v(t) + v(t)\cos(\psi(t)).\,dt \\
y_v(t+1) &= y_v(t) + v(t)\sin(\psi(t)).\,dt \\
\psi(t+1) &= \psi(t) + \frac{v(t)}{L_f}\delta(t).\,dt \quad with\ L_f \approx 2.67m \quad a\ geometrical\ constant\ caracterizing\ the\ vehicle \\
e_\psi(t+1) &= \psi(t) - \psi_{desired}(t) + \frac{v(t)}{L_f}\delta(t).\,dt \\
cte(t+1) &= fit(x_v(t)) - y_v(t) + v(t)\sin(e_\psi(t)).\,dt
\end{aligned}
$$

In practice, Ipopt (the non-linear optimization library used in this project) needs these constraints to be passed as:

$C_x(0) = x_v(0) \quad , \quad C_x(0)\ bounded\ by\ [x_v(0), x_v(0)]$

$C_x(t+1) = x_v(t+1) - x_v(t) + v(t)\cos(\psi(t)).\,dt \quad , \quad C_x\ bounded\ by\ [0,0] \quad for \quad t = 0, \ldots, N-1$

and so on for the other four equations. Each constraint at each time step is thus assigned a lower bound and an upper bound which are all exactly 0, with the exception of the initial state, which receives the corresponding value from the state vector.

Lower and upper bounds are also defined for the state variables themselves to ensure that the solutions found by Ipopt are realistic. For x_v, y_v, ψ, v, we simply define very low minimum and very high maximum values, simply to avoid overflows. For δ and a, however, we use the physical model's limitations. The vehicle can steer 25^o in both directions and, as previously mentioned, has acceleration capabilities between $a_{min} \approx -7.7m.\,s^{-1}$ and $a_{max} \approx 3.9m.\,s^{-1}$. Therefore we set:

$\delta \in [-25\pi/180, +25\pi/180] \quad and \quad a \in [a_{min}, a_{max}]$

1.3.4. Solving the optimization problem

At this point, we have the following optimization problem:

$$\min_S cost(S) \quad subject\ to \quad \{constraints,\ bounds\}$$

with constraints and bounds defined above.

This problem is passed to the Ipopt solver and it returns a solution vector that contains, among othe7r information, the optimal actuation inputs $\hat{\delta}$ and $\hat{a}$, as well as the predicted values for the state vector, for each of the computed time steps. As far as actuation inputs are concerned, we are only interested in the very first time step so we discard the rest. However, we collect $\widehat{x_v}$ and $\widehat{y_v}$ for all time steps in order to be able to plot them in the simulator.

1.3.5. Implementing the solution

$\hat{\delta}$ and $\hat{a}$ are then passed to the simulator for execution. The projected positions of the vehicle are also passed for graphical representation. We also plot the polynomial fitted to the next 6 waypoints for comparison.

The vehicle steers and accelerates as instructed, modifying its trajectory. A new simulator time step is generated and a new state variable is input into our model.

1.4. Latency

1.4.1. Principle

In real life, a vehicle does not react instantly to actuation inputs and by the time it does, its state has already changed compared to the state that was used to calculate these inputs. This can cause erratic behaviors. To simulate this effect, this project introduces a 100ms latency between the output of a solution and its execution.

1.4.2. Dealing with latency

A simple way to handle latencies is to transform the initial state vector and replace it with a prediction after the latency period. Concretely, at the start of each round of computations (i.e. when a new simulator time step begins), instead of feeding the optimizer with the actual vehicle state ($x_c(0), y_c(0), \psi(0), v(0)$), we predict its value at time $t = latency$ assuming constant acceleration $a(0)$ and steering $\delta(0)$. The corresponding equations are:

$$
\begin{aligned}
x_c(0) &:= x_c(0) + v(0)\cos(\psi(0)) \times latency \\
y_c(0) &:= y_c(0) + v(0)\sin(\psi(0)) \times latency \\
\psi(0) &:= \psi(0) + v(0)\frac{\delta(0)}{L_f} \times latency \\
v(0) &:= v(0) + a(0) \times latency
\end{aligned}
$$

This is obviously an approximation; for instance when calculating x_c and y_c, we assume that both v and ψ remain constant, which is only true a and δ are both zero. But we assume that these changes are second order and latency is sufficiently small for the approximation to hold. Note also that when $latency = 0$, all the variables remain unchanged.

These new values are then used as starting points by the optimizer and the rest of the process is exactly as described above.

2. Model tuning

The model has many parameters that can be tuned:

- N (number of prediction time steps)
- dt (size of prediction time steps)
- v_{ref} (target speed) -- although so much a parameter as a goal, it has a great influence on the model's reliability
- $k_0, \ldots, k_6$ (weights of each term of the cost function)

As is often the case, they are all inter-related and one cannot drastically change one parameter value without revising the others.

2.1. Role of each parameter

I tested many combinations of parameters, both with and without latency and my conclusions with regards to their respective influence on the model's behavior are as follows:

- N: Perhaps contrary to intuition, increasing the number of prediction time steps does not improve the model. I found that 7 works best. Values of 5 or less do not work, maybe because the optimizer needs a minimum number of data points to be able to fit a stable solution. Values above 10 also generate instability. This could be due to the increase computation requirement that forces the model to skip frames. More probably, a distant prediction horizon reduces the relative weight of the closest trajectory points (which are going to be executed) to the benefit of points further away (which we will discard and re-compute anyway).
- dt: I found that dt does not need to be very small, despite the assumption we made in section 1.4.2. Values between 0.15 and 0.20s generate mostly stable solutions. This true both with and without latency, but it can be argued that higher values are especially useful with latency because they counter-balance it: The next time step calculation is already based on a predicted state that would occur later than the latency value.
  At v_{ref} increases above 80mph, the value of dt needs to be adjusted slightly, simply due to the greater distance travelled in the same amount of time. For instance, with my final model, $dt = 0.13s$ is fully stable at $v_{ref} = 100mph$, whereas $dt = 0.2s$ is not (note that the car does not actually reach 100mph but stabilizes around 92.5mph).
- v_{ref}: In my experience, a good set of parameters (especially N and dt) offer great flexibility with regards to the target speed. My final set of parameters works for speeds up to 100mph and possibly higher (I did not run any further tests). However, with a sub-optimal parameter set, a working model will rapidly fail as the target speed is increased.
- k_0, k_1 (cross-track and orientation error weights): Increasing these parameters increases the emphasis of following the fitted trajectory over smoothness. While this is usually desirable, it can cause over-corrections when the car moves away from the center of the track (e.g. during a sharp bend). These corrections can then be over-corrected themselves, causing increasingly large oscillations until the car leaves the track. High values for these parameters must therefore be matched with high values of the steering dampening coefficients (k_3, k_5, see below).
- k_2, k_4 (speed error, magnitude of acceleration): These value needs to be set in relation to each other. A high value of k_4 will penalize large throttle application, which can prevent the car from going near its target speed. This can be balanced by increasing k_2. On the other hand, a low value of k_4 can help by allowing the car to brake harder if required, but also makes speed changes more sudden which can cause issues with the predictions because the model assumes a constant speed when computing new x_c and y_c values.
- k_3, k_5 (magnitude of steering input, derivative of steering angle): As explained above, these values act as dampening factors for the cross-track error and the orientation. They can prevent the amplifying oscillations that can occur with high values of k_0 and k_1.
- k_6 (derivative of acceleration): This has fairly minor effects on the model. I was not very concerned about smoothness in throttle application although with a more complete vehicle model including a tire model, we would definitely need to be. I therefore use a value of 1.

Finally, what matters is the relative value of the k parameters. Therefore increasing one is equivalent to decreasing all others.

2.2. Final model parameters and results

My final model uses the following parameters:

- $N = 7$
- $dt = 0.20$ (decrease to 0.13 at $v_ref = 100mph$)
- $v_{ref} = 70 \times 1.609/3.6$ (i.e. 70mph, but this can be adjusted as mentioned before)
- $k_0 = k_1 = k_2 = k_3 = k_5 = 10$
- $k_4 = 5$
- $k_6 = 1$

As mentioned before, the vehicle successfully navigates the track at all speeds up to 100mph -- and possibly higher, but I did not run such tests.

3. Discussion

This project taught me the fundamentals of an MPC controller through an interesting and appealing application. The model works well and is fairly robust to different target speeds. However it is likely to be difficult to generalize for the following reasons:

- The model is very sensitive to its parameters N and dt. To obtain a stable controller, I had to use a fairly low value for N and a fairly high value for dt. While this is perfectly reasonable in a very controller environment such as a track with no sharp turn, in real life this might be a limitation. In a changing environment with traffic, pedestrians, crossings, traffic lights etc., the ability to react faster and plan further ahead may well be a requirement.
- With the simple bicycle model that we used, and the simple physics of the simulator, the car is able to go around the track at over 92mph barely slowing down in corners. In real life, tire physics must be taken into account at high speed, and the model needs to be able to brake before corners to avoid accidents. A more complex car model would be useful, but it obviously comes with more expensive calculations and additional parameters to tune.

Within the setup of this project, there are nonetheless some possible ideas to improve upon this particular implementation. The first would be to further refine the weight parameters. I did not spend a lot of time fine-tuning these as their effect on the model is much smaller than N and dt, however there might be some weight values that allow the use of smaller dt values , thus mitigating the first limitation mentioned above.

Another ideas would be to add a constraint on the admissible values of cte. The distance to the kerbs is around 2.3m each side of the car when it is standing in the exact middle of the track, so just like we constrain the values of δ, we can force the optimizer to find only solutions where cte does not exceeds these values.

I tried implementing this very quickly and it did seem to help at 80mph, but the can ran wide at 100mph even though it was driving safely before I added this constraint. I did not have the time to further investigate, but it is not inconceivable that the added constraint means more calculations, and therefore skipped frames at high speed.

Appendix: Development environment

Code developed on an i5-4440 x64 3.10GHz, quad-core CPU running MS Windows 10 Pro. The code was written and tested within the Windows Ubuntu Bash (16.04). The text editor used is Sublime Text.