# Term 2 Project 5 – Model Predictive Controller – Writeup

**David Chrzanowski**

---

## Model Overview:

### 1. Data Read in from Simulator

The controller begins by reading data in from the simulator. This includes the universal x and y coordinates of waypoints, the 4 elements of the car's state vector (x and y position, heading, velocity), and the current position of the car's 2 actuators (accelerator and steering angle).

### 2. Fitting of Waypoints and Error Calculation

The waypoints are fit to a 3$^{rd}$ order polynomial using the polyfit function provided in the starter code. This polynomial is then used to calculate the Cross Track Error (cte) and Heading Error (epsi) associated with the current state.

### 3. Step Forward for Latency

The current state (and vehicle motion model + actuations see equations in step 4 below) is used to predict the future state – 100 miliseconds later – after the known latency has passed. This ensures that the optimizer is solving for actuations that are appropriate for the state the car will be in when those actuations are actually applied.

### 4. Optimize Actuations

The vehicle "state" (stepped forward per previous step), along with model equations, and a cost function definition are fed to the ipopt optimizer. The optimizer projects the vehicle's state N (20) timesteps (0.1 s each) into the future, and finds a throttle and steering angle actuation for each timestep that minimize the value of the cost function.

Model equations:

```
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] / Lf * delta[t] * dt
v_[t+1] = v[t] + a[t] * dt
```

```
cte[t+1] = f(x[t]) - y[t] + v[t] * sin(epsi[t]) * dt
epsi[t+1] = psi[t] - psides[t] + v[t] * delta[t] / Lf * dt
```

Cost function:

```
// Cost is stored in first element of fg vector
fg[0] = 0.0;

// The part of the cost based on the reference state.
for (int t = 0; t < N; t++) {
  fg[0] += W_CTE * CppAD::pow(vars[cte_start + t], 2);
  fg[0] += W_EPSI * CppAD::pow(vars[epsi_start + t], 2);
  fg[0] += W_V * CppAD::pow(vars[v_start + t] - ref_v, 2);
}
// Minimize the use of actuators.
for (int t = 0; t < N - 1; t++) {
  fg[0] += W_DELTA * CppAD::pow(vars[delta_start + t], 2);
  fg[0] += W_A * CppAD::pow(vars[a_start + t], 2);
}
// Minimize the value gap between sequential actuations.
for (int t = 0; t < N - 2; t++) {
  fg[0] += W_DDELTA * CppAD::pow(vars[delta_start + t + 1] - vars[delta_start + t], 2);
  fg[0] += W_DA * CppAD::pow(vars[a_start + t + 1] - vars[a_start + t], 2);
```

The weights of the cost function (W_********) are used to tune the optimization by relatively scaling the various elements.

Ultimately – only the FIRST actuation elements are returned from the optimization.

5. **Apply Results**

The returned optimized steering angle and throttle actuations are given to the simulator for activation.

6. **Plot Waypoints and Model Projection**

The waypoints (ideal path of the car) are given to the simulator to be plotted in yellow, and the optimized projected state at each forward timestep (N – 20 in my case – states) is plotted in green. This assists greatly both in tuning of the model's cost function, as well as visualizing the "Model Predictive" aspect of "Model Predictive Control." The process then repeats.

## Timestep Length and Elapsed Duration:

In the lessons it was mentioned that a timestep of 0.1 seconds was realistic for real world self-driving car applications, so I stuck with that. I began with an N of 20 because it seemed foolish to project any less than 2 seconds ahead of the car's current state, but that still seemed like a lot of computations and I didn't want to have too many.

I ended up deciding to tune the target velocity and cost function weights rather than to play around with these values. In retrospect, I believe that with a higher target velocity I would likely need a higher N or dt or both to project more than 2 seconds into the future.

## Polynomial Fitting and MPC Preprocessing:

There were several places where the data required adjustment or preprocessing:

- The velocity value provided by the simulator seemed to be in mph, but the equations of motion require m/s. I simply change this as soon as it is read in from the simulator.

```
double v_mph = j[1]["speed"];
double v = v_mph * 0.447;
```

- The waypoints read in from the simulator are in universal coordinates, but I chose to convert them into the vehicle's coordinate frame based on its current state. I do this before fitting the polynomial:

```
for(int i = 0; i < n; i++) {
    const double dx = ptsx[i] - px;
    const double dy = ptsy[i] - py;
    x_vehicle[i] = dx * cos(-psi) - dy * sin(-psi);
    y_vehicle[i] = dy * cos(-psi) + dx * sin(-psi);
  }
```

  This method came with the added benefit that calculation of CTE becomes very simple – it is simply the intercept (or 0-indexed coefficient) of the fit polynomial.

  The vehicle state is translated into the local vehicle coordinate frame during the step-forward for latency described below.

- On the advice of the Tips and Tricks in the project – the steering angle provided by the optimizer needs to be negated and normalized to [-1,1] before being sent back to the simulator:

```
msgJson["steering_angle"] = -steer_value/deg2rad(25);
```

## Adjusting MPC for Latency

As described in step 3 of the model overview, I chose to correct for this by stepping the vehicle's state forward 100 miliseconds (the known latency) using the established vehicle model. This forward state is fed to the optimizer as the actual current state – so that by the time the actuation occurs, it is an actuation for the appropriate reality. This step simultaneously serves as the opportunity to translate the vehicle's state value (provided by the simulator in local coordinates into local vehicle coordinates.

```
const double px_stepforward = v * dt;
const double py_stepforward = 0;
const double psi_stepforward = - v * steering_angle * dt / Lf;
const double v_stepforward = v + throttle * dt;
const double cte_stepforward = cte + v * sin(epsi) * dt;
const double epsi_stepforward = epsi + psi_stepforward;
```

Where dt is the latency (100 ms) and Lf is the distance from the car's CoM to the single point of steering (approximately the front axle).