MPC Controller

*Write up of work done for Udacity CarND Term 2, Model Predictive Controller (MPC) Project, by **Mani Srinivasan***

Project Objectives

The goals / steps of this project are the following:

1. Clone/fork the project's template files from the **project repository**.
2. Choose the state, input(s), dynamics, constraints and implement MPC.
3. Test the solution on basic examples.
4. Test the solution on the simulator!
5. When the vehicle is able to drive successfully around the track, submit!
6. Try to see how fast you get the vehicle to **SAFELY** go!

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

---

1. Compiling

Your code should compile.

Code compiles without errors with cmake and make.

See the output of the cmake and make steps below.

```
Manis-MacBook-Pro:Submitted srnimani$ mkdir build
Manis-MacBook-Pro:Submitted srnimani$ cd build
Manis-MacBook-Pro:build srnimani$ cmake .. && make
-- The C compiler identification is AppleClang 8.1.0.8020042
-- The CXX compiler identification is AppleClang 8.1.0.8020042
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/srnimani/Desktop/Udacity/CarND/Term2/MPC Controller/Submitted/build
Scanning dependencies of target mpc
[ 33%] Building CXX object CMakeFiles/mpc.dir/src/MPC.cpp.o
[ 66%] Building CXX object CMakeFiles/mpc.dir/src/main.cpp.o
[100%] Linking CXX executable mpc
[100%] Built target mpc
Manis-MacBook-Pro:build srnimani$
```

2. Implementation

The Model

Student describes their model in detail. This includes the state, actuators and update equations.

The Model Predictive Controller implemented in this project closely follows what was taught in the course. I have applied the concept and taken help from the quiz as well as the solution walk thru' by Udacity instructors Aaron Brown et al.

The steps followed were:

1. Set the time-step and duration (*N* and *dt)*.
2. Fit the polynomial to the waypoints.
3. Calculate initial cross track error and orientation error values.
4. Define the components of the cost function (state, actuators, etc.). You may use the methods previously discussed or make up something, up to you!
5. Define the model constraints. These are the state update equations defined in the *Vehicle Models* module.

Time-step Length and Elapsed Duration (N & dt)

Selecting the time-step and elapsed duration looks straight forward but was tricky. I initially chose a higher number of time step length (100) and smaller duration (0.01 sec). I just could not get the car to turn around immediately after the bridge, although' computational overhead was not an issue. I saw the car straightening as predicted by the long line and going off the track. After a few trial and error, I settled for a length of 20 steps and a delta of 0.01.

Polynomial Fitting and MPC Pre-processing

A polynomial was fitted to waypoints as described in the course. It uses the same algorithm as in the function, with was adapted from

https://github.com/JuliaMath/Polynomials.jl/blob/master/src/Polynomials.jl#L676-L716

I converted the way point coordinates from the original reference to the one based on the car position. This meant shifting the x and y coordinates to align the car along the x-axis which will mean that the angle will be '0'. The code to do that is below:

```
for (int i =0; i < n; i++)
{
    double shift_x  = ptsx[i] - px;
    double shift_y  = ptsy[i] - py;

    // Transalte to car coordinates by shifting the angle, now the x
    // axis is aligned towards the car direction

    ptsx[i]         = (shift_x * cos(0-psi) - shift_y * sin(0-psi));
    ptsy[i]         = (shift_x * sin(0-psi) + shift_y * cos(0-psi));
}
```

This simplifies the calculations, as well and the car reference state is translated to the one below:

```
Eigen::VectorXd state(6);
state << 0, 0, 0, v, cte, epsi;
```

where x, y and psi are all '0'.

Model Predictive Control with Latency

To account for latency, I have modelled a simple dynamic system and incorporated into the vehicle model. This system simulates the position and angle of the car at a future point in time while calculating the actuator values. The following code implements this model.

```
// Account for latency
const double latency = 0.1;  // 100 ms
const double Lf = 2.67;

px = v * latency;
psi = - v * steer_angle / Lf * latency ;
```

This latency changes the initial state of the car as follows:

```
Eigen::VectorXd state(6);
state << px, 0, psi, v, cte, epsi;
```

The latency does affect the constants which are used for reducing the cost, especially the ones related to *cte* and *psi*.

3. Simulation

The vehicle successfully drives multiple laps around the track.

The car successfully drives around the track, with the maximum speed set to 80 mph. I have left the simulator running for more than 30 minutes without the car going off the track. It is important to note that there is no background task running on the computer while running the simulation, as it affects the simulator behavior. Even capturing the screen to record the video affects the way the car goes around. With a powerful processor and more memory, I guess the simulator should run fine without crashing.

The following is the link to the YouTube video showing 3 laps of the car on the simulator.

https://youtu.be/0xEdRiOU5s8
