MPC Control Project

Introduction

Using Model Predictive Control (MPC), this project involves writing a C++ program that can drive a simulated car around a virtual track using specific waypoints from the track itself. The simulated car's actuators have a 100ms latency (delay) that must be accounted for as well as part of the MPC calculation.

Compilation

The programs that have been written to accomplish the project are: src/MPC.cpp, and src/main.cpp. In addition, file CMakeLists.txt had a little of necessary modification.

This project made a subfolder 'build' in the project folder, and start compiling by typing the following:

```
cmake .. && make
```

Then, the C++ code is compiled without errors.

Implementation

- **The Model**

The kinematic model includes the vehicle's x and y coordinates, orientation angle (psi), and velocity, as well as the cross-track error and psi error (epsi). Actuator outputs are acceleration and delta (steering angle). The model combines the state and actuations from the previous timestep to calculate the state for the current timestep based on the equations below:

$$x_{t+1} = x_t + v_t * cos(\psi_t) * dt$$

$$y_{t+1} = y_t + v_t * sin(\psi_t) * dt$$

$$\psi_{t+1} = \psi_t + \frac{v_t}{L_f} * \delta_t * dt$$

$$v_{t+1} = v_t + a_t * dt$$

$$cte_{t+1} = f(x_t) - y_t + (v_t * sin(e\psi_t) * dt)$$

$$e\psi_{t+1} = \psi_t - \psi des_t + (\frac{v_t}{L_f} * \delta_t * dt)$$

- **Timestep Length and Elapsed Duration (N & dt)**:

In my program MPC.cpp, the values chosen for N and dt are 10 and 0.1, respectively. Admittedly, this was at the suggestion of Udacity's Q&A. These values mean that the optimizer is considering a one-second duration (called prediction horizon) in which to determine a corrective trajectory. Maybe at first, it seems like having a larger N value would be nice, but it's kind of more computationally difficult for the system because as it would go further out in time and makes a cost function had to work harder too. Adjusting either N or dt (even by small amounts) often produced erratic behavior. Other values tried by me in the simulator including 20 / 0.1, 10 / 0.05, and some others.

- **Polynomial Fitting and MPC Preprocessing**

The waypoints provided by the simulator are transformed to the car coordinate system (at /src/main.cpp from line 97 to 107). Then a 3rd-degree polynomial is fitted to the transformed waypoints. These polynomial coefficients are used to calculate the cte and epsi later on. They are used by the solver as well to create a reference trajectory.

- **Model Predictive Control with Latency**

To handle actuator latency, the state values are calculated using the model and the delay interval. These values are used instead of the initial one. The code implementing that could be found at /src/main.cpp from line 133 to line 143.

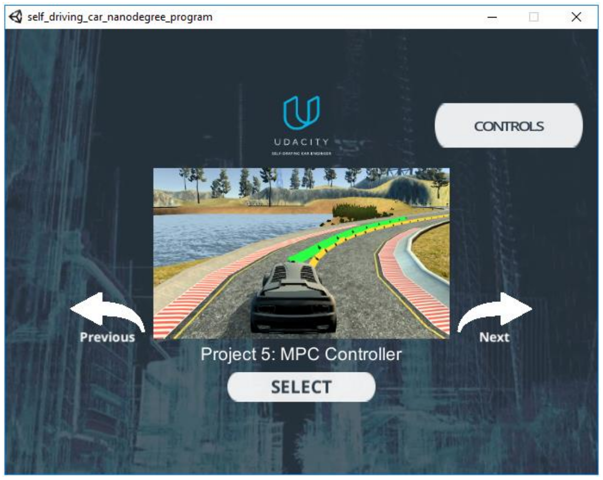
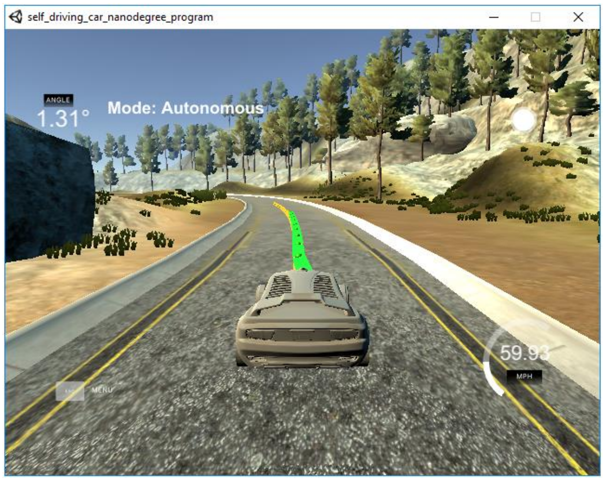
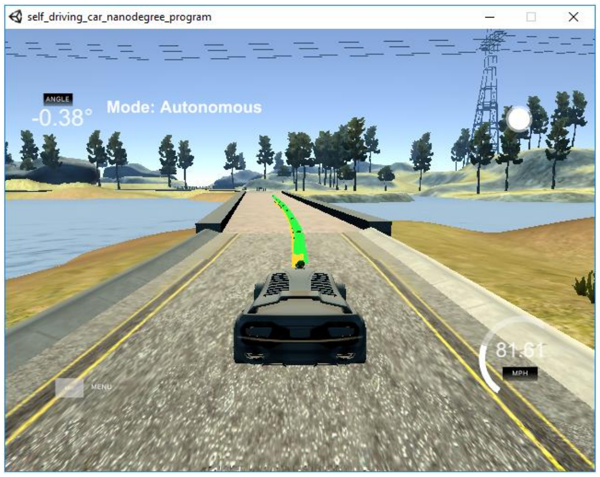
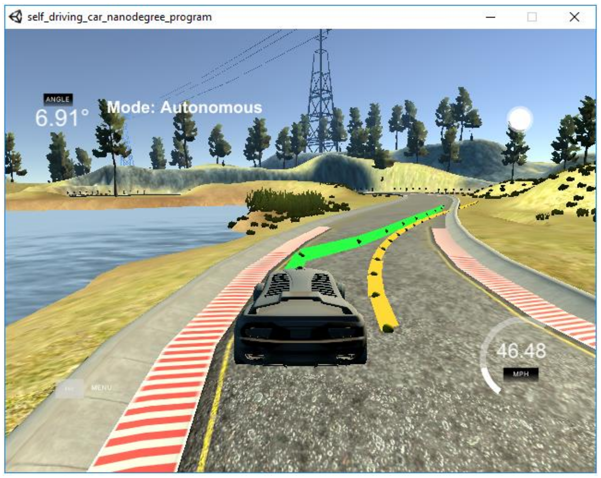
Simulation

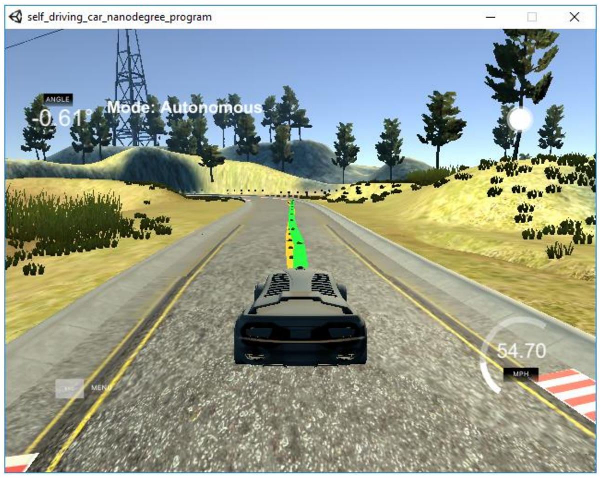
After the C++ code is compiled by command 'cmake .. && make', following command could be input:

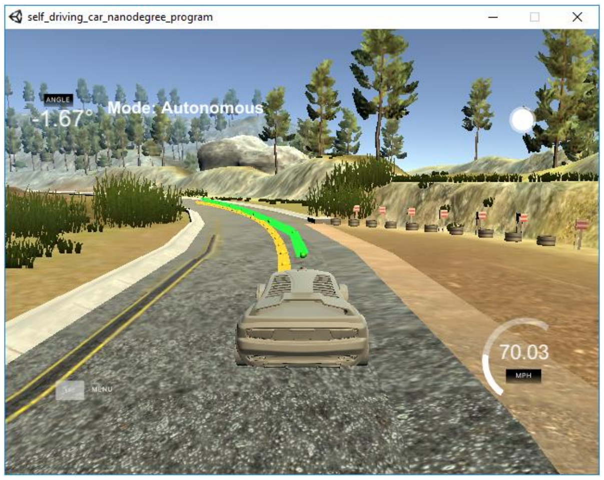
```
./mpc
```

Now the MPC controller is running and 'listening to port 4567' for messages from the simulator. Next step is to open Udacity's simulator. Using the right arrow, you need to go to the 'Project 5: MPC Controller'. Click the "SELECT" button, and the car starts driving. You will see the debugging information on the MPC controller terminal.

The pictures below show the demonstration on MPC Controller in the simulator.

 self_driving_car_nanodegree_program UDACITY CONTROLS Previous Next Project 5: MPC Controller SELECT	 self_driving_car_nanodegree_program ANGLE 1.31° Mode: Autonomous 59.93 MPH
 self_driving_car_nanodegree_program ANGLE -0.38° Mode: Autonomous 81.61 MPH	 self_driving_car_nanodegree_program ANGLE 6.91° Mode: Autonomous 46.48 MPH





The vehicle successfully driven a lap around the track, and no tire leave the drivable portion of the track surface. The car didn't pop up onto ledges or roll over any surfaces.

Conclusion

It is very interesting to compare the results using the MPC Controller with the same car on the same track using a PID controller. The PID controller can be tuned to specific sections of track. But, it tends to fail on other aspects of the track, becoming grossly unstable.

The MPC approach is far more adaptive. At each point in time that actuator values are updated, an entire collection of future states are computed analytically and optimized using a solver that has a cost function referencing a specific desired trajectory. In general, the MPC controller gives better composite results on the track. It has myriad more controllability in terms of bounds and setpoints than the PID controller.