carnd_t2_p5_mpc_reproject

Carnd - term 2 - project 5 - mpc project

Overview

The goal of this project is to build a Model Predictive Control (MPC) model to drive the vehicle around the track. The model processes a series of sensor data (ptsx,ptsy,x,y,psi,v,cte,epsi) to track the reference path and to predict the vehichle drive path. IPOPT library is used to optimize the control inputs. Similar to pervious projects in Term2, the MPC model is connected to a simulator via uWebSocketIO with waypoints of the reference path.

Project Repository

All resource are located in Udacity's project repository CarND-MPC-Project (https://github.com/udacity/CarND-MPC-Project)

Project Submission

All modified code including results are committed to my personal github page carnd_t2_p5_mpc_project (https://github.com/chriskcheung/carnd_t2_p5_mpc_project)

Key Files

main.cpp

establishes communication between simulator and PF model using uWebSocketIO, and reads in data during set time intervals. Coordinates transform from global coordinate system to vehicle coordinates is needed for the way points of the reference path to be useful for finding the coefficient of line fitting using polyfit(). Sensor measurements and coefficient of fitted line are fed to MPC::Solve() for predicting and optimizing the actuation inputs.

MPC.h and MPC.cpp

contains 2 main classes: FG_eval and MPC.

FG_eval mainly manages the constraints of the key error parameters and setting the cost function and each error weight base to allow tuning: FG_eval::operator(). MPC vehicle model is implemented as part of the constraint for finding the future x,y coordinates.

MPC sets up the upper and lower bound of the actuator input constraints and rest of the state inputs and calls the optimization function to obtain the best solution the actuator inputs.

Implementation Challenge

Handling latency

In the idle world, MPC provides a instant actuation control prediction with sensor telemetry input. However, in the real world, the predicted actuation control takes time to propagate through the machanical parts to reflect to the actual turnning. To deal with latency, I choose to update the state vector elements to determine the future position of px, py, and the vehicle orientation psi in the delta time of 100ms. The delayed state vectors is then used for transforming the waypoints to coordinate. This provides that the transformed coordinates are representing the delayed referring point of px, and py.

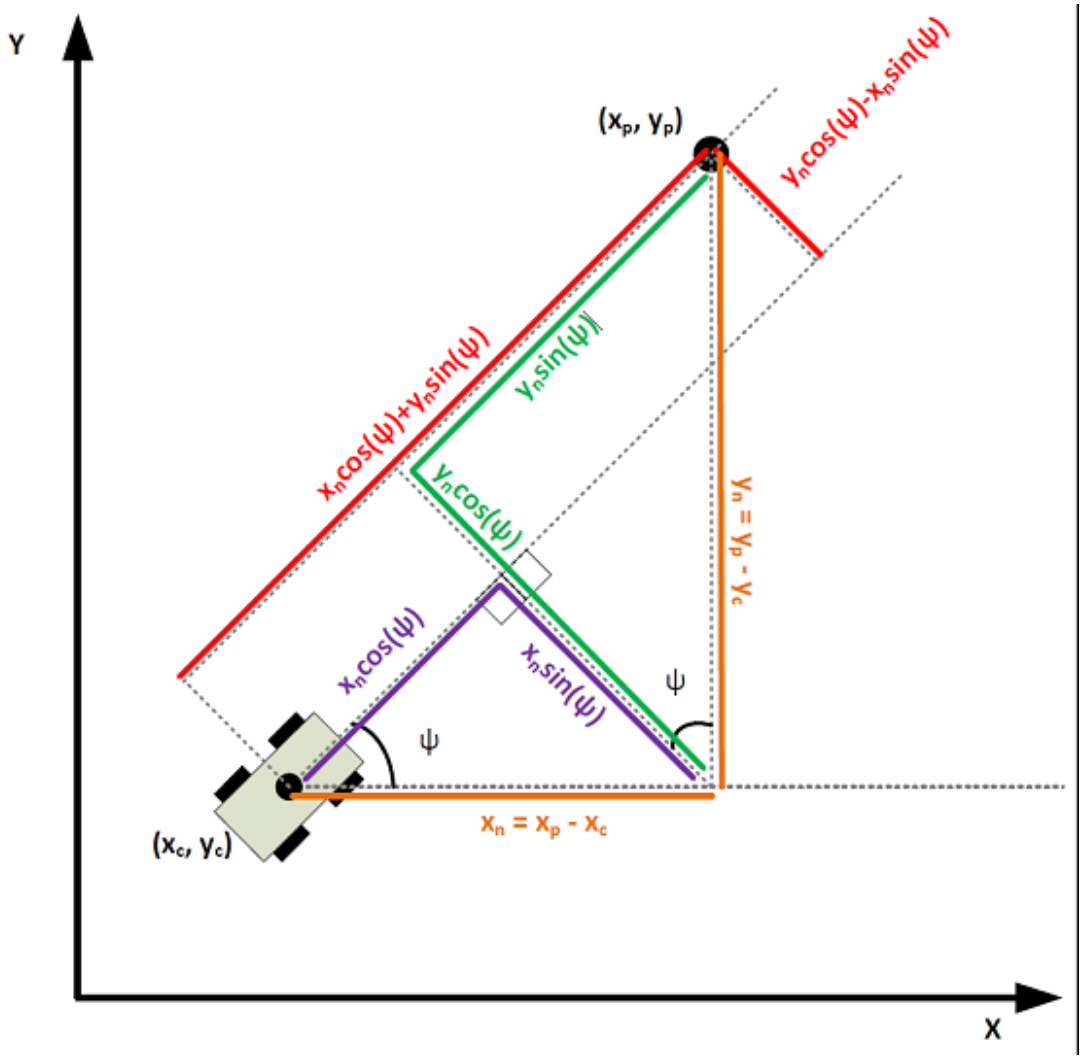
The original solution updated the state vectors with MPC model equations with dt=100ms before feeding into polyfit, hoping the coefficents of fitted curve would reflect the latency. However, the solution didn't totally work with my AWS remote VM setup while running simulator locally in my laptop. Some none zero network latency may have contributed to the inputs telemetries from simulator, which throw off the same MPC model.

In my latest solution, I handled the latency after polyfit but before MPC.solve(), with the assumption that px, py and psi were reset to 0 due to transforming to vehicle coordinates, as if vehicle was the new origin. The result seemed much better this time.

```
    // To deal with 100ms latency between current state and the actual a
ctuation
    // control to be reflected to the vehicle, update the state vector u
sing
    // dt=latency before passing state vector to mpc.Solve() the new pre
dicted state
    // include the error prediction for 100ms latency
    // Note: after transforming coordinates, px, py and psi are reset to
 0
    double dt = latency*0.001;  // in millisecond
    double Lf = 2.67;
    //     px1 = 0.0 + cos(0)*v*dt;
    double px1 = v*dt;
    //     py1 = 0.0 + sin(0)*v*dt;
    double py1 = 0.0;
    //     psi1 = 0.0 - v*delta*dt/Lf;
    double psi1 = 0.0 - v*delta*dt/Lf;
    double cte1 = cte + v*sin(epsi)*dt;
    double epsi1 = epsi - v*delta*dt/Lf;
```

Setting up the parameters for fitting

The sensor telemetry provides a series of way points for x,y coordinates for reference path. These way points are measured based on global coordinates. To predict the vehicle actuations, we need to transform the way points to vehicle coordinates. The telemetry provided the angel of the vehicle orientation (psi in radian) in the global coordinates.



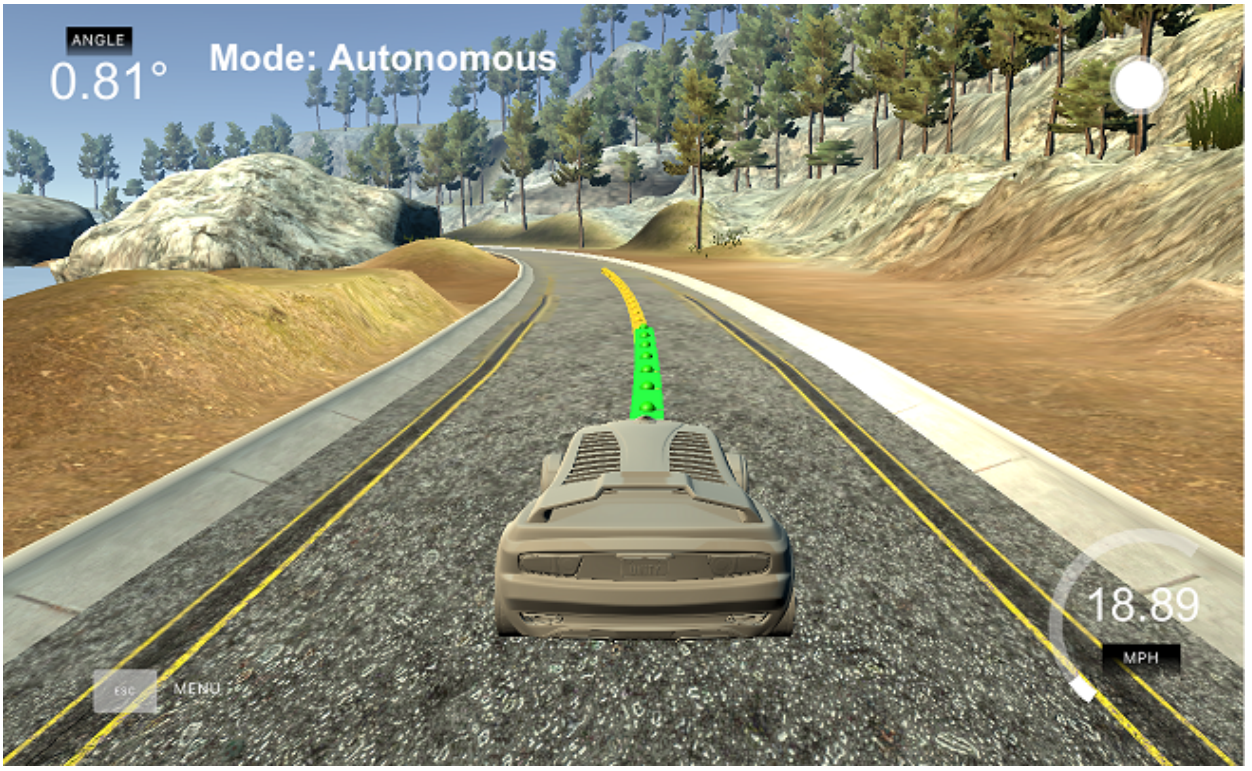
To transform the way points, turn them all to the right by the angle of psi.

```
for (int i=0; i<int(ptsx.size()); i++) {
    double x_car = ptsx[i] - px;
    double y_car = ptsy[i] - py;
    ptsx_car[i] = x_car*cos(psi)+y_car*sin(psi);
    ptsy_car[i] = y_car*cos(psi)-x_car*sin(psi);
}
```

Use the newly transformed way points for line fitting.

```
auto coeffs = polyfit(ptsx_transform, ptsy_transform, 3);
```

This provides a fitted line in the direction same as the vehicle as shown below.



Determine CTE and Error of PSI

The coefficient found from the fitted line is used to determine the cross track error by feeding it to polyeval(). polyeval() evaluate the result of function f(x) with the coefficients of the differnt degree.

```
for (int i = 0; i < coeffs.size(); i++) {
result += coeffs[i] * pow(x, i);
}
```

The error of psi, epsi, is the different between psi and arctangent of derivative of f(x).

```
    double epsi = psi - atan(coeffs[1]) + 2*coeff[2]*px + 3*coeffx[3]*px
*px;
```

AS we transform the waypoints to the vehicle coordinates based on (x,y) from telemetry, we can assume (x,y) is the new origin of the vehicle at the time being. Therefore, (x,y) = (0,0) applies to the determining the cte and epsi in vehicle's prespective. Same goes to the input state vector for the MPC.solve().

```
double cte = polyeval(coeffs, 0);
double epsi = - atan(coeffs[1]);
//state << px, py, psi, v, cte, epsi;
state << 0, 0, 0, v, cte, epsi;
auto vars = mpc.Solve(state, coeffs);
```

References, Constraints and Cost

We start with the following reference as our goal is to find the smallest error as possible. Setting the reference cte and epsi to zero. For now, I settled ref_v to 80 for project submission. ref_v defines the medium velocity the model will push to.

```
    // reference value of each error constraints that we care
    const double ref_cte = 0;
    const double ref_epsi = 0;
    const double ref_v = 80*0.44704;  // in meter per second format for
 consistency
```x1

In FG_eval, cost of the constraints are stored in the first element in F
G_eval.fg vector. Cost includes the weight of error elements that are in
 consideration during optimization. The goal is to establish a cost func
tion that can minimize the error for cte, epsi, v, delta of actuation an
gle, and acceleration. The heavier the weight, the more it gets penalize
 in the MPC solver function, which will lead to smaller error to the coo
responding cost element.
```c++
    // The cost is stored is the first element of `fg`.
    // Any additions to the cost should also be added to `fg[0]`.
    fg[0] = 0;

    // Reference State Cost
    // TODO: Define the cost related the reference state and
    // any anything you think may be beneficial.
    for (int i=0; i<int(N); i+=1){
        fg[0] += cnstr_weight_cte  * CppAD::pow(vars[cte_start+i]-ref_ct
e,   2);
        fg[0] += cnstr_weight_epsi * CppAD::pow(vars[epsi_start+i]-ref_e
psi, 2);
        fg[0] += cnstr_weight_v    * CppAD::pow(vars[v_start+i]-ref_v,
     2);
    }

    // minimize the use of actuators
    for (int i=0; i<int(N)-1; i+=1){
        fg[0] += cnstr_weight_delta * CppAD::pow(vars[delta_start+i],2);
        fg[0] += cnstr_weight_a     * CppAD::pow(vars[a_start+i],    2);
    }

    // minimize the gap between each sequential actuations
    for (int i=0; i<int(N)-2; i+=1){
        fg[0] += cnstr_weight_delta_gap * CppAD::pow(vars[delta_start+i+
1]-vars[delta_start+i], 2);
        fg[0] += cnstr_weight_a_gap     * CppAD::pow(vars[a_start+i+1]-v
ars[a_start+i],         2);
    }
```

Constraints of each state vectors are governing by the MPC vehicle model equations. Each state vector constraints are defined N number of times base on the timesteps. The MPC model

```
    // Recall the equations for the 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] / 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
'''
```

is used to define the constraint state elements. The ultimate goal is to find the value where the delta between state[t+1] and state[t] are as minimal or equal to zero as possible. The code below shows how each state vector constraints are defined.

```c++
    for (int t = 1; t < int(N); t++) {
      AD<double> x1 = vars[x_start + t];
      AD<double> y1 = vars[y_start + t];
      AD<double> psi1 = vars[psi_start + t];
      AD<double> v1 = vars[v_start + t];
      AD<double> cte1 = vars[cte_start + t];
      AD<double> epsi1 = vars[epsi_start + t];

      AD<double> x0 = vars[x_start + t - 1];
      AD<double> y0 = vars[y_start + t - 1];
      AD<double> psi0 = vars[psi_start + t -1];
      AD<double> v0 = vars[v_start + t - 1];
      AD<double> cte0 = vars[cte_start + t - 1];
      AD<double> epsi0 = vars[epsi_start + t - 1];

      // Only consider the actuation at time t.
      AD<double> delta0 = vars[delta_start + t - 1];
      AD<double> a0 = vars[a_start + t - 1];
      AD<double> f0 = coeffs[0] + coeffs[1]*x0 + coeffs[2]*CppAD::pow(x0
,2) + coeffs[3]*CppAD::pow(x0,3);
      AD<double> psides0 = CppAD::atan(coeffs[1] + 2*coeffs[2]*x0 + 3*co
effs[3]*CppAD::pow(x0,2));

      // Recall the equations for the 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] / 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
      // constraint equals the delta between state[t+1] and state[t] to
 be as minimum or equal to zero as possible
      fg[1 + x_start    + t] = x1    - (x0 + v0 * CppAD::cos(psi0) * dt
);
      fg[1 + y_start    + t] = y1    - (y0 + v0 * CppAD::sin(psi0) * dt
);
      fg[1 + psi_start  + t] = psi1  - (psi0 - v0 * delta0 / Lf * dt);
```

```
      fg[1 + v_start    + t] = v1    - (v0 + a0 * dt);
      fg[1 + cte_start  + t] = cte1  - ((f0 - y0) + (v0 * CppAD::sin(eps
i0) * dt));
      fg[1 + epsi_start + t] = epsi1 - ((psi0 - psides0) - v0 * delta0 /
 Lf * dt);
    }
```

The upper and lower bound of the constraints and results are also necessary to be set before calling MPC.solve().

```
// Initial value of the independent variables.
// SHOULD BE 0 besides initial state.
Dvector vars(n_vars);
for (int i = 0; i < int(n_vars); i++) {
    vars[i] = 0;
}
// Set the initial variable values
vars[x_start] = state[0];
vars[y_start] = state[1];
vars[psi_start] = state[2];
vars[v_start] = state[3];
vars[cte_start] = state[4];
vars[epsi_start] = state[5];

Dvector vars_lowerbound(n_vars);
Dvector vars_upperbound(n_vars);
// TODO: Set lower and upper limits for variables.
// for state variables
for (int i = 0; i < int(delta_start); i++) {
    vars_lowerbound[i] = -1.0e19;
    vars_upperbound[i] = 1.0e19;
}
// for delta actuator, constraint within [-deg2rad(25),deg2rad(25)]
for (int i = delta_start; i < int(a_start); i++) {
    vars_lowerbound[i] = -0.436332*Lf;
    vars_upperbound[i] = 0.436332*Lf;
}
// for a actuator, constraint within [-1,1]
for (int i = a_start; i < int(n_vars); i++) {
    vars_lowerbound[i] = -1.0;
    vars_upperbound[i] = 1.0;
}

// Lower and upper limits for the constraints
// Should be 0 besides initial state.
Dvector constraints_lowerbound(n_states);
Dvector constraints_upperbound(n_states);
// for state variables
for (int i = 0; i < int(n_states); i++) {
    constraints_lowerbound[i] = 0;
    constraints_upperbound[i] = 0;
}
constraints_lowerbound[x_start] = state[0];
constraints_lowerbound[y_start] = state[1];
constraints_lowerbound[psi_start] = state[2];
constraints_lowerbound[v_start] = state[3];
constraints_lowerbound[cte_start] = state[4];
constraints_lowerbound[epsi_start] = state[5];

constraints_upperbound[x_start] = state[0];
```

```
        constraints_upperbound[y_start] = state[1];
        constraints_upperbound[psi_start] = state[2];
        constraints_upperbound[v_start] = state[3];
        constraints_upperbound[cte_start] = state[4];
        constraints_upperbound[epsi_start] = state[5];
```

Result

Tuning the following parameters are the key to success in this project.

Choice of N: N defineds the number of timesteps we want to predict ahead of current state. If N is too large, the model will be penalized by long computational time during IPOPT due to larger constraint entries as well longer iteration to performing line searchs for finding the optimal point. We just need enough of timesteps for our prediction.

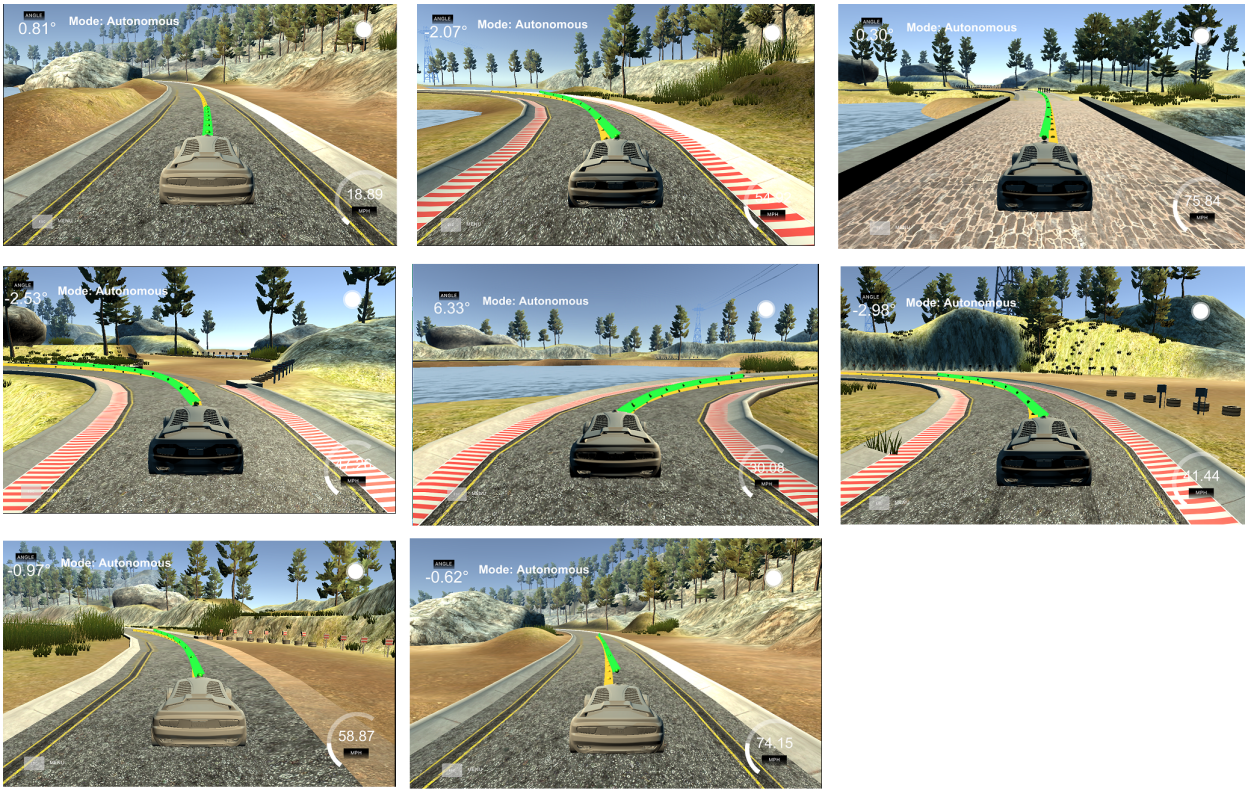
Choice of dt: dt defines the delta time between each time steps N. The smaller it is, the finer the resolution it is. However, if the delta time is to small, that means the timesteps will be too close to each other and they may loose their characteristic, which means they may all look the same which is not a good representation for prediction.

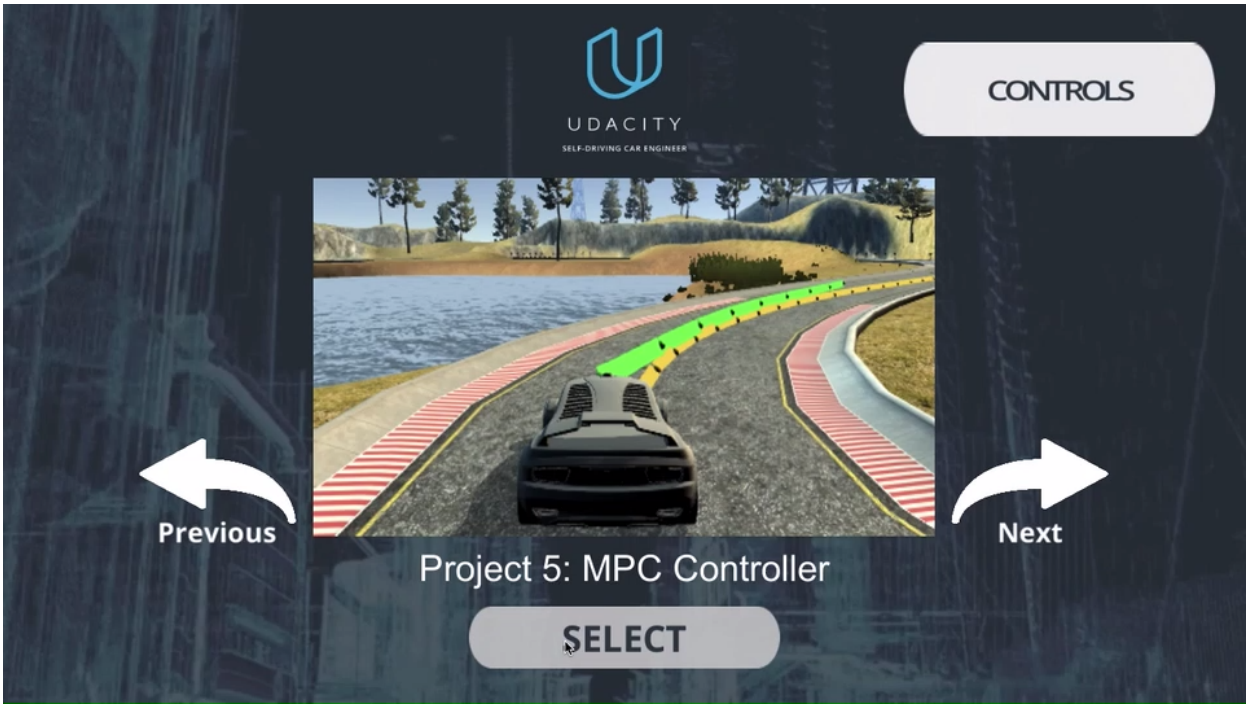
Choose of N*dt: I started with N=8 and dt=0.05. However, when dt is smaller than 0.1, the vehicle would start swinging left and right even before reaching the first turn. When dt is larger than 0.5, it seems the vehicle stalled in between time steps. The stall becomes more obviouse when N is above 15, which is due to long computational time. After couple iteration of trial, dt=0.1 seems to works the best with N=10.

```
        // TODO: Set the timestep length and duration
        size_t N = 10;
        double dt = 0.1;

        // reference value of each error constraints that we care
        const double ref_cte = 0;
        const double ref_epsi = 0;
        const double ref_v = 120;

        // weight distribution of each error constraint, the heavier distrib
    ution the more we care about that constraint
        double cnstr_weight_v = 1.0;
        double cnstr_weight_cte = 4000.0;
        double cnstr_weight_epsi = 4000.0;
        double cnstr_weight_delta = 5.0;
        double cnstr_weight_a = 5.0;
        double cnstr_weight_delta_gap = 400.0;
        double cnstr_weight_a_gap = 10.0;
```

I pushed ref_v to 120 in order to get vehicle goes as fast as above 92+ mph. The solution seemed to be better this way. When I lowered ref_v to 100, vehicle got a bit jerky around the corner, which I suspect the IPOPT optimizer got to closer to the specified constraints and caused shape change during optimization. Padding extra room on ref_v makes the optimizer keep trying harder to get close to the constraint even when it reaches its maximum, around 92+. I increased the constraint weight of

cte and epsi to 4000 to amplify the penality for large cte and epsi. The weight of delta_gap is increased moderately to avoid the turn update in between each timestep will not be too big to avoid sudden turn. The final setting demostrated vehicle stayed on track for multiple laps.





0:00 / 1:35

In [ ]: