Reflections on Project5: Model Predictive Controller (MPC):
Link to video on dropbox:
https://www.dropbox.com/s/4r0mcywfy6duoyq/VID_20171115_012944.mp4?dl=0

The Model:

The basic idea behind MPC is to convert the controller problem to a mathematical optimization problem, where we try to minimize the error between the path that the self-driving care should follow and the predicted location and orientation of the car.

The output of the solver function (optimizer), finds the set of actuations, steering angle and acceleration needed to minimize the "error" and hence follow the path planned for the car.

The model starts by indentifying the current state of the car. That is defined as location (x,y), Orientation psi, velocity v, cross track error cte which is distance from defined path for the car. And finally orientation error epsi.

Using Kinematics laws, we can estimate a prediction for the future states of an object/ car, using the below formulas:

- px = px + v*cos(psi)*dt;
- py = py + v*sin(psi)*dt;
- psi = psi + v*delta/Lf*dt;
- v = v + acceleration*dt;
- epsi = epsi + v * -delta/Lf *dt;
- cte = cte + v * sin(epsi) * dt;

These equations formulate model constrains for the solver function, with upder and lower boundries of zero.

The actuations components of the model are acceleration/Deceleration and steering (delta). Therefore we have to comeup with a cost function and work on solving for reducing the cost by adjusting the acutations. And different weights should be used with the components of most importance to have a suitable cost function to solve to. And below is the cost functions nd weights I used in my code.

The first loop account for errors in CTE, orientation error epsi, and velocity v (penalizing deviating from a certain refernce velocity, I used 20 km/H for safe driving ☺. and second loop penalizes steering frequently and accelerating / decelerating and the last loop penalizes the sudden steering / sudden throttle, etc.

```
fg[0] = 0;
   for (int i = 0; i < N; i++){
     fg[0] += 10* CppAD::pow(vars[cte_start+i],2);
```

```
    fg[0] += CppAD::pow(vars[epsi_start+i],2);
    fg[0] += CppAD::pow(vars[v_start+i] - ref_v,2);
  }

  for (int i =0 ; i<N-1; i++){
    fg[0] += 2000 * CppAD::pow(vars[delta_start+i],2);
    fg[0] += CppAD::pow(vars[a_start+i],2);
  }
      for (int i =0 ; i <N-2; i++){
    fg[0] += 1500* CppAD::pow(vars[delta_start+i+1] - vars[delta_start+i],2);
    fg[0] += 100*CppAD::pow(vars[a_start+i+1] - vars[a_start + i],2);
  }
```

Finally we use an optimizer called IPOPT, which forces us to formulate the data in a certain way, and spits out a solution. We use the first set of actuation parameters, steering and throttle, and discard the rest, and run the process and the next point again. We also use the x,y points prediceted to plotted for this project.

Although I implemented the code for plotting the lines, but for some reasons it doesn't appear on the screen.

Timestep Length and Elapsed Duration (N & dt)

I played around with many numbers for this parameter. But the basic intuition I had was to try to minimize the predicted time horizon, as it is expected that the environment will change constantly, and any predictions beyond 2 or 3 seconds might be not help us to find a suitable solution. So I focused on having a total time horizon T of 1 or 2 seconds

After adjusting the cost gains to reach satisfying performance of the self driving car on track, I started playing with the "dt" paramter. I realized the more I increase it, the poorer the performance of the car gets. It actually starts adopting a timid driving behaviour correcting the path. But if I drive it lower then I get a smoother technology.

In this case I fixed N=20, and tested dt = 0.1 and dt = 0.05, and the 0.05 had a smoother driving behaviour on the track

Polynomial Fitting and MPC preprocessing

I noticed from many comments on the forum that switching all coordinates from map to car coordinates is a better solution, and it turned out to be so.

```
for (int i = 0; i < ptsx.size(); i++){
        double x = ptsx[i] - px;
        double y = ptsy[i] - py;
        ptsx_[i] = x * cos(-psi) - y * sin(-psi);
        ptsy_[i]= x * sin(-psi) + y * cos(-psi);
```

```
}
```

However with such change, I had to do some preprocessing to see the world from the eyes of the car coordinate

- Vehicle state is updated. It is now the origin from which we see the world.
- The starting orientation of the car is considered head on the x-axis, no angles

After that I can do the polynomial fitting using the car coordinates, and calculate the cte according, as the perpendicular distance f(x) from the line, at least at the start of the movement before any turning.

As specified in the lecture notes, the steering angle is reversed to fit the simulator condition.

Model Predictive Control Latency

To incorporate the latency in the model, I updated the state to account for the future, i.e. in 100 msec, using the kinematics roles (the motion model). In the following code, you will see how I combined latency, and conditioning for coordinate change to car coordinates in the following lines:

```
double delta = j[1]["steering_angle"];
double acceleration = j[1]["throttle"];
double latency = 0.1;
double px_latency = 0.0 + 1.0 * latency;
double py_latency = 0.0;
double psi_latency = 0.0 + v * -delta / Lf * latency;
v = v + acceleration*latency;

// Fit a polynomial to provided waypoints from simulator
auto coeffs = polyfit(ptsx_, ptsy_, 3);

// Calculate the Cross Track Error cte
double cte = polyeval(coeffs, 0);

// Orientation error
double epsi= - atan(coeffs[1]);
double epsi_latency = epsi + v * -delta/Lf *latency;
double cte_latency = cte + v * sin(epsi) * latency;
```