Question:

- For my video, at beginning why there is a gap at straight road? (The MPC trajectory is predicted close to the center line, so the adjustment is little? ,how to improve this)
- I don't find effective way to tune the parameter of the cost functions. Would you please give some suggestions?
- To implement in real vehicle what kind of vehicle model can be used?
- If I want to simulate in CARSIM, but cannot get the constraints of the vehicle model used in the software, is it possible?

```
122        Eigen::MatrixXd waypoints = TransformGlobleToLocal(px, py, psi, ptsx, ptsy);
123        Eigen::VectorXd local_ptsx = waypoints.row(0);
124        Eigen::VectorXd local_ptsy = waypoints.row(1);
125
126        // fit polyniminal
127        Eigen::VectorXd coeffs = polyfit(local_ptsx, local_ptsy, 3);
128        double cte = polyeval(coeffs, 0);
129        double epsi = -atan(coeffs[1]); // x=0 position
130
131       //consoder delay 0.1
132        double delay_t = 0.1;
133        double x_delay = 0 + v * cos(psi) * delay_t;
```

REQUIRED

As you mention in line 129, x=0 in car coordinate, so is psi but you're still using psi from global coordinate which can be large number despite the car is driving in straight line.

I don't get your point here, I think I have already transfer to vehicle coordinate.

CarModel:

The car model is a bicycle model which consists of 6 variables: [x,y,psi,v,cte,epsi]. They are x and y vehicle position, vehicle orientation, vehicle speed, cross track error and orientation error. To measure the constrains, the model has been transformed to the following equations.

```
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); //modify turn direction as mentioned
fg[1 + v_start + t] = v1 - (v0 + a0 * dt);
fg[1 + cte_start + t] = cte1 - ((f0 - y0) + v0 * CppAD::sin(epsi0)*dt);
fg[1 + epsi_start + t] = epsi1 - ((psi0 - psides0) - v0 * delta0 / Lf * dt); //modify turn direction as mentioned
```

Lf measures the distance between the front of the vehicle and its mass center. f0 is the evaluation of the polynomial f at point x0 and psides0 is the tangential angle of the polynomial f evaluated at x0.

Timestep Length and Elapsed Duration (N & dt)

The number N determines the number of variables optmized by the controller. So, higher N will result in extra computational cost. Here is the code for time consuming.

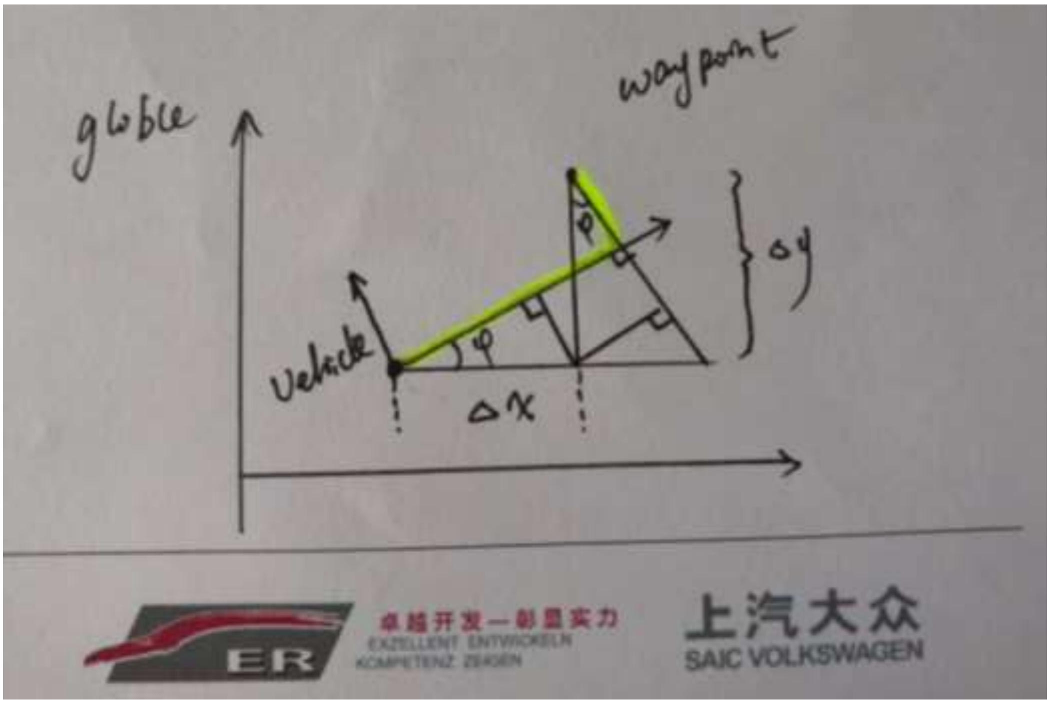
```
start = clock();
count++;
vector<double> result = mpc.Solve(state, coeffs);
stop = clock();
durationTime += ((double)(stop - start)) / CLOCKS_PER_SEC;
if (count == 10){
    std::cout << "Time consume" << durationTime << std::endl;
}
```

N	dt	Time consume(10 times)	Path follow stable
5	0.1	0.146	Unable to follow
10	0.1	0.284	yes
15	0.1	0.416	Unable to follow
20	0.1	1.57	Unable to follow
10	0.05	0.264	Yes but less stable
10	0.15	0.4137	Yes but less stable
10	0.2	0.628	Yes but less stable
5	0.2	0.13	Unable to follow

It was found that if N * dt is too low, the vehicle will oscillate around the lane center. If N * dt is too high, the prediction period is too long and the vehicle may leave the sharp turn.
If N is too high, the computational time increase, and so long time may lead to unable to follow. if N is too low, the MPC curve becomes too volatile for the car to be able to drive.
The influence of dt is not observed that clear.

The conclusion above may also be affected by other parameters I choose, like the parameters of cost functions and the speed to follow. I think iterature review is needed.

Polynomial Fitting and MPC Preprocessing
Before fitting the path returned from the simulator, we have to transfer the waypoints from global position to vehicle coordinate.




```
Eigen::MatrixXd TransformGlobleToLocal(double x, double y, double psi, const vector<double> &ptsx, const vect
    assert(ptsx.size() == ptsy.size());
    int len = ptsx.size();
    Eigen::MatrixXd waypoints(2, len);
    for (int i = 0; i < len; i++) {
        waypoints(0, i) = cos(psi) * (ptsx[i] - x) + sin(psi) * (ptsy[i] - y);
        waypoints(1, i) = -sin(psi) * (ptsx[i] - x) + cos(psi) * (ptsy[i] - y);
    }
    return waypoints;

}
```

Model Predictive Control with Latency

Latency was considered in the main function. The next state was predicted before calling the MPC solve by using vehicle model.

```
//consoder delay 0.1
 double delay_t = 0.1;
 double x_delay = 0 + v * cos(psi) * delay_t;
 double y_delay = 0 + v * sin(psi) * delay_t;
 double psi_delay = 0 - v * delta / 2.67 * delay_t; //modify turn directic
 double v_delay = v + a * delay_t;
 double cte_delay =(cte - 0) + v * sin(epsi)*delay_t;
 double epsi_delay=epsi - v * delta / 2.67 * delay_t;

 Eigen::VectorXd state(6);
 state << x_delay, y_delay, psi_delay, v_delay, cte_delay, epsi_delay;
```