Model Predictive Control (MPC)

1) **Compilation**
    a. Code compiles and builds.
    b. Source code modifications were made in main.cpp and mpc.cpp.
    c. The was done on Windows 10 using Bash.
    d. Ipopt 3.12.9 was installed.
2) **Implementation**
    a. Description of the Model in details
        i. The state model has 4 parameters, the 2-dimensional position, px, py, the orientation of the velocity, ψ , and the velocity, v.
        ii. The model also has control inputs, the steering angle, δ, and the linear acceleration, a.
        iii. The kinematics describes the evolution of the state as a function of time. If time change is dt, then,

$$x_{t+1} = x_t + v * \cos(\psi) * dt,$$
$$y_{t+1} = y_t + v * \sin(\psi) * dt,$$
$$\psi_{t+1} = \psi_t + \frac{v}{L_f} * \delta * dt,$$

        Where $L_f$is the length from the center of mass of the vehicle to the center of the axle of the front tires. It affects the turn radius of the car and how fast it turns for a given steering angle.

$$v_{t+1} = v_t + a * dt,$$

        The control input, a, is the linear accelerator applied to the car. It changes the car speed.
        iv. Actuators/Control Inputs
        There are two controls that we model the action, the steering and the acceleration/deceleration of the car.
            1. Steering(δ): The steering input is the angle of the front wheels as in the bicycle model. The range of the steering is set to [-25, +25] degrees. The actual range of the input will be [+1,-1] but is mapped back to the min, max range.
            2. Linear Acceleration, a. The range will be -1 -max deceleration and +1 max acceleration.
        v. MPC
            1. The goal will be stay on track of the reference trajectory by feeding the appropriate steering and acceleration based on the deviation from reference:
                a) Cross track error – the distance the car to the reference trajectory (closest distance), b) the orientation error.

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

Model Predictive Control (MPC)

i. $y_t - f(x_t)$ gives the distance the car is from the reference trajectory
ii. The term $v_t * \sin(e\psi_t) * dt$ is the projected error from the car's expected move in the next time step.

b) Orientation Error
i. $e\psi_{t+1} = \psi_t - \psi_t^{desired} + (\frac{v_t}{L_f}\delta_t dt)$
ii. $\psi_t - \psi_t^{desired}$ is the deviation away from desired
iii. $\frac{v_t}{L_f}\delta_t dt$ is the error from the car's dynamics

c) MPC
MPC will solve for the control input values that minimizes an objective function to be built from the above errors subject to the constraints of the problem. MPC uses the open source software, Ipopt (https://projects.coin-or.org/Ipopt). It is a large-scale optimizer developed for the optimization of a function, linear and non-linear, operating under constraints.

The output is an estimable of the state variables described above for some number of time steps defined by N. The length of the time steps will depend on the how fast things need to change to get a good approximation.

b. Explain timestep and elapsed duration (N and dt)
i. The time step is the resolution.
ii. N is the number of time steps the model will be projecting forward based on the current inputs.
iii. N*dt is the duration over which the project is made for.
iv. The tradeoff for more time steps to improve accuracy but it increases in computation time to find a solution.
v. In the tuning below, through trial and error, the values of N (10) and dt (0.1s) was determined to give good results.
vi. Also the latency in the system is hard code here in the variable latency. In seconds, it is 0.1s.

```
// TODO: Set the timestep length and duration
size_t N = 10;
double dt = .1;
double latency = 0.1; // 100 milliseconds
```

c. Preprocessing: Polynomial Fitting and MPC

Model Predictive Control (MPC)

i. The waypoints define the ideal trajectory that we want the car to follow. In the simulation, the next 6 waypoints are provided as part of the data package sent by the simulator.
ii. In this model, we will transform the waypoints from the global coordinate system into the vehicle system. This transformation effectively makes the car position and orientation zero, thus simplifying the model equations.
iii. Once the reference trajectory is fitted, we can compute the cte and epsi from the fit and then they are added to the variable vector along with the state variables.
iv. The results are return from the solver. The actuator predictions are in the first two elements of the vector array. The remain array values are the project positions of the car found by the solver. This is send back to the simulator for visualization (green line).
v. Additionally, in main.cpp, the polynomial fit of the way points is used to generate the fitted reference track and are sent to the simulator for visualization (yellow line). The yellow line fluctuates with the care since the waypoints are actually fixed, but after transforming them into the car frame, and with the car moving about in a less ideally about the reference trajectory, the yellow line will move with the care unless it is spot on the reference.

d. Model Predictive Control with 100 millisecond latency.
i. First the MPC was tuned without the latency. The code without any modification based on the code from the lessons, led to the car oscillating wildly immediately and going off the road.
ii. To get the solver to pay attention to the such large errors that come from going off the read, we multiplied the cte and epsi contribution by a factor greater than one. Below is a screenshot of the 1000.0 factor multiplying the cte and epsi costs. The result was only slightly better. There were still big oscillation leading to the car getting off the track.

```
// Cost function
// The part of the cost based on the reference state.
for (size_t t = 0; t < N; t++) {
  fg[0] += 1000.0*CppAD::pow(vars[cte_start + t] - ref_cte, 2);
  fg[0] += 1000.0*CppAD::pow(vars[epsi_start + t] - ref_epsi, 2);
  fg[0] += CppAD::pow(vars[v_start + t] - ref_v, 2);
}
```

iii. I didn't have much luck using the actuator penalty so based on the experience from the PID project, the following derivative term was added to the cost function to help reduce the oscillations, see below. With these terms and factors, the car was able to navigate the road fairly well.

Model Predictive Control (MPC)

```
// Add cost ~ derivative term - penalize big changes in the errors
for (size_t t = 0; t < N-1; t++) {
  fg[0] += 300.0*CppAD::pow(vars[cte_start + t + 1] - vars[cte_start + t], 2);
  fg[0] += 300.0*CppAD::pow(vars[epsi_start + t + 1] - vars[epsi_start + t], 2);
  fg[0] += 10*CppAD::pow(vars[v_start + t+1] - vars[v_start + t], 2);
}
```

iv. Turning on the latency of 100 milliseconds or 0.1 seconds resulted in big oscillations and the car going off the track.

v. In this case, with the latency, the actuator value returned by the solver were not correct by the time the commands arrived at the car. The car had gone much further than allowed in the time step. So to compensate, I added the addition latency time into the calculation of the error terms:

```
// Account for latency by adjusting for additional error
// due motion of the car and the "extra" time it has to drift
// during the latency
//double del_t = dt;
//double del_t = dt + latency * 1/(t*t);
//double del_t = dt + latency * 1/t;
double del_t = dt + latency/sqrt(t) ;
fg[1 + cte_start + t] = cte1 - ((f0 - y0) + (v0 * CppAD::sin(epsi0) * del_t));
fg[1 + epsi_start + t] = epsi1 - ((psi0 - psides0) + v0 * delta0 / Lf * del_t);
```

vi. Basically, because of the latency, the car has drift for an addition time equal to the latency. The cte and epsi error each have a component contribution to the error that depends fact that it is moving. Normally, these components are proportional to the time step. But we need to add the additional error due to the latency so dt -> del_t = dt + latency when computing the error – see above screenshot.

vii. With the full latency correction, the prediction overcorrect for the errors in the later time steps. You can see the green curve at point away from the car:

viii. The observation here is that we don't need to apply the latency for latter points in the prediction since it should be corrected for in the earlier time steps. To make the contributions smaller I tried latency/t and latency/t^2. That helped but it still caused oscillations because the latency was decaying too fast. Finally, I settled on $del\_t = dt + \ latency/\sqrt{t}$ (see code).

Model Predictive Control (MPC)

ix. With this adjustment, the car was able to make it around the track in a fairly stable fashion. The green track, the predict, still exhibited a curved end far from the current car position, but the curve near the car was good enough to follow the track.

3) Simulation

a. The video of the car driving around the track with latency is in the folder video and is named mpc_final.mp4.