## Table of Contents



# 1. Objective

The objective of this project is to implement Model Predictive Control to drive the car around the track. The project is expected to calculate cross track error and consider 100 millisecond latency between actuations commands.

# 2. Why MPC

A PID controller will never strictly converge and does not take care complex nature of driving conditions.

When we drive, we aim to stay in a lane, keep to a speed, and turn smoothly.
To do this we look ahead. For a self-driving car this challenging.

For example, after perception and localization systems have identified a path and we need to look at how far we are from the path right and adjust the steering angle and throttle. Using PID controller based approach will always overshoot the path. In PID controller, we add a derivative to make it more graceful and integral to counter drifts over time. PID controller always overshoots and does not account for complex road conditions and we need better approach.

A MPC controller can keep a perfectly straight line, solving our constant overshooting problem.

# 3. MPC Summary

A MPC controller can keep a perfectly straight line and solve the constant overshooting problem. It looks into the near future to predict outcomes, similar to

the way we look ahead when we drive. It will account for an arbitrarily complex environment, including allowing for things like snow and ice.

- In MPC, vehicle model is implemented using kinematic bicycle model that ignore tire forces, gravity, and mass. This simplification reduces the accuracy of the models, but it also makes them more tractable. At low and moderate speeds, kinematic models often approximate the actual vehicle dynamics.

- MPC control receives the trajectory from simulator as an array of waypoints ptsx and ptsy in the World (map) coordinate space.

- The cross-track error (CTE) and the orientation error (EPSI) are calculated in the vehicle coordinate space. To do this, the waypoints are transformed to the vehicle coordinate space, then MPC control approximates the trajectory with 3rd order polynomial and does prediction of N states with N-1 actuator changes of the car using a prediction horizon T, that is a duration over which future predictions are made. T is the product of two other variables, N and dt, where N is the number of timesteps in the horizon and dt is how much time elapses between actuations.

- The CTE is calculated as the value of the polynomial function at the point x = 0 and the EPSI is -arctan of the first derivative at the point x = 0.

- After that, MPC control predicts N state vectors and N-1 actuator vectors for prediction horizon T, using optimization solver Ipopt (Interior Point OPTimizer, pronounced eye-pea-Opt), and returns a new actuator vector, which is the transfer between predicted t+2 and t+3 states.

To predict N optimal states and N-1 actuator changes, the following cost function was used in the class FG_eval:

w0*cte^2 + w1*epsi^2 + w2*(v - ref_v) + w3*(delta(t+1)-delta(t))^2 + w4*(a(t+1)-a(t))^2 + w5*a^2 + w6*delta^2

where MPC hyperparameters are:

1. weights w0, w1, w2 are used to balance cte, epsi and distance to target speed,
2. weights w3 and w4 are used to control smoothness steering, smoothness of acceleration,
3. weights w5 and w6 are used to minimize the use of steering and the use of acceleration.

## 4. Processing Pipeline

1. Initialize mpc class
2. Collect data from simulator
3. Convert map space to car space

4. Fit line to get coefficients
5. Calculate errors (Cross track and Psi) and state definition.
6. Solve - Identify the best trajectory
7. Pass output to simulator
10. Add latency to mimic real world driving conditions

## 5. The model

In order to drive the car around we need to know the state of the car, the actions we need to perform and lastly the outcome of our actions.

In addition, we also have a reference trajectory which we desire to follow. MPC uses the state of the car and the errors between the desired and the reference trajectory to predict an optimal trajectory by simulating different actuator inputs and then selecting a resulting trajectory with the minimum cost.

Our state variables are:
* `x` and `y` the position of the car,
* `psi` the orientation of the car, and
* `v` the velocity of the car.

Our actuations are
* `delta` the steering angle, and
* `a` the acceleration.

Steps

1) First, a third degree polynomial is fitted to waypoints received from the simulator and the cross track error (`cte`) is obtained by evaluating the polynomial at current `x` position. The orientation error `epsi` is obtained by evaluating derivative of the curve at the same position

2) Current State variables are updated using global kinematic model

3) The prediction timesteps `N` and intervals `dt` are defined and used to set up the variables for the MPC optimizer.

4) The state variables and the obtained coefficients are used to set up constraints for the MPC optimization. The aim of the constraints is to make the difference between values at time `t` and time `t+1` equal to zero.

5) An optimizer ([Ipopt](https://projects.coin-or.org/Ipopt)) is given the initial state, then it returns the vector of control inputs that minimize the cost function.

5) The first control input is applied to the vehicle, and we repeat the process for subsequent waypoints.

## 6. Algorithm overview

Main.cpp

```
	MPC mpc
	onMessage from Simulator() {

		get the simulator provided info in following variables
			vector<double> ms_ptsx , vector<double> ms_ptsy, px, py, psi, v

		// car space way points
		vector<double> ptsx, vector<double> ptsy

		// transform map space coordinates (waypoints) to car space
		mapToCarspace() // ptsx and ptsy now holds tranformed (car space)
coordinates.

		Fit the waypoints to third order polynomial and identify the coefficients

		eoeffs = polyfit(ptsx, ptsy,3)

		calculate cte for the approximate function represented by the coeefs
		cte = polyeval(coeffs,0)
		epsi = -atan(coeffs[1])

		initialize starting state to
		Eigen::VectorXd state(6) << 0., 0., 0., v, cte, epsi

		calulate steering angle and throttle using MPC
		vector<double> next_state = mpc.Solve(state, coeffs)

		control Simulator using

		steering_angle = mpc.steeringValue()
		throttle = mpc.throttleValue()

		// send message to Simulator

		Following gives you the predicted path

		mpc.pred_path_x ;
		mpc.pred_path_y ;

		}
```

MPC.cpp

All state variables and actuator variables are stored in single vector. Following index and span establish when one variable ends and other starts

```
x_start      =   0
y_start      =   xstart + N
psi_start    =   y_start + N
v_start            =   psi_start + N
cte_start    =   v_start + N
epsi_start   =   cte_start + N

MPC.solve(state, coeffs) {

      //set the no of model variables

      size_t n_vars = N *6 + (N -1 )*2
      size_t n_constraints = N*6

      vars(n_vars) to hold state of N states
      initialize vars[0] to the first state

      set vars_lowerbound and vars_higherbound
      // This is different from coefficients

      CppAD:ipopt:solve<Dvector, FG_eval>(options, vars, vars_lowerbound,
vars_upperbound ...)

      ok &= solution.status = CppAD::ipopt::solve_result<DVector>:sucess

      cost = solution.obj_value

      The first element in the solution is the best waypoint

      return { solution.x[x_start + 1], .....}
}
```

FG_eval.cpp

```
Eigen::VectorXd coeffs
```

All state variables and actuator variables are stored in single vector. Following index and span establish when one variable ends and other starts

```
x_start      =   0
y_start      =   xstart + N
psi_start    =   y_start + N
```

```
       v_start             =   psi_start + N
       cte_start    =   v_start + N
       epsi_start   =   cte_start + N


       operator()(fg, vars) {

              //  The cost is stored in first element of fg
              //  vars contains state variables of N instances which belongs to one of
the possible way (trajectories)
              //  cpadd::ipopt::solve will be calling this method multiple times and
              //  each time it passes one possible way that represents the path
              //  The idea is that this function calculates the cost and ipopt::solve
will choose the
              //  lowest cost of all possible ways (trajectories)

              fg[0] = 0

              for i=0 to N
                     fg[0] =+  cte cost of this state
                     fg[0] =+  epsit cost
                     fg[0] =+  ref_v cost

              //Minimize the use of actuators
              for i=0 to N-1
                     fg[0] =+   steering angle cost
                     fg[0] =+   throttle cost

              //Minimize the value gap between sequential actuators
              for i=0 to N -2
                     fg[0] =+  cost based on steering angle diff between last two
states
                     fg[0] =+  cost baed on throttle diff between last two states

              //Setup constraints

              for t=0 to N-1
                     consider state at t and t+1
                     consider actuation at t

                     find out the approximate function using coeffs from state t
                     calculate the derivate
                     update the predicted next state in fg[]


              }
```

## 7. Run and Optimize

```
/opt/github/public/selfdrivingnd/selfdrivingnd_currentproject/CarND-MPC-Project $ ls -al
total 160
drwxr-xr-x  22 krramasw  wheel    748 Sep  3 16:00 .
drwxr-xr-x   9 krramasw  wheel    306 Sep  3 13:34 ..
drwxr-xr-x  12 krramasw  wheel    408 Sep  3 16:08 .git
-rw-r--r--   1 krramasw  wheel     53 Sep  3 13:34 .gitignore
drwxr-xr-x   8 krramasw  wheel    272 Sep  3 17:08 .idea
-rw-r--r--   1 krramasw  wheel    689 Sep  3 15:05 CMakeLists.txt
-rw-r--r--   1 krramasw  wheel  12191 Sep  3 13:34 CMakeLists.txt.user
-rw-r--r--   1 krramasw  wheel  12196 Sep  3 13:34 CMakeLists.txt.user.d2a7158
-rw-r--r--   1 krramasw  wheel   1234 Sep  3 13:34 DATA.md
-rw-r--r--   1 krramasw  wheel   5998 Sep  3 13:34 Instructions_README.md
-rw-r--r--   1 krramasw  wheel   9010 Sep  3 13:34 README.md
drwxr-xr-x   7 krramasw  wheel    238 Sep  3 15:57 build
drwxr-xr-x   9 krramasw  wheel    306 Sep  3 15:46 cmake-build-debug
-rw-r--r--   1 krramasw  wheel   1102 Sep  3 13:34 cmakepatch.txt
drwxr-xr-x   6 krramasw  wheel    204 Sep  3 16:04 experiments
-rwxr-xr-x   1 krramasw  wheel    964 Sep  3 13:34 install-ipopt.sh
-rwxr-xr-x   1 krramasw  wheel    332 Sep  3 13:34 install-mac.sh
-rwxr-xr-x   1 krramasw  wheel    326 Sep  3 13:34 install-ubuntu.sh
-rw-r--r--   1 krramasw  wheel   1279 Sep  3 13:34 lake_track_waypoints.csv
drwxr-xr-x   5 krramasw  wheel    170 Sep  3 13:34 media
-rwxr-xr-x   1 krramasw  wheel    128 Sep  3 13:38 run.sh
drwxr-xr-x  10 krramasw  wheel    340 Sep  3 15:58 src
/opt/github/public/selfdrivingnd/selfdrivingnd_currentproject/CarND-MPC-Project $
```

```
/opt/github/public/selfdrivingnd/selfdrivingnd_currentproject/CarND-MPC-Project/src $ ls -al
total 1000
drwxr-xr-x  10 krramasw  wheel     340 Sep  3 15:58 .
drwxr-xr-x  22 krramasw  wheel     748 Sep  3 16:00 ..
-rw-r--r--   1 krramasw  wheel     379 Sep  3 15:53 Context.h
drwxr-xr-x  31 krramasw  wheel    1054 Sep  3 13:34 Eigen-3.3
-rw-r--r--   1 krramasw  wheel    8462 Sep  3 15:58 MPC.cpp
-rw-r--r--   1 krramasw  wheel     404 Sep  3 14:52 MPC.h
-rw-r--r--   1 krramasw  wheel  447926 Sep  3 13:34 json.hpp
-rw-r--r--   1 krramasw  wheel    9716 Sep  3 15:54 main.cpp
-rw-r--r--   1 krramasw  wheel   22804 Sep  3 13:34 matplotlibcpp.h
-rwxr-xr-x   1 krramasw  wheel     489 Sep  3 15:57 run.sh
/opt/github/public/selfdrivingnd/selfdrivingnd_currentproject/CarND-MPC-Project/src $
```

To execute, go to src directory ( as shown above) and do

./run.sh

```
/opt/github/public/selfdrivingnd/selfdrivingnd_currentproject/CarND-MPC-Project/src $ cat run.sh
#!/bin/bash

echo "compiling ..."
cd ../build
make

echo "running ..."

N=20
dt=0.05
ref_v=20
ref_cte=0
ref_epsi=0
Lf=2.65

nactuator_limit=1.0e19
delta_limit=0.436332
acc_limit=1.0


echo "N=" $N
echo "dt=" $dt
echo "ref_v=" $ref_v
echo "ref_cte=" $ref_cte
echo "ref_epsi=" $ref_epsi
echo "Lf=" $Lf

echo "nactuator_limit=" $nactuator_limit
echo "delta_limit=" $delta_limit
echo "acc_limit=" $acc_limit

./mpc $N $dt $ref_v $ref_cte $ref_epsi $Lf $nactuator_limit $delta_limit $acc_limit
```

You can pass N, dt, ref_v, ref_cte, ref_epsi and other hyperparams as input to mpc in run.sh

I have tried with following hyperparams

| | hyperparams | Result |
|---|---|---|
| Run1 | N=20<br>dt=0.25<br>ref_v=60<br>ref_cte=0<br>ref_epsi=0<br>Lf=2.65 | Car overshoots and get stuck |
| Run2 | N=20<br>dt=0.10<br>ref_v=80<br>ref_cte=0<br>ref_epsi=0<br>Lf=2.65 | Car overshoots and get stuck |

| | | |
|---|---|---|
| Run3 | N=10<br>dt=0.05<br>ref_v=20<br>ref_cte=0<br>ref_epsi=0<br>Lf=2.65 | Car completed the trip properly |
| Run4 | N=10<br>dt=0.05<br>ref_v=70<br>ref_cte=0<br>ref_epsi=0<br>Lf=2.65 | Car completed the trip properly |

## 8. Pros & Cons of MPC with PID

* With MPC, it gives better integration of speed and steering angle. While you can use two different PID controllers (ie one for speed and one for steering), they don't directly interact with each other.

* Better performance going straight. A PID controller will never strictly converge, where as a MPC controller can in theory reach 0. This is visible in practice, in the PID project to get good performance on tight corners the model generally needed to wander a bit on straightaways. Where as in the MPC project you can get good performance on corners and straight lines.

* Smoother performance in the real world. The ability to "look ahead" helps mimic a human driver as discussed above. Further our model can account for arbitrary dynamics including latency. More advanced models could consider tire performance, snow, etc. Related to this is better recovery. While a far off course PID controller sometimes gets a little "lost" the MPC appears to be more robust. I'm guessing this is due to the "look ahead" nature, which more naturally follows the target path, where as the PID controller tries to get back to the "goal of the moment", so as the complexity of the target path increases, the PID can get more and more "lost".

## 9. Beyond current MPC project implementation

- Add coefficients in the logic and pass it from command line experiments
- Deeper understanding of space transformations