| **Criteria** | **Meets Specifications** |
| --- | --- |
| The Model | Student describes their model in detail. This includes the state, actuators and update equations. |

the model consists of the following steps:

* “Solve” function contains the steps to run the model and compute actuators
* set the size of the model variables according to the timestep length (N) for states and inputs
* initiate the first state elements with the values passed to the model
* Set all non-actuators upper and lowerlimits to the max negative and positive values
* The upper and lower limits of delta are set to -25 and 25
* Acceleration/decceleration upper and lower limits.
* Lower and upper limits for the constraints (0, except for initial state)
* call FG\_eval to define cost function and constraints

FG\_eval:

* extended the class to pass steering angle and throttle from last run (used by the simulator)
* as a prerequisite, I defined weights to penalize errors when calculating cost
* first part of the cost based on the reference state (cross track error, orientation error, speed)
* for speed cost, I applied reference speed (75 mph) to penalize the vehicle for not maintaining reference velocity
* second part of the cost is based on the actuators (steering angle and throttle)
* in addition to the cost for the 2 actuators I added a new component that uses the combination of steering angle/throttle/speed. By implementing this cost, the car is slowing down in bigger curves, so the car stays on the road, excursion is smaller.

(fg[0] += p\_delta\_av \* CppAD::pow(vars[delta\_start + t] \* vars[v\_start + t] \* vars[a\_start + t], 2);)

* 3rd part of the cost is to minimize the value gap between sequential actuations
* then I set up the rest of the constraints. ideally the result of the following equations must be 0:

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

* I pass all the above calculated constraints, variables, cost function to the IPOPT Solver to optimize the control inputs. The result is N-1 number of control input combination (delta, throttle)
* the first control input pair is passed to the simulator
* the predicted x,y states (N number) are also handed over to the simulator for visualization

|  |  |
| --- | --- |
| Timestep Length and Elapsed Duration (N & dt) | Student discusses the reasoning behind the chosen *N* (timestep length) and *dt* (elapsed duration between timesteps) values. Additionally the student details the previous values tried. |

the best N & dt that worked for me are N = 10, dt = 0.05.

I have tried several combinations. Higher N and dt resulted bigger deviation from the reference line. Calculation with lower N causes faster execution time, but increasing the dt causes less accurate actuators.

|  |  |
| --- | --- |
| Polynomial Fitting and MPC Preprocessing | A polynomial is fitted to waypoints.  If the student preprocesses waypoints, the vehicle state, and/or actuators prior to the MPC procedure it is described. |

In order to fit polynomial first I convert waypoints (ptsx, ptsy) provided in map coordinates to car coordinates. I calculate distance between the car’s current position (px, py) and converted waypoint, then calculate relative angle to the car’s orientation (psi). This relative angle is the basis to calculate the waypoints’ x, y coordinates in the car’s system.

Then I applied 3rd order polynomial to get coefficients.

I am passing the converted waypoints coordinates to the simulator to display the reference trajectory (yellow line).

vehicle initial state is:

* x: 0
* y: 0
* psi: 0
* v: value provided by the simulator
* etc: 0
* epsi: 0

|  |  |
| --- | --- |
| Model Predictive Control with Latency | The student implements Model Predictive Control that handles a 100 millisecond latency. Student provides details on how they deal with latency. |

before calling the MPC model I recalculate initial state ([0,0,0,v,0,0]) considering latency (dt = 0.1 (100 millisecond)). I used kinematic model:

x((latency)​​=x(t)​​+v(​t)​​∗cos(ψ(​t)​​)∗dt

yt+1=yt+vt∗sin(ψt)∗dty\_{t+1} = y\_t + v\_t \* sin(\psi\_t) \* dty(​latency)​​=y(​t)​​+v(​t)​​∗sin(ψ(​t)​​)∗dt

ψt+1=ψt+vtLf∗δt∗dt\psi\_{t+1} = \psi\_t + \frac {v\_t} { L\_f} \* \delta\_t \* dtψ(latency)​​=ψ(​t)​​+​ ​​​​v(​t)​​​​/ L​f ∗δ(​t)​​∗dt (δ​t is the actual steering angle)

vt+1=vt+at∗dtv\_{t+1} = v\_t + a\_t \* dtv(latency)​​=v(​t)​​+a(​t)​​∗dt (a(t) is the actual throttle)

recalculate cros track error and orientation error with coeffs, new px and py

ctet+1=f(xt)−yt+(vt∗sin(eψt)∗dt)cte\_{t+1} = f(x\_t) - y\_t + (v\_t \* sin(e\psi\_t) \* dt)cte(​latency)​​= polyeval(coeffs, x((latency)) - y(​latency)​​

eψt+1=ψt−ψdest+(vtLf∗δt∗dt)e\psi\_{t+1} = \psi\_t - \psi{des}\_t + (\frac{v\_t} { L\_f} \* \delta\_t \* dt)eψ(latency)​​= psi - atan(coeffs[1]);

The resulting state is the new initial state for MPC.