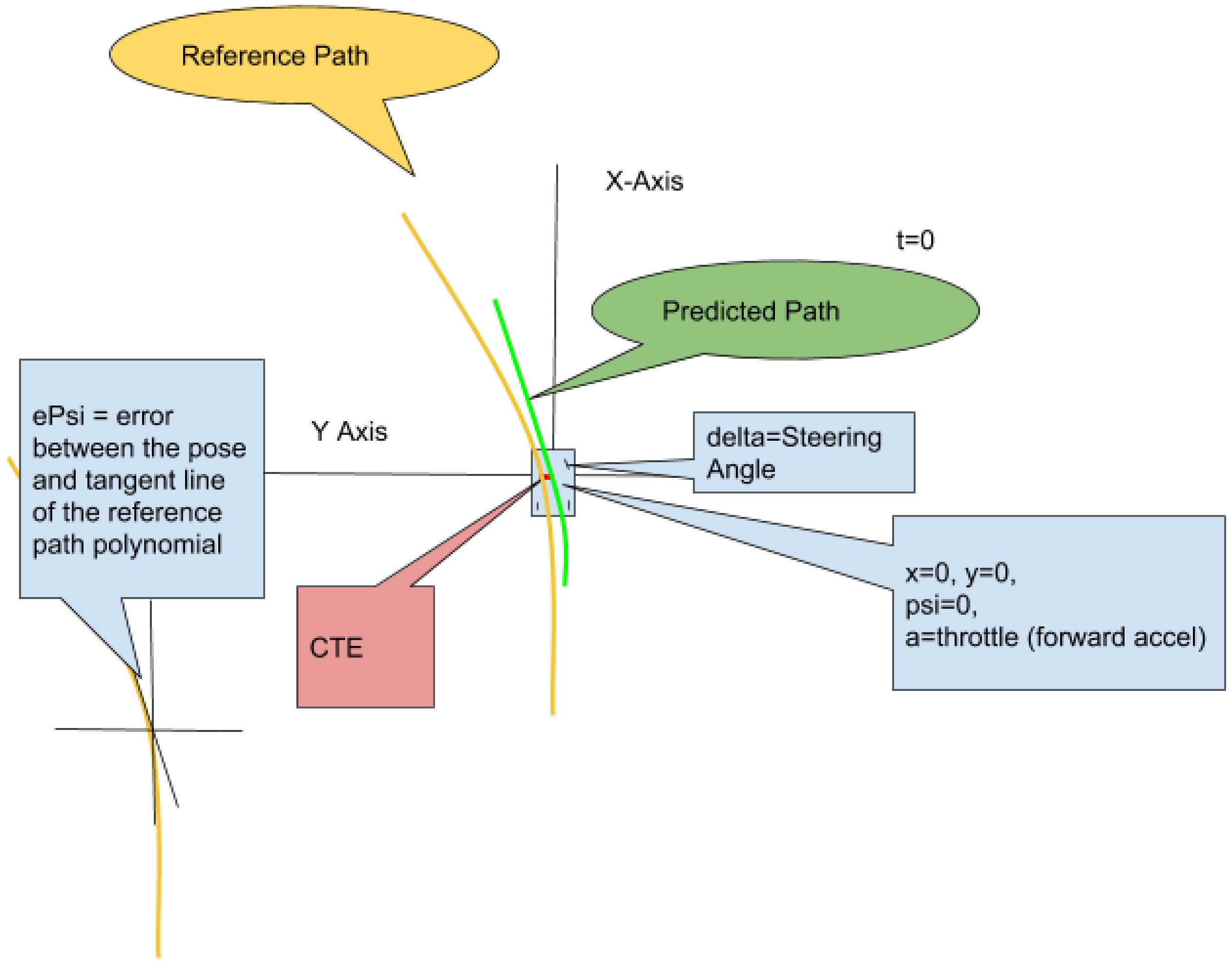
Model Predictive Control Project

Author: Liam O’Gorman

The run with 100ms steps over 1s video here.
Google drive folder with google sheets, csv data and video captures here.

The Model

Student describes their model in detail. This includes the state, actuators and update equations.
The model consists of the vehicle pose.


Reference Path
X-Axis
t=0
Predicted Path
ePsi = error between the pose and tangent line of the reference path polynomial
Y Axis
delta=Steering Angle
x=0, y=0,
psi=0,
a=throttle (forward accel)
CTE

Robot Pose

x	Robot Forward distance
y	Robot Lateral distance

v	Robot velocity in m/s
Psi	Angle of the motion of the robot - initially 0

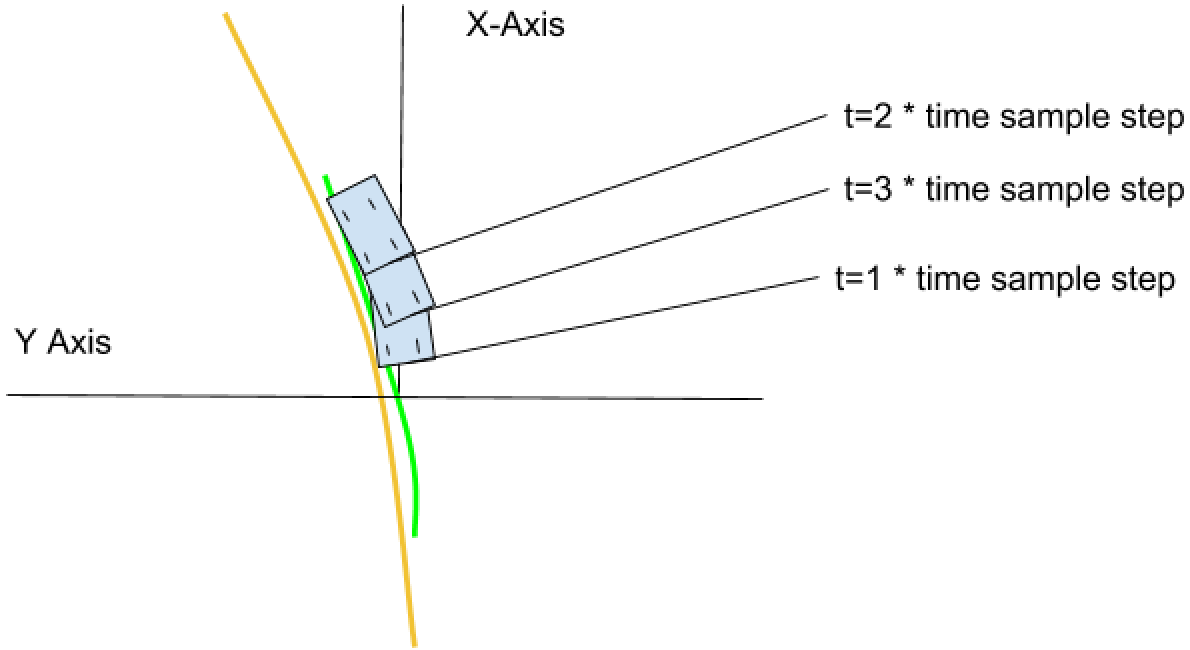
Error state relative to reference trajectory.

CTE	Cross track error - lateral (y) distance between the reference trajectory and the center of the robot.
ePsi	Error between the angle of the car and the reference trajectory.

Actuator control values

Delta	The steering angle
Acceleration	Throttle. For the purposes of this project the throttle value is assumed to be the longitudinal acceleration of the robot.

All of these state variables are sampled over a time horizon. The steering angle will apply a change in the orientation that is dependent on the forward speed of the car. So that over time the position of the car can be predicted.



The equations for this prediction are as follows. These are an approximation of motion - using psi at each sample point but not taking into account the yaw rate and acceleration between each step.

ROBOT POSE update EQUATIONS

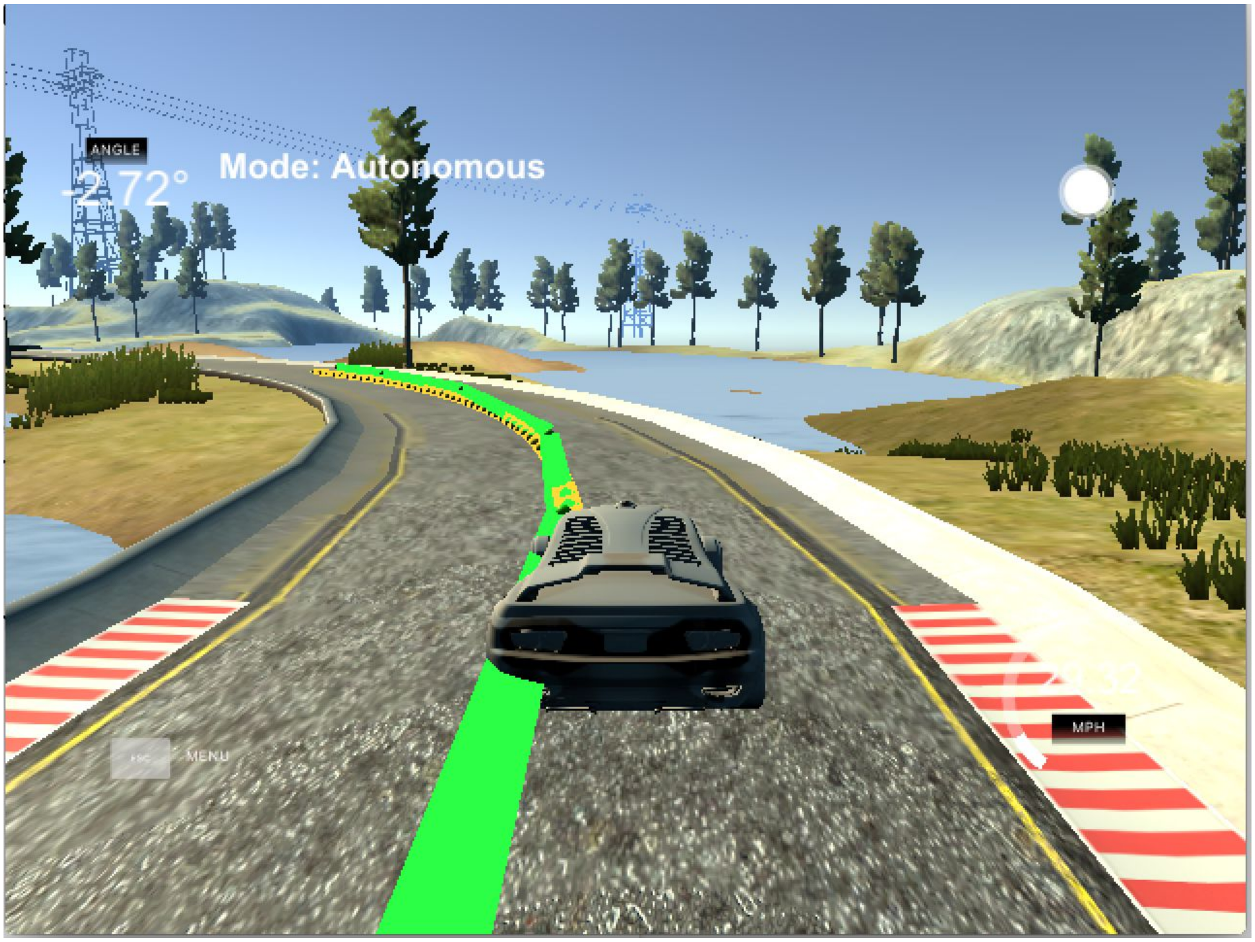
```
x[t+1] = x[t] + v[t] * cos(psi[t]) * dt
y[t+1] = y[t] + v[t] * sin(psi[t]) * dt
psi[t+1] = psi[t] + v[t] / Lf * delta[t] * dt
v[t+1] = v[t] + a[t] * dt
```

```
TRACKING ERROR
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
```

Generating Reference Trajectory

First step in the MPC process is to create a reference trajectory. This is done by taking the trajectory waypoints from the incoming message and converting these to the robot's coordinate system.
This is a two step process, first slide the x & y by the car's position, then rotate the point around the origin by the angle of the robots orientation.

These points are then passed to the function polyfit to create a 3rd order polynomial representation of the line. In order to visualize this and ensure it's correctness I utilized the visualization.



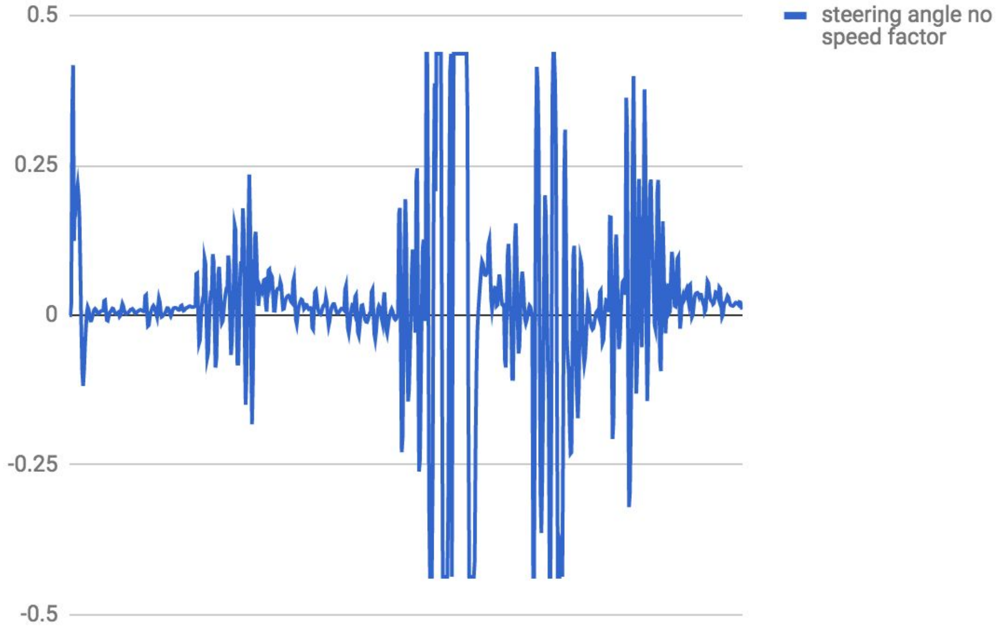
Optimization cost function.

After getting the initial code working with the IPOPT optimizer , it was necessary to tune the cost function.
I chose initial costs for Cross Track Error (CTE), Orientation Error (ePSI) , reference speed error and costs for use of the actuators at each time step.

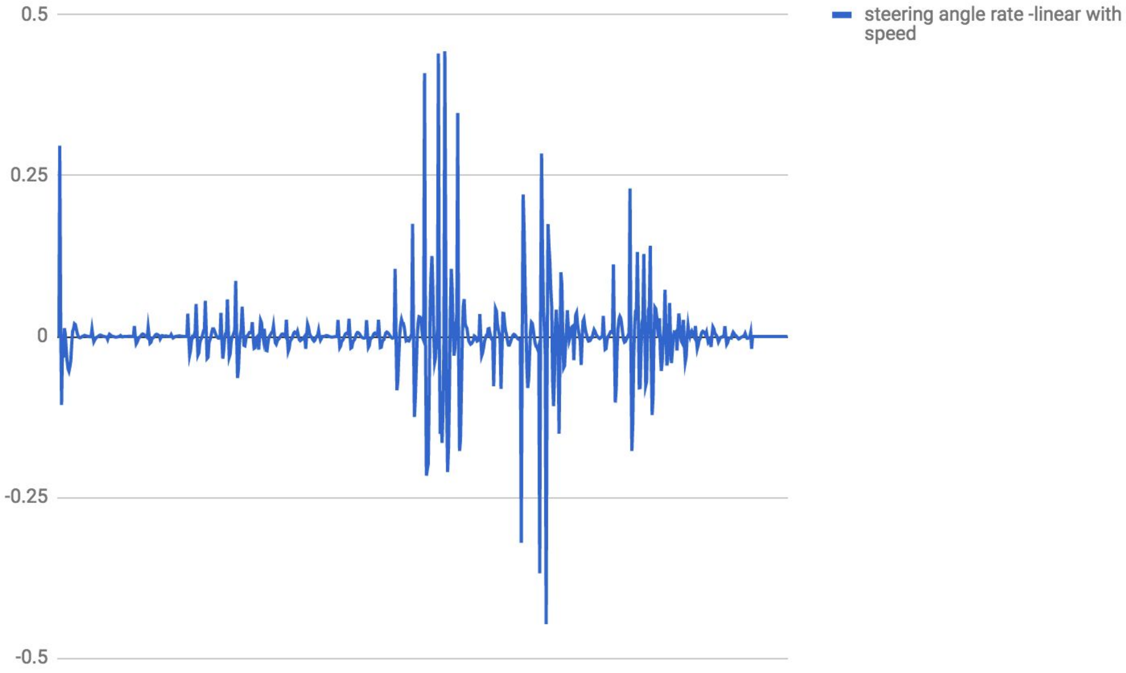
I found that this was very unstable while driving. Generally the steering angle would just flip flop from full left, to full right.
I then started experimenting with scaling factors ,but didn’t really improve. I then added in costs for the rate of change of change of actuators by using the difference between timesteps.

I found that I had to factor the penalty for the rate of change of the steering angle by a large constant to damp the steering reaction to CTE significantly to make the robot stable. I also experimented by varying this factor by the speed at each step also led to some improvements in stability.

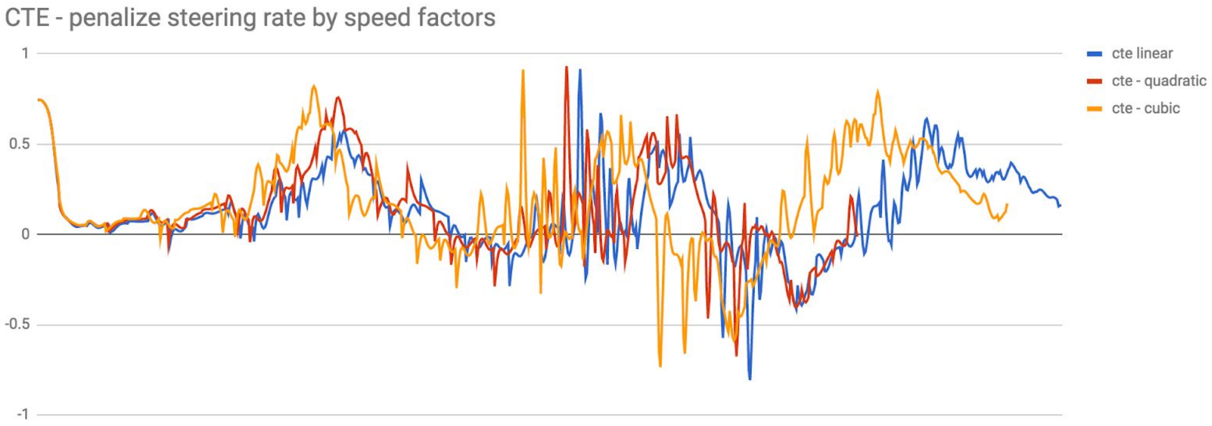
This is a chart of the rate of change of steering angle for a full run around the track.



This is the same chart, but with the cost of the rate of steering angle chance scaled linearly with the velocity of the robot. In this case there are less harsh steering jerks than the case above. Though there is still room for improvement.



I also experimented with varying this cost by linear/quadratic/cubic speed. In the end the linear scaling seemed to work.

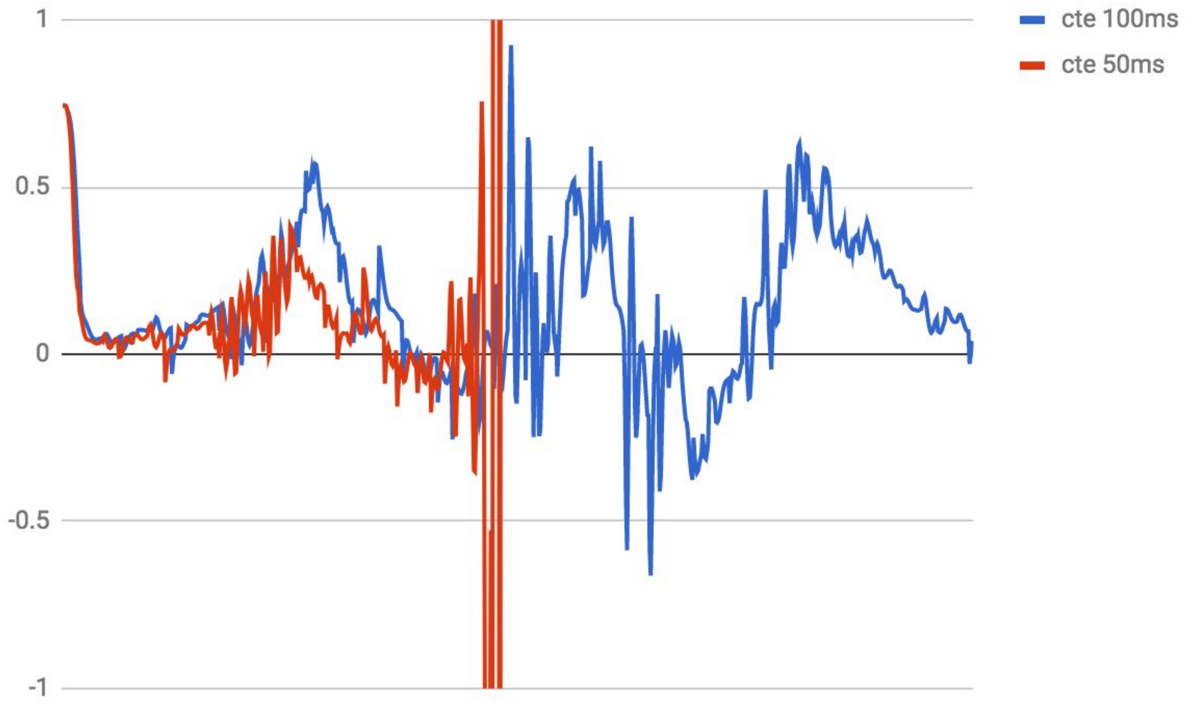


Optimization Timestep Length and Elapsed Duration (N & dt)

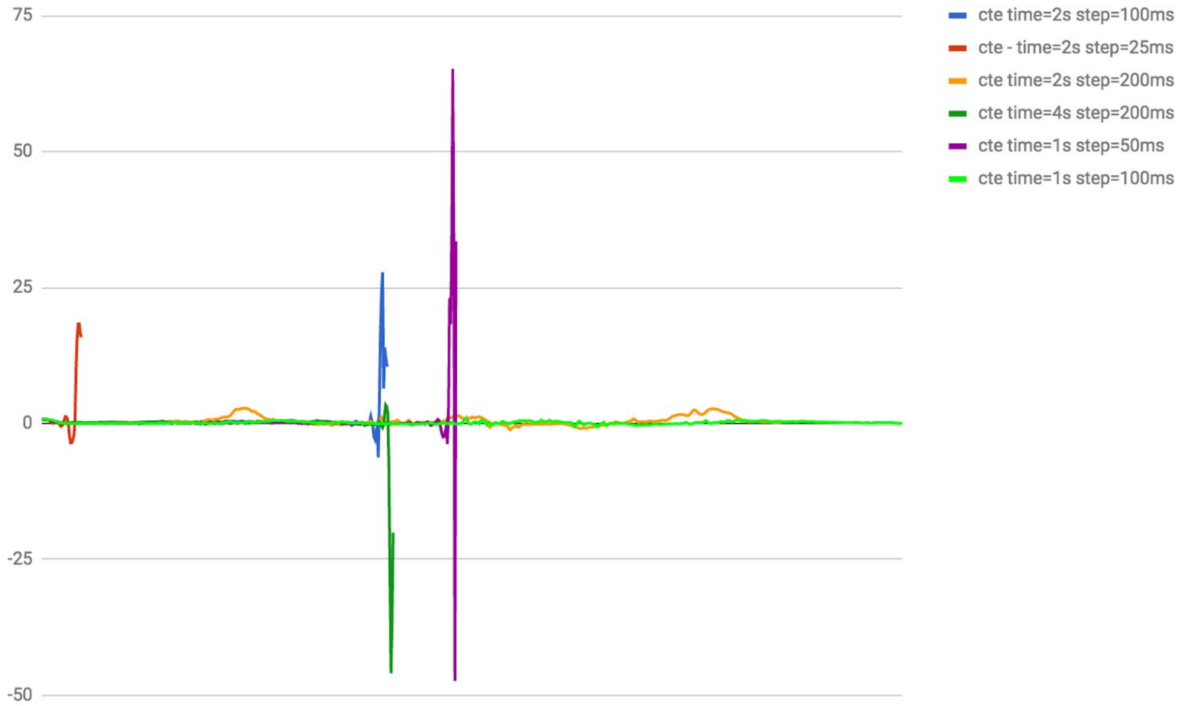
It would appear that having too many steps across the time horizon introduced more latency into the system causing instability. In the end I settled on 10 steps of 100ms for each step.

This chart shows the CTE with different timesteps but a time horizon of 1 second.

Notice that the 50ms steps seems to track better but then drives off the track about half way around.



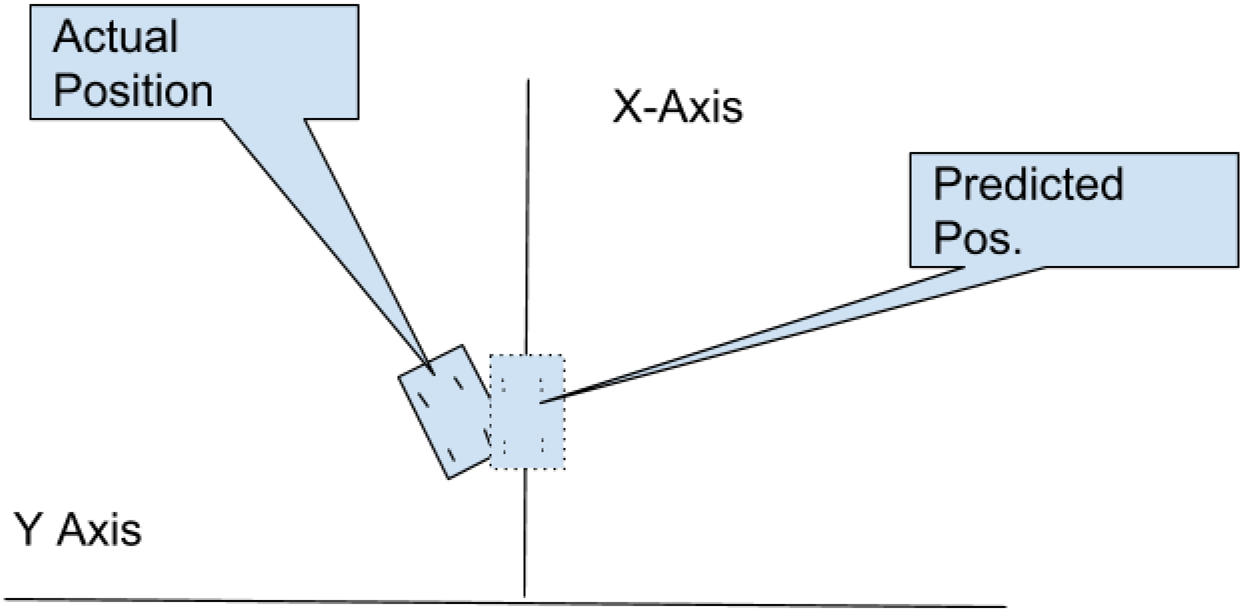
I also tried other variations of time horizon and step size. Many caused the robot to drive off the road. The bright green line was the most stable (1s time horizon at 100ms steps)

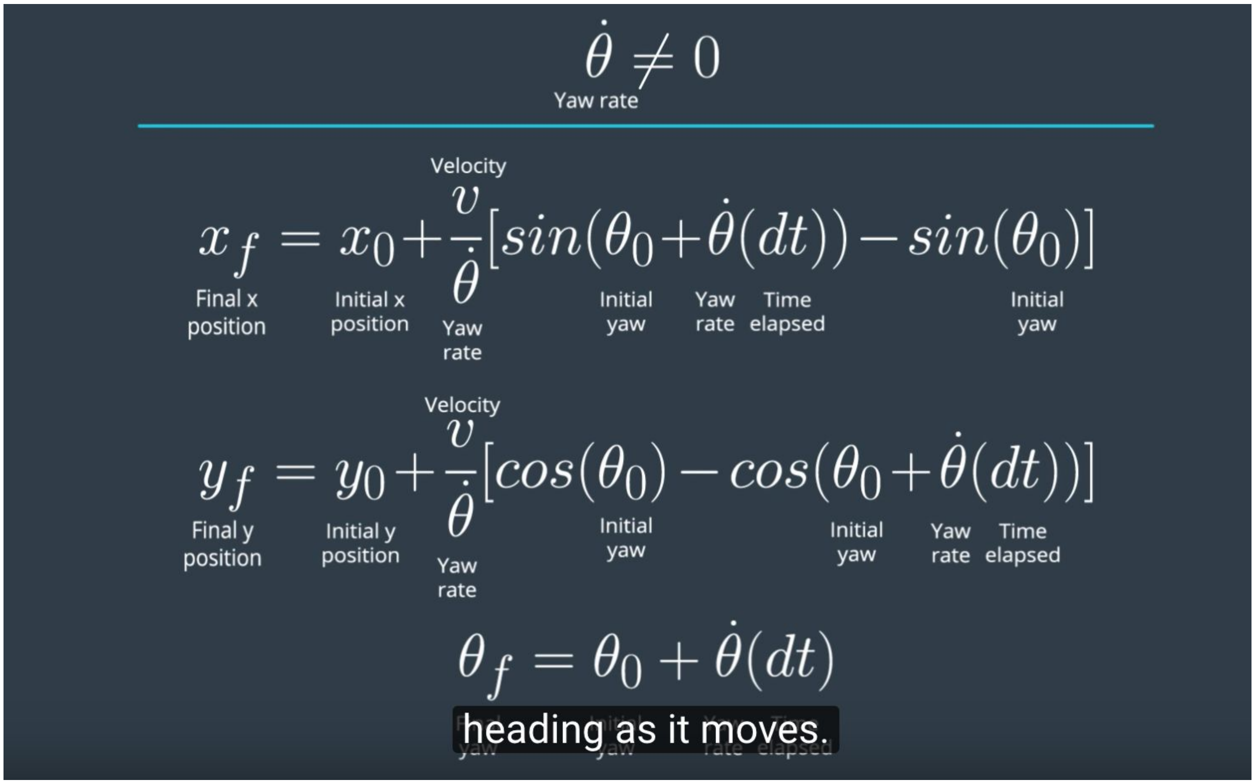


Model Predictive Control with Latency.

I took 2 different approaches with dealing with the latency. My first approach was to constrain a starting portion of the predicted trajectory in the optimization step. Basically preventing the optimizer from adjusting the actuator inputs during an immutable time region = to the latency of the control. I was unable to make this work and debugging it was quite difficult. However, I think that this approach would still work if I had a good cost function to start with. During my initial implementation

Then I used the approach of predicting the initial state 100ms into the future. My first attempt to do this used the simplified update equations. But I realized that the initial angle is always 0 , so at higher speeds with relatively high steering angle the predicted position would be inaccurate. So I modified the equations from the motion model.





$$\dot{\theta} \neq 0$$

$$x_f = x_0 + \frac{v}{\dot{\theta}}[sin(\theta_0 + \dot{\theta}(dt)) - sin(\theta_0)]$$

$$y_f = y_0 + \frac{v}{\dot{\theta}}[cos(\theta_0) - cos(\theta_0 + \dot{\theta}(dt))]$$

$$\theta_f = \theta_0 + \dot{\theta}(dt)$$

heading as it moves.

If the steering angle > 0 we will a non zero yaw rate.

Yaw rate = (Phi_t+1 - Phi_t) / kLatencySeconds , where phi_t = 0.
We can calculate the Phi_t from the steering angle.

Phi_t= v * (steering_angle_rads / Lf) * kLatencySeconds;

Plugging into the above equations this leads to.

```
            future_x = (v * kLatencySeconds / future_psi) *
sin(future_psi);
            future_y =
                (v * kLatencySeconds / future_psi) * (1 -
cos(future_psi));
```

Then using the rest of the update equations was able to create the initial state of the robot at t=100ms which was fed into the optimizer to find the control actuation values.

Areas for improvement.

Periodically as the car drives around the waypoints change and the polynomial function fitting jumps significantly from the previous iteration. This large jump in CTE causes some instability. One improvement would be to use more points from the previous iteration to help keep the reference trajectory smooth.

Further experimentation with the cost function I think would allow for a more comfortable drive around the track and possibly at higher speeds too.