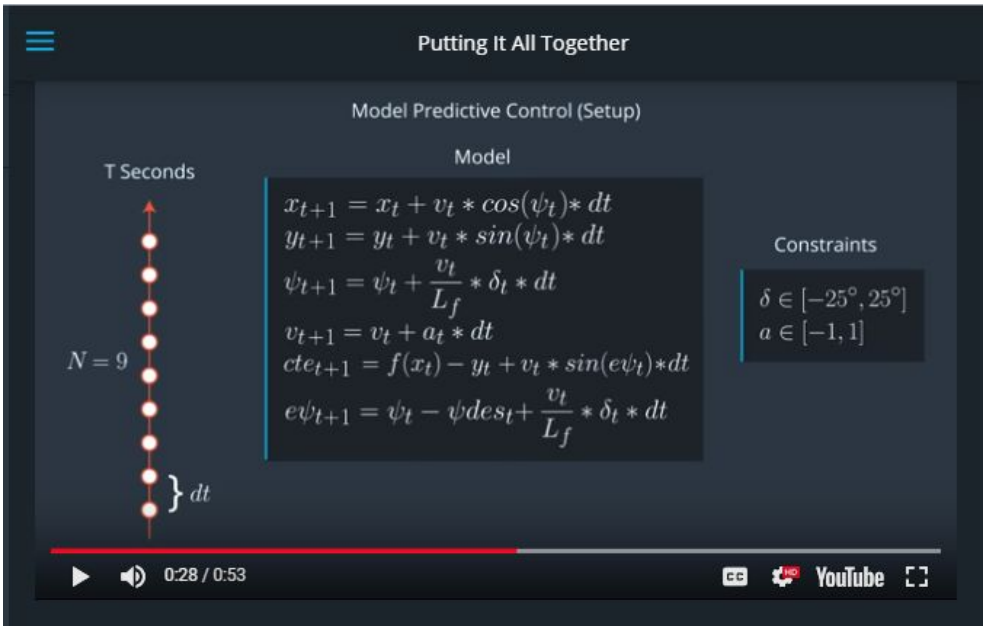
Model Predictive Control
Udacity Term 2 (SDCND)
Christopher Palazzolo

Model Used:

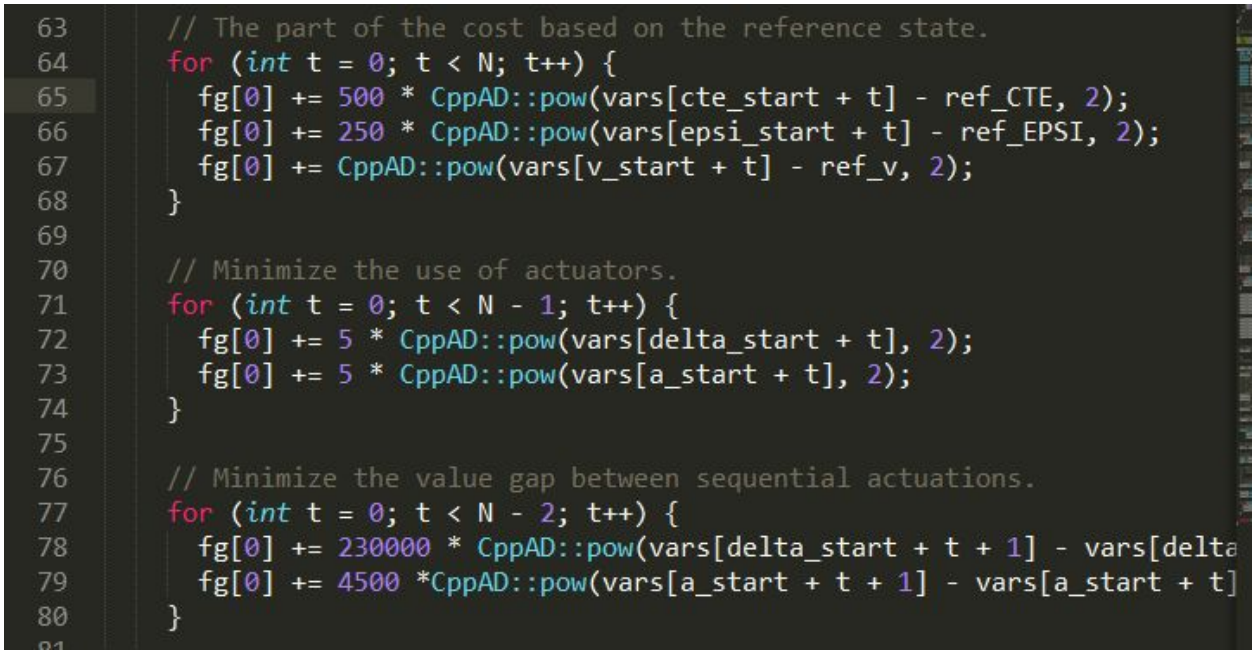
The model that was used in this project is shown below. This came from Lesson 19. It highlights some of the key points on the left is the concept for the receding horizon, the middle has the state equations and to the right is the models constraints. For the constraints in the model the steering constraint “delta” and acceleration “accel” are used as inputs to the model to actuate the vehicles controls.



The model above describes the method to calculate the x and y position of the vehicle, the orientation or heading ‘psi’, velocity, “CTE” which is +/- offset from lane center error, and “epsi” which represents the error between the actual heading and the desired heading.

Model Outputs:

The output or “Actuators” for the model result in a change acceleration and steering angle. A cost function is implemented in MPC.ccp from lines 64 - 80. The cost function has contributions from three main areas, the reference state, use of the vehicles actuators and a penalizing due to sequential actuations resulting in a drastic change. All of the calculations for this part come from the SSE using the CppAD module for the C++ library. As for the weighting coefficients I found that by far the largest impact of stable driving for my model was that related to steering actuation. Initially I could not even get past the first corner without curb hopping right off of the course!! As I raised this value on line 78 I realized that I was getting a much more stable output from the steering actuation.



```
63    // The part of the cost based on the reference state.
64    for (int t = 0; t < N; t++) {
65      fg[0] += 500 * CppAD::pow(vars[cte_start + t] - ref_CTE, 2);
66      fg[0] += 250 * CppAD::pow(vars[epsi_start + t] - ref_EPSI, 2);
67      fg[0] += CppAD::pow(vars[v_start + t] - ref_v, 2);
68    }
69
70    // Minimize the use of actuators.
71    for (int t = 0; t < N - 1; t++) {
72      fg[0] += 5 * CppAD::pow(vars[delta_start + t], 2);
73      fg[0] += 5 * CppAD::pow(vars[a_start + t], 2);
74    }
75
76    // Minimize the value gap between sequential actuations.
77    for (int t = 0; t < N - 2; t++) {
78      fg[0] += 230000 * CppAD::pow(vars[delta_start + t + 1] - vars[delta
79      fg[0] += 4500 *CppAD::pow(vars[a_start + t + 1] - vars[a_start + t]
80    }
```

Prediction Horizon:

For the model we need to choose a timestep length (N) and duration (dt). After trying several combinations I settled on an N = 8 and dt = 0.1. I watched the QA video and they suggested starting with a 10 and 0.1. That caused an unstable behavior for me in my model.

```
1  #include "MPC.h"
2  #include <cppad/cppad.hpp>
3  #include <cppad/ipopt/solve.hpp>
4  #include "Eigen-3.3/Eigen/Core"
5
6  using CppAD::AD;
7
8  // TODO: Set the timestep length and duration
9  size_t N = 8;
10 double dt = 0.1;
```

Polynomial fitting and MPC PreProcessing:

The waypoints that are generated by the simulator are initially transformed for simplicity. By transforming the waypoints to be from the cars perspective we can shift the x and y coordinates such that the origin is at the car. The orientation angle is also corrected in a similar manner.

Main.cpp

```
109         // Push calculated values back to Waypoint X/Y Vectors. Homogeneous TM used
110         // https://www.miniphysics.com/coordinate-transformation-under-rotation.html
111         // Transforms the perspective for the waypoint to be from the vehicles point of vi
112            double x_Shift = ptsx[i] - px;
113            double y_Shift = ptsy[i] - py;
114            wayPt_x.push_back( x_Shift * cos(-psi) - y_Shift * sin(-psi) );
115            wayPt_Y.push_back( x_Shift * sin(-psi) + y_Shift * cos(-psi) );
116            }
117
```

Model Predictive Control with Latency:

```
148
149         const double del_t = 0.1;
150         const double Lf = 2.67;
151         const double px_t = 0.0 + velocity * del_t;
152         const double py_t = 0.0;
153         const double psi_t = 0.0 - ( velocity * delta * del_t )/ Lf;
154         const double vel_t = velocity + accel * del_t;
155         const double cte_t = cte + velocity * sin(epsi) * del_t;
156         const double epsi_t = epsi - ( velocity * del_t * delta ) / Lf;
157
158         //
```

To simulate a more real world situation where there would be a latency between commanding an actuator and actually achieving the desired result that was built into our model. 100 ms was the amount of time that we were told to use as a benchmark. In main.cpp the kinematic equations implemented were calculated with a value of 100 ms as a delay interval for the model.