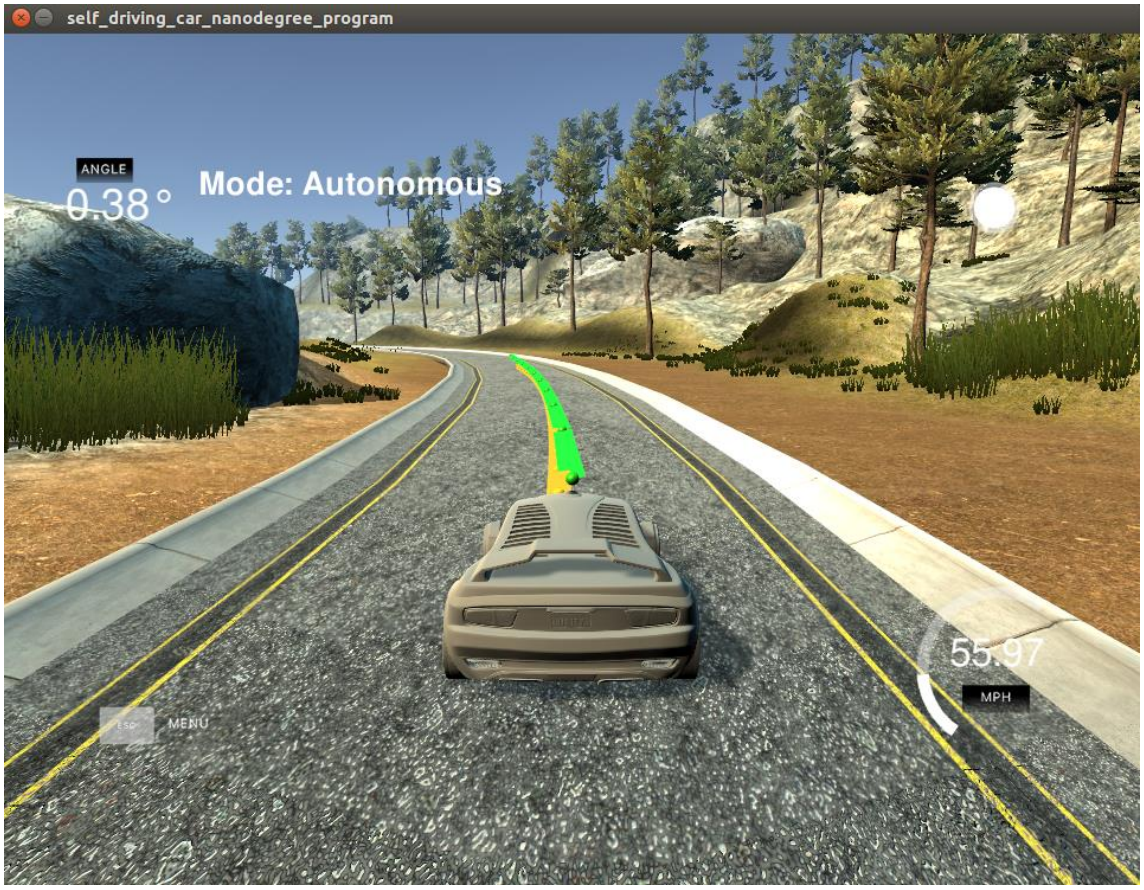
Model Predictive Controller Project – Udacity Self-Driving Car Nanodegree.

By Alejandro Trigo

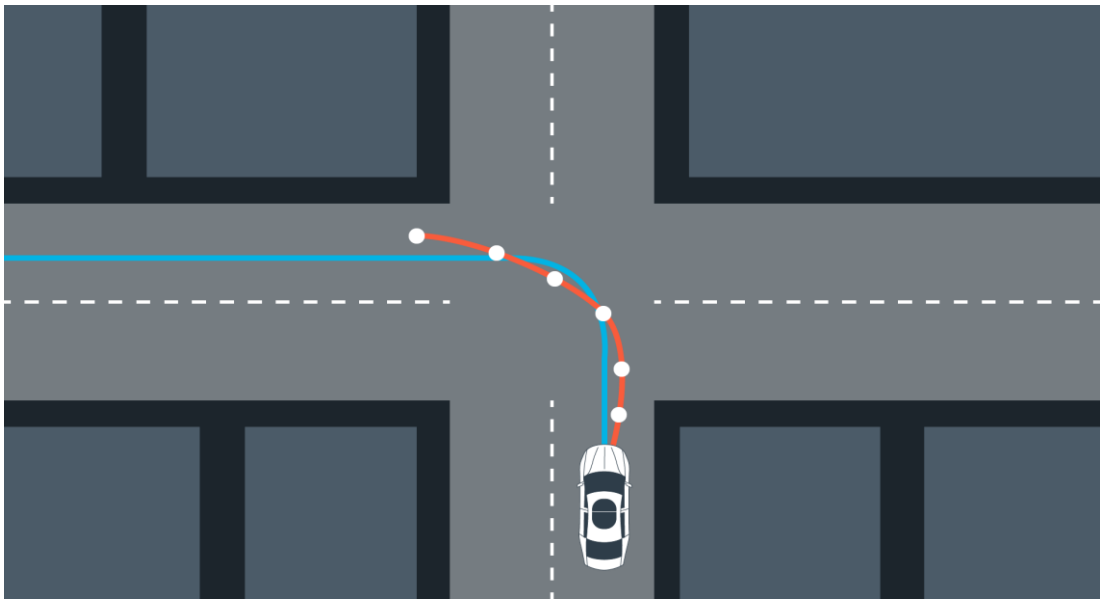


1. The Model Predictive Control

The MPC is a model that predicts the resulting trajectory and selects the trajectory with minimum cost.
In this project, we have to implement this model with an extra of difficulty, latency. This delay of time can make your stable control model fail with bad trajectories. There are two ways of dealing with latency in the MPC:

a. Use kinematic equations to predict the states for after 100ms before sending them to MPC. This is by far the most popular method and easy to understand.

b. The other method is to include the effect of latency in MPC. This can have two further approaches:
   - One is to only simulate the latency for the first actuator value. instead of taking solution index 0 from MPC, you take index-i instead where matches the time after 0.1s (in other words i=0.1/dt) but you will need to fix the actuator values for all indices before i by setting vars_upperbound and lowerbound with values from previous run.
   - The other is simulate latency in every timestep, instead of taking delta0 and a0 from last timestep, you take the values from another timestep before. The disadvantage of this is you will need to use the same latency value for dt.

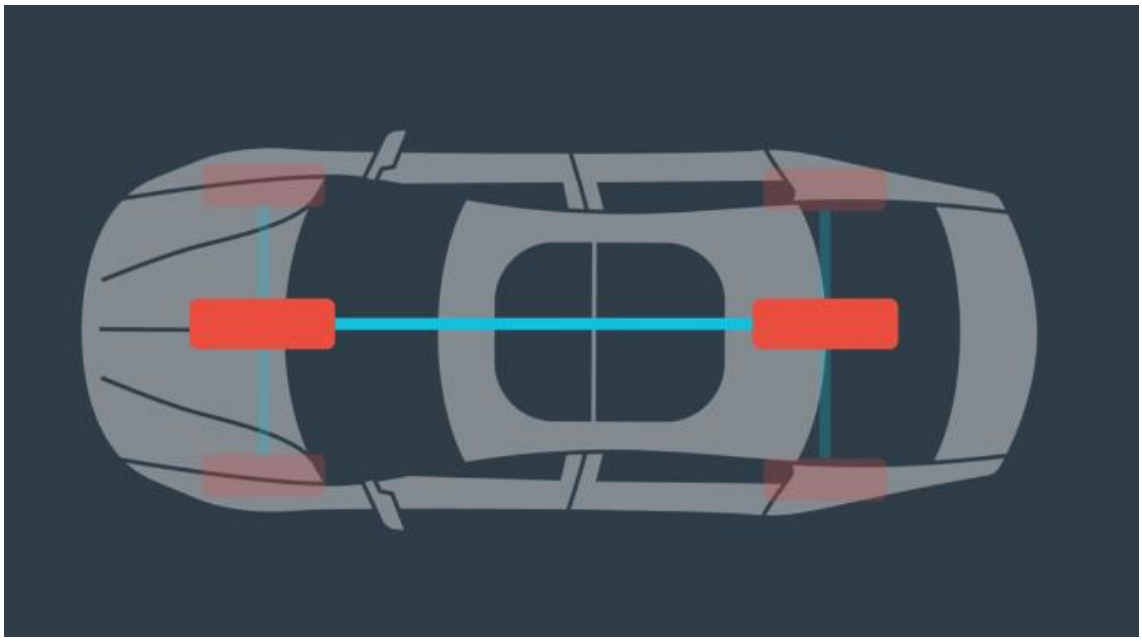


2. The model

The model used in this project is a kinematic bicycle model. It does not evaluate the dynamical effects, like inertia or friction. The model consist in the following equations:

```
x_[t+1] = x[t] + v[t] * cos(psi[t]) * delta_t
y_[t+1] = y[t] + v[t] * sin(psi[t]) * delta_t
psi_[t+1] = psi[t] + v[t] / Lf * delta[t] * delta_t
v_[t+1] = v[t] + a[t] * delta_t
cte[t+1] = f(x[t]) - y[t] + v[t] * sin(epsi[t]) * delta_t
epsi[t+1] = psi[t] - psides[t] + v[t] * delta[t] / Lf * delta_t
```

Since the model controls also the heading direction, it becomes a non-linear model.



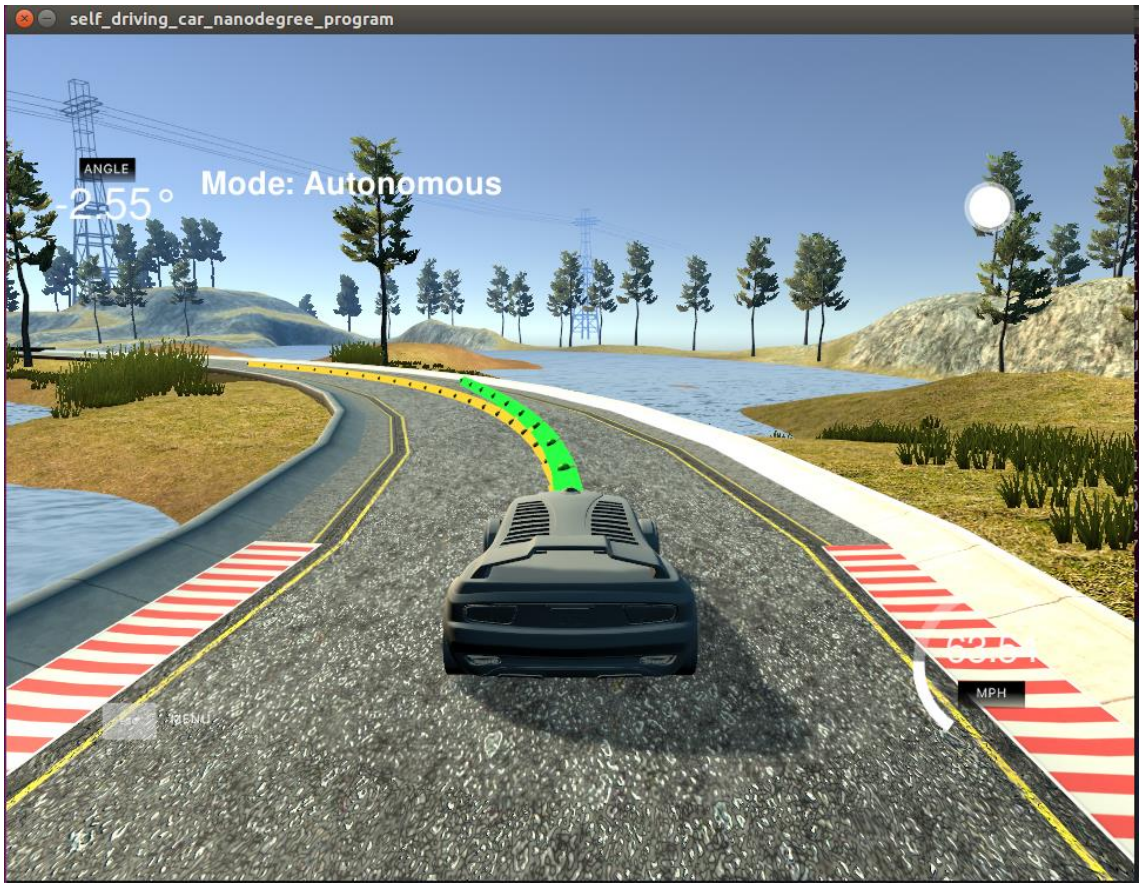
I think the most important part of the model is the cost function; here is where all the magic happens. By increasing the cost when the car wants to speed up and steer in the same time, I can fight the delay of 100ms. The other constants I set for the cost function where selected by trial and error.

The initial state is set to 0 for the x, y and psi values, the speed is got in the JSON data. The cte is calculated by the coeffs (polynomial vector of 3er degree of the points got in the JSON message) and the epsi is the atan of the coeffs negated.

3. Timestep Length and Elapsed Duration (N & dt)

The prediction horizon is the duration over which predictions are made. Is defined by `T = N * dt` so, to have short prediction horizon allows your model to have more responsive controls (Less accurate) and long prediction horizon are usually related to more smooth controls.

In the last moment, I found that the perfect N for my model was 12 and dt 0.05, first I tried with 10 and 0.1 and it was working, but sometimes it was going over the ledges with one wheel. Because of that, I thought that with a bit more steps it could fight a little bit more with the delay.



4. Conclusion

The model deals with the delay quite good for slow (30 mph) and fast velocity (80 mph). I think one of the most important equations of the cost function is the one that penalizes the car if it uses the accelerator and steering wheel in the same time:

```
Fg[0] += 0.25 * CppAD::pow(vars[delta_start + t] * vars[v_start+t], 2);
```

Also the T and the previous state vector which stores the last predicted value of throttle and steering. This all together allows the car to drive all arroun the circuit without going over the ledges.