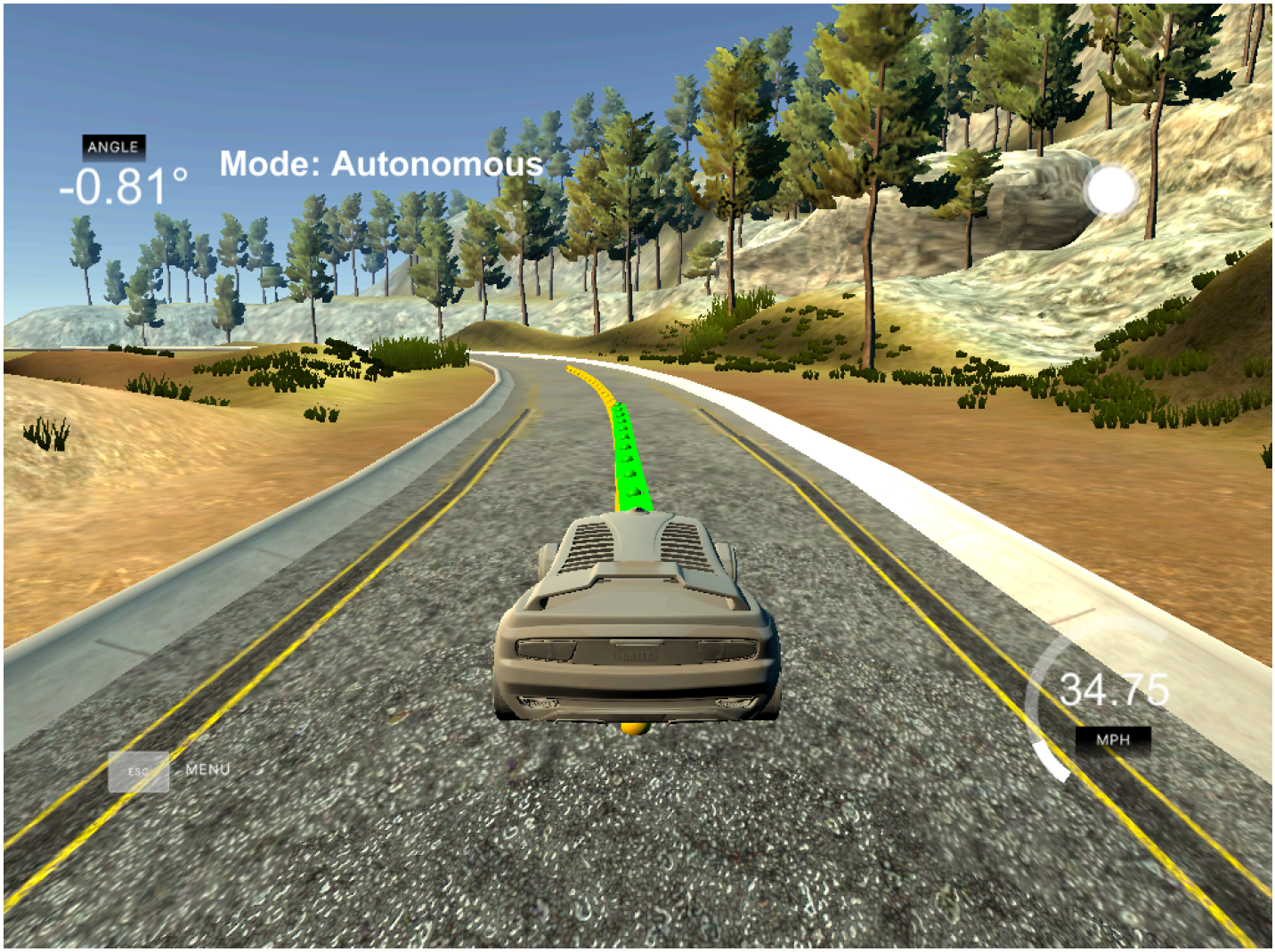
MPC-Project

The goal of this project is to use a kinematic vehicle model to determine steering angle and throttle of a simulated vehicle.



Given a set of waypoints (here in yellow), the aim is to solve a constrained optimization problem, resulting in an optimal trajectory that steers the vehicle safely around the track (here in green).

Model

The kinematic model used to solve this problem is shown below.

Model

$$x_{t+1} = x_t + v_t * cos(\psi_t) * dt$$
$$y_{t+1} = y_t + v_t * sin(\psi_t) * dt$$
$$\psi_{t+1} = \psi_t + \frac{v_t}{L_f} * \delta_t * dt$$
$$v_{t+1} = v_t + a_t * dt$$
$$cte_{t+1} = f(x_t) - y_t + v_t * sin(e\psi_t) * dt$$
$$e\psi_{t+1} = \psi_t - \psi des_t + \frac{v_t}{L_f} * \delta_t * dt$$

The model predicts the vehicle position (x,y), vehicle orientation (psi), velocity (v), Cross-Track-Error (CTE) and Orientation-Error (epsi) N*dt steps into the future, thereby trying to minimize the costs of the model and taking into account additional constraints (see below).

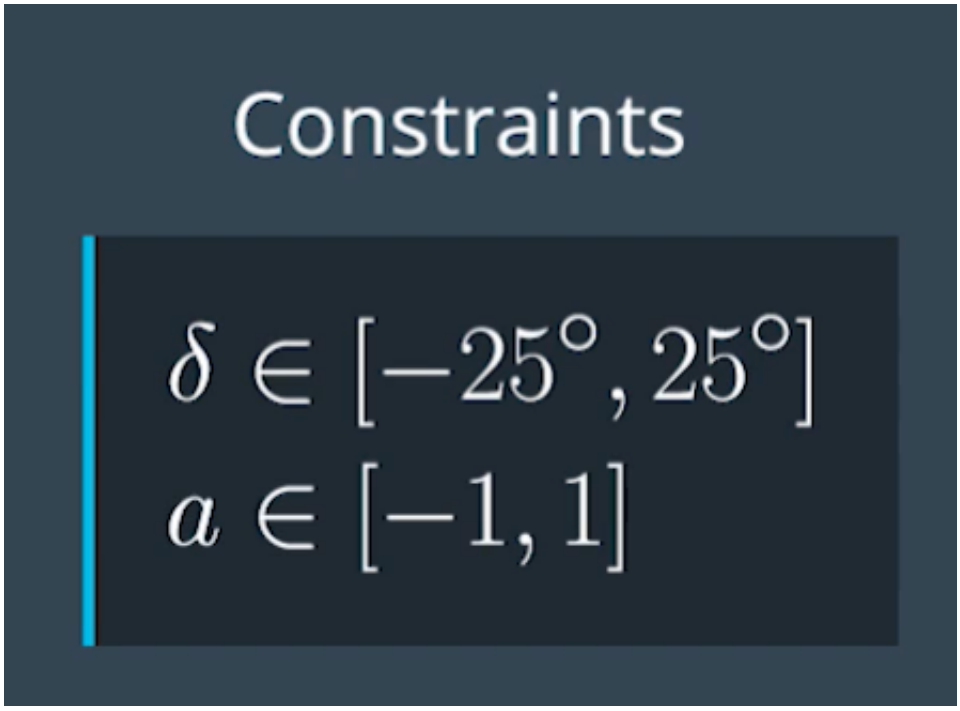
```
for (int i = 0; i < N; i++) {
   fg[0] += 2000*CppAD::pow(vars[cte_start + i], 2);
   fg[0] += 2000*CppAD::pow(vars[epsi_start + i], 2);
   fg[0] += CppAD::pow(vars[v_start + i] - ref_v, 2);
 }

 for (int i = 0; i < N - 1; i++) {
   fg[0] += 25*CppAD::pow(vars[delta_start + i], 2);
   fg[0] += 25*CppAD::pow(vars[a_start + i], 2);
 }

for (int i = 0; i < N - 2; i++) {
  fg[0] += 200*CppAD::pow(vars[delta_start + i + 1] - vars[delta_start + i], 2);
  fg[0] += 20*CppAD::pow(vars[a_start + i + 1] - vars[a_start + i], 2);
}
```

The cost function is a combination of different elements. The first block simply adds a (weighted) squared cost for all differences in Cross-Track-Error and Orientation-Error as well as for differences between the current velocity and the reference velocity.

Additionally, the aim is to punish the change rate of of the actuations (block 2) as well as value gaps between sequential actuations to smoothen out the trajectory.



Timestep Length and Elapsed Duration

I experimented with various N and dt values and found that a dt of 0.1 (instead of 0.05) gave me generally better results because the projection seemed to be more stable. With a step size of 0.05 seconds I found the polynomial changes more "restless". I experimented with N values between 5 and 20 and found that N=10 resulted in a good trade-off between computational effort needed to solve the optimization problem and keeping the car on the track. A N too small yielded projections too much focused on the current position not taking into account waypoints further away (especially in curvy parts of the road), whereas a N too large didn't yield any additional benefits for making decisions for the actuations or even worsened them in more curvy parts of the road.

Polynomial fitting and MPC preprocessing

I followed the recommendation to transform the simulator (map) coordinates into the coordinate system of the vehicle, projecting x, y and psi onto 0. This is done by translating by (-px, -py) and the rotating over -psi.

(ptsx, ptsy) are the waypoints, (px, py) the vehicle (map) coordinates, psi the current vehicle orientation.

```
// shift car reference angle to 90 degrees
double shift_x = ptsx[i] - px;
double shift_y = ptsy[i] - py;

ptsx[i] = shift_x * cos(-psi) - shift_y * sin(-psi);
ptsy[i] = shift_x * sin(-psi) + shift_y * cos(-psi);
```

Model predictive control with latency

In order to deal with latency I project the state one timestep into the future (0.1s) before sending it to the solver. This is done in main.cpp (lines 79-88). The update equations are the same as before.

```
// Predict the state 0.1s into the future to capture the delay
// x = y = psi = 0.0
double dt = 0.1;
double pred_x    = 0.0 + v * dt;
double pred_y    = 0.0;
double pred_psi  = 0.0 - v * delta / Lf * dt;
double pred_v    = v + a * dt;
double pred_cte  = cte + v * sin(epsi) * dt;
double pred_epsi = epsi + pred_psi;
state << pred_x, pred_y, pred_psi, pred_v, pred_cte, pred_epsi;
```