Model Predictive Project Write Up

The model predictive method defines the duration of the trajectory and set N points in the duration with a duration of delta_t. The define the current state as:

$$[x_1, y_1, \psi_1, v_1, cte_1, e\psi_1]$$

Use the following update function and to get the state function in the N points:

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

Then the cost function is defined to characterize the difference the predicted state and the target state. By minimizing the cost function, the optimized actuator, which is a vector of acceleration and steer angle with constraints for each prediction is obtain, then update the state with the first prediction.

- Why smaller dt is better?

Smaller dt provides the model with finer resolution.

- Why larger N isn't always better?

Larger N means predictions more points, so for the same duration of trajectory, the computational time is larger.

- How does time horizon (N*dt) affect the predicted path?

Time horizon N*dt relates to the car speed. If car speed is large, the N*dt should be smaller to reduce the possibility of large cost function.

3rd order polynomial is used to fit the waypoints:

```
for (int i = 0; i < ptsx.size(); i++) {
    double dx = ptsx[i] - px;
    double dy = ptsy[i] - py;
    waypoints_x.push_back(dx * cos(-psi) - dy * sin(-psi));
    waypoints_y.push_back(dx * sin(-psi) + dy * cos(-psi));
}

double* ptrx = &waypoints_x[0];
double* ptry = &waypoints_y[0];
Eigen::Map<Eigen::VectorXd> waypoints_x_eig(ptrx, 6);
Eigen::Map<Eigen::VectorXd> waypoints_y_eig(ptry, 6);

auto coeffs = polyfit(waypoints_x_eig, waypoints_y_eig, 3);
```

To solve the issue of latency, previous actuation is used:

```
// Only consider the actuation at time t.
AD<double> delta = vars[delta_start + t - 1];
AD<double> a = vars[a_start + t - 1];
if (t > 1) {   // use previous actuations (to account for latency)
    delta = vars[delta_start + t - 2];
    a = vars[a_start + t - 2];

}
```