CarND Path Planning Project Report

Wolfgang Steiner
wolfgang.steiner@gmail.com

August 14, 2017

I. Introduction

The goal of this project is to implement a path planning algorithm that enables a car to autonomously navigate highway traffic. The main requirements are keeping a safe distance to the other cars, going as fast as the traffic and speed limits allow and to ensure a comfortable ride for the passengers by limiting the maximum acceleration and jerk.

In this project I followed the approach discussed in [1]: The vehicle controller generates a multitude of candidate trajectories with different target speeds, durations, and target driving lanes. For each candidate trajectory, a cost function is evaluated that also involves the predicted trajectories of other nearby vehicles. The trajectory with the smallest cost is then executed next. This mechanism is complemented by a simple state machine that ensures the correct execution of the intended maneuvers.

II. Coordinate System and Waypoint Preprocessing

For this project I decided to work exclusively in Frenet coordinates [2]. To facilitate this, I implemented the class `TWaypoints` that loads the waypoint coordinates from the supplied `csv` file and fits two splines to the x- and y-coordinates, both having the longitudinal s-coordinate as a parameter. In order to ensure a smooth transition at the starting point of the track, the first and the last waypoint of the track are repeated while subtracting/adding the maximum s-coordinate (i.e. length of the track).

When defining trajectories in Frenet coordinates (s, d), transforming into cartesian coordinates can be accomplished by evaluating the splines $f_x(s)$ and $f_y(s)$ at the s-coordinate and adding the normal to the curve, multiplied by the d-coordinate:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f_x(s) \\ f_y(s) \end{bmatrix} + d \cdot \begin{bmatrix} -\frac{d}{ds} f_y(s) \\ \frac{d}{ds} f_x(s) \end{bmatrix} \tag{1}$$

The inverse transformation from Cartesian to Frenet coordinates is also required when initializing the vehicle's position from the first simulator update and when updating the positions of the other cars at each time step. The simulator does provide Frenet coordinates but they turned out to be of insufficient accuracy. Given a position in Cartesian coordinates (x, y), the corresponding s-coordinate is computed by a simple gradient descent solver [3] that minimizes the error term:

$$J_{xy}(s) = \sqrt{(x - f_x(s))^2 + (y - f_y(s))^2} \tag{2}$$

After having established the point on the spline that is closest to the target position, the d-coordinate can be computed from (1).

III. Jerk Minimizing Trajectories

a. Velocity Keeping Trajectories

The most common state of the car is keeping a lane while maintaining a constant velocity, which should be as close as possible to the speed limit, and while keeping a safe distance to the leading cars. As we are not interested in the exact position at the end of the trajectory, jerk minimizing trajectories can be generated from *quartic* polynomials [1]:

$$s(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 \tag{3}$$
$$\dot{s}(t) = a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 \tag{4}$$
$$\ddot{s}(t) = 2a_2 + 6a_3 t + 12a_4 t^2 \tag{5}$$

The boundary conditions at the *start* of the trajectory are the longitudinal position s_0, velocity $\dot{s}_0$ and acceleration $\ddot{s}_0$ of the current state. From this, the first three parameters of the polynomials can be directly determined:

$$s(0) = s_0 = a_0 \tag{6}$$
$$\dot{s}(0) = \dot{s}_0 = a_1 \tag{7}$$
$$\ddot{s}(0) = \ddot{s}_0 = 2a_2 \tag{8}$$

The boundary conditions at the *end* of the trajectory are the longitudinal velocity $\dot{s}_1$ and acceleration $\ddot{s}_1$ (which is commonly zero). From (4) and (5) we can now formulate the linear system of equations as:

$$\dot{s}(T) = \dot{s}_1 = \dot{s}_0 + \ddot{s}_0 T + 3a_3 T^2 + 4a_4 T^3 \tag{9}$$
$$\ddot{s}(T) = \ddot{s}_1 = \ddot{s}_0 + 6a_3 T + 12a_4 T^2 \tag{10}$$

This can be reformulated as the following matrix equation which in the code is solved for the missing parameters a_3 and a_4 by applying the Householder transformation from the *Eigen* library:

$$\begin{bmatrix} \dot{s}_1 - \dot{s}_0 - \ddot{s}_0 T \\ \ddot{s}_1 - \ddot{s}_0 \end{bmatrix} = \begin{bmatrix} 3T^2 & 4T^3 \\ 6T & 12T^2 \end{bmatrix} \cdot \begin{bmatrix} a_3 \\ a_4 \end{bmatrix} \tag{11}$$

b. Lane Changing Trajectories

For the longitudinal component of lane changing trajectories, the exact end position of in the s-coordinate is still of no interest. For the lateral component d this is different: The trajectory should end at a specific d-coordinate, which is usually the center of the target lane. So I combine a *quartic* polynomial for the longitudinal with a *quintic* polynomial for the lateral component of the trajectory. The derivation is similar to that of (11) and leads to a linear equation system of the form:

$$\begin{bmatrix} d_1 - d_0 - \dot{d}_0 T - \frac{\ddot{d}_0}{2} T^2 \\ \dot{d}_1 - \dot{d}_0 - \ddot{d}_0 T \\ \ddot{d}_1 - \ddot{d}_0 \end{bmatrix} = \begin{bmatrix} T^3 & T^4 & T^5 \\ 3T^2 & 4T^3 & 5T^4 \\ 6T & 12T^2 & 20T^3 \end{bmatrix} \cdot \begin{bmatrix} a_3 \\ a_4 \\ a_5 \end{bmatrix} \tag{12}$$

IV. Tracking of Other Vehicles

Each time the vehicle controller is called by the simulator, sensor fusion data is received. This data contains a unique ID for each vehicle along with position and speed in Cartesian coordinates. The sensor data also contains the current position in Frenet coordinates, which, unfortunately, is generally not accurate enough to be usable.[1]

[1] The longitudinal Frenet coordinate s is used as a starting point for the gradient descent solver.

The sensor data is passed to an object of type `TSensorFusion` which maintains a hash table of `TOtherCar` objects, indexed by ID. Each `TOtherCar` maintains the state in the form $(s, d, \dot{s}, \dot{d})$ to predict its future trajectory during the calculation of the safety distance cost. Each vehicle position is converted from Cartesian to Frenet coordinates (s, d), while the total velocity is used for $\dot{s}$. With these state variables, the vehicle position is predicted under the constant velocity assumption. Changes in velocity are picked up by the frequent state updates through the sensor fusion data.

V. Trajectory Optimization

For finding optimal trajectories I follow the approach described in [1]. At every time step a multitude of trajectories is generated by varying the duration T and the target velocity $\dot{s}$. For lane changing trajectories, the target d-coordinate is also (discretely) varied, as well as the duration of the lane change maneuver. These candidate trajectories are then evaluated using the following cost function:

$$J_{traj} = k_v J_v + k_a J_a + k_j J_j + k_\rho \max_i(J_\rho(i)) + k_{\Delta d} J_{\Delta d} \tag{13}$$

Here, J_v, J_a and J_j are terms that penalize the deviation from the target velocity, the maximum acceleration and the maximum jerk, respectively. They are calculated by integrating the respective cost functions over the planning horizon time T_H, which is set to $8s$.

$$J_v = \frac{1}{T_H} \int_0^{T_H} \left(\sqrt{\dot{s}(t)^2 + \dot{d}(t)^2} - v_{max} \right)^2 dt \tag{14}$$

$$J_a = \frac{1}{T_H} \int_0^{T_H} \ddot{s}(t)^2 + \ddot{d}(t)^2 dt \tag{15}$$

$$J_j = \frac{1}{T_H} \int_0^{T_H} \dddot{s}(t)^2 + \dddot{d}(t)^2 dt \tag{16}$$

The safety distance cost of the candidate trajectory in relation to the predicted trajectory of the i-th other vehicle is calculated by integrating an exponential cost function:

$$J_{\rho,i} = \frac{1}{T_H} \int_0^{T_H} j_{\rho,i}(t) dt \tag{17}$$

$$j_{\rho,i}(t) = \begin{cases} 0 & d > d_{min} \vee \rho(t) > \rho_s(v(t)) \\ 1 + \frac{1 - c_{min}}{1 - e^{-\alpha \rho_s(v)}} \left(e^{-\alpha(\rho(t) - \rho_0)} - 1 \right) & \text{otherwise} \end{cases} \tag{18}$$

$$\rho(t) = s(t) - s_i(t) \tag{19}$$

$$\rho_s(v) = v \cdot t_s + \rho_0 \tag{20}$$

Here $\rho_s(v)$ is the velocity-dependent safety distance that is calculated based on a constant-time rule with $t_s = 2s$. The parameter α can be used to tune the steepness of the cost function and thus fine-tune how close the path planner will steer the car to other vehicles. The parameter c_{min} determines the residual cost at the safety distance, while ρ_0 determines the absolute minimum acceptable distance which is set to approximately one car-length. When the lateral distance is greater than a safety threshold, the two cars drive on different lanes and the cost is set to zero. Also, when the distance is greater than the safety distance $d_s(v)$ the cost is set to zero.

The cost $J_{\Delta d}$ uses a similar cost function as J_ρ but penalizes the lateral offset of the car in relation to the current lane. Thus this cost term will encourage the path planner to stay near the center of the lane and to prefer quick lane change maneuvers:

$$J_{\Delta d} = \frac{1}{T_H} \int_0^{T_H} 1 + \frac{1 - c_{min}}{1 - e^{\beta(d_0 - d_1)}} \left(e^{\beta(d - d_1)} - 1 \right) dt \tag{21}$$

After the path planner has generated a collection of candidate trajectories, it adds these together with the predicted trajectories of the nearby other cars to an instance of the class `TTrajectoryCollection`. This object then updates the total cost of each candidate trajectory and determines the one with the minimum cost.

VI. State Machine

I implemented an object-based state machine (`TStateMachine`) to handle the different vehicle states (`TVehicleState`). For this, it maintains a queue of currently active states with the top of the queue being the current state. Every time the vehicle controller is called by the simulator, the trajectory is first updated with the previous coordinates in order to ensure its smoothness. Then the state machine executes the current vehicle state, which returns an updated trajectory along with a pointer to the *next* active state. This pointer can have one of three different values:

1. When the current state returns a `this` pointer, it remains the current state.
2. Returning a `nullptr` signifies that the current state is finished. The previous (or *parent*) state will become current again.
3. If the current state returns a pointer to a newly constructed state, it thereby spawns a new sub-task which will then be current.

a. Keep Lane State

The `TKeepLaneState` is the default state of the car. It generates velocity keeping trajectories that allow the car to keep the lane close to the speed limit, while also keeping a safe distance to the leading cars. In addition, this state will generate candidate trajectories that change to an adjacent lane and to the next-to-adjacent lane, if applicable.

If the car has to follow a slowly leading vehicle, the cost of this trajectory will increase due to J_v increasing. As soon as a trajectory with a different target lane has a lower total cost, the lane keeping state will construct and return an object of type `TChangeLaneState`, which will then execute the lane change.

b. Change Lane State

The `TChangeLaneState` is responsible for executing a safe lane change maneuver. When this state is executed for the first time after construction, the car has already started to move towards the target lane. The lane changing state will generate target trajectories with varying target velocities and durations in the s-direction. For the lateral direction, a quintic polynomial is generated with the target d-coordinate in the center of the target lane and with a fixed lane change duration.[2] For all of these candidate trajectories, the predicted trajectories of all nearby cars in the traversed lanes are considered for evaluating the safety distance cost J_ρ.

In addition, fall-back trajectories are generated that return the car to the original lane. These trajectories enable the car to react to other vehicles' sudden lane changes while executing the maneuver. The safety distance cost of the best lane change trajectory is compared to that of the best fall-back trajectory. Should

[2]The lane change duration is fixed to one value in order to limit the number of generated candidate trajectories.

the safety distance cost of the lane change increase, the fall-back trajectory will be selected and the safer lane will be set as the new target lane, thus avoid collisions with other cars mid-maneuver.

As soon as the lateral Frenet coordinate d is close to its target value, the change lane state will return the `nullptr`. Consequently the lane change maneuver will end, the object is deallocated, and trajectory planning is returned to the parent keep lane state object.

VII. Results

The implemented path planner is able to safely navigate through highway traffic near the speed limit. Owing to the large planning horizon of 8s, the car slows down or switches lane with foresight when approaching a slower leading car from behind. It also keeps clear of faster cars that come up from behind. An example of a lane changing maneuver is shown in Fig. 1. In the maneuver shown, the lane change is interrupted by another car that overtakes on the middle lane. Because the lane changing trajectory has become unsafe, the controller steers the car back to the original lane and lets the overtaking car pass before initiating and completing the lane change in a second attempt.

Figure 1: *Example of a fall-back maneuver during lane changing. The car initiates a lane change maneuver (1-2) but then resets the target lane to the original lane (3). It then safely returns to the original lane (4-5). The car that caused the fall-back maneuver is seen overtaking on the middle lane in (6-7). After the red car has passed, a new lane change maneuver is initiated and successfully completed (8-12).*

VIII. Outlook

There are many ways in which the path planner could be improved. More states could be added to the state machine that would implement following of leading vehicles at a predetermined safety distance or merging into an adjacent lane between other cars. Sometimes the car gets stuck behind a slow car with a second car slightly ahead on the middle lane. If the planner would slow down for a moment, it could direct the car to the third lane, overtaking both other cars. This kind of behavior could be implemented by adding a variable start time to the lateral direction polynomials of lane change candidate trajectories.

The planner also occasionally has difficulties with the erratic driving behavior of the other cars. They tend to violently slow down and accelerate when trailing a slower car. While such a driving behavior would

be uncommon in the real world, a robust path planner should not be disturbed by this. Maybe the planner could internally mark dangerously driving vehicles and keep an increased safety distance in order to stay safe.

References

[1] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, "Optimal trajectory generation for dynamic street scenarios in a frenet frame," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pp. 987–993, IEEE, 2010.

[2] "Frenetserret formulas - wikipedia." `https://en.wikipedia.org/wiki/Frenet%E2%80%93Serret_formulas`. (Accessed on 08/12/2017).

[3] "Gradient descent - wikipedia." `https://en.wikipedia.org/wiki/Gradient_descent`. (Accessed on 08/12/2017).