Highway Path Planning Writeup

Kun L.

2021/12/07

Contents

1 System Overview

The entry point of the system is in **system.cpp**.

This highway driving system takes in a map of the virtual highway and realtime sensor fusion data from the simulator and generates waypoints such that the ego vehicle in the simulator and automatically drive on this virtual highway safe and fast.

2 Map

The entry point of the module is in **map.cpp**.

A high way map is provided by the project. It contains a set of reference waypoints. Each waypoint in the list contains $[x, y, s, dx, dy]$ values. x and y are the waypoint's map coordinate position, the s value is the distance along the road to get to that waypoint in meters, the dx and dy values define the unit normal vector pointing outward of the highway loop.

The waypoints in the original map is rather sparse. Because we are planning in Frenet frame, after planning and converting the Frenet points back to Cartesian points, the resulting trajectory will be very zigzagy. Because of this, during the initialization of the system, the waypoints are further interpolated using spline to be denser.

During initialization time, we pre-compute a set of 4 spline functions from the original data. These 4 spline functions will map s into $[x, y, dx, dy]$ respectively. The related code is in map.cpp, starting from line 49.

And during runtime, after planning is finished and we get a set of waypoints, we can use these pre-computed spline functions to convert the waypoints into Cartesian frame as below,

$$x = \text{Spline}_{s,x}(s) + d \cdot \text{Spline}_{s,dx}(s) \tag{1}$$

$$y = \text{Spline}_{s,y}(s) + d \cdot \text{Spline}_{s,dy}(s) \tag{2}$$

Such that we can get a very accurate $[x, y]$ coordinates back in the Cartesian world. The code is in **map.cpp**, line 174.

3 Tracker

The entry point of the module is in **tracker.cpp**.

The tracker's responsibility is rather simple. It takes in the perceptions and create a dictionary where the key is the vehicle ID, and the value is the

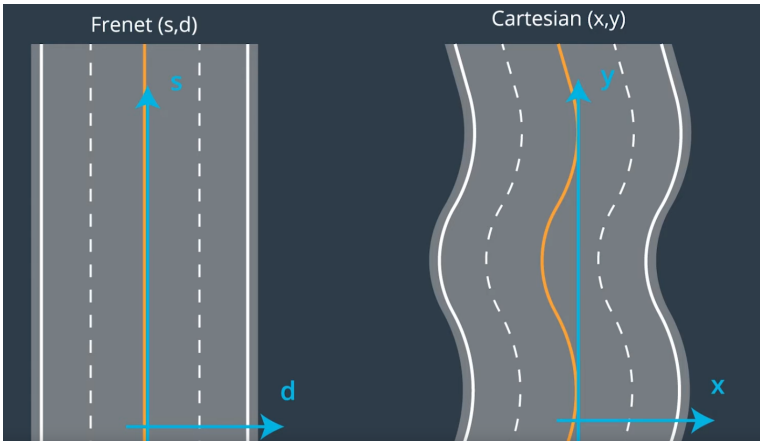

Figure 1: Frenet frame. Image Courtesy: Udacity CarND

vehicle entity. See **vehicle.cpp** for more details about vehicle class. Each vehicle will have in the sensed position and velocity, and these information will be used to predicting the vehicle's future kinematics for downstream tasks like collision checking.

4 Path Planning

Both behavior and path planning are done in the Frenet frame, which is a vehicle's local frame. s is the longitudinal direction and d is the lateral direction, an illustration of the Frenet frame looks like this 3. The benefit of doing planning in Frenet frame is that it drastically simplifies the planning process, since it removed the non-linearity from the trajectory generation and postponed it to downstream process, that is the frame conversion from Frenet to Cartesian, which can easily done.

The process is described as follows. We get the current ego configuration from the simulator, and convert them into Frenet frame. In the Frenet frame, for every lane in the same direction adjacent to the ego lane, we do 2 1D jerk minimizing trajectory generations and find multiple trajectories in both s and d direction separately. Then we can find the optimal combination of both direction as the optimal 2D jerk minimizing trajectory in that lane w.r.t. some cost functions. Then the trajectories from every lane will be compared and a final optimal trajectory will be selected. Then it evaluated into a set of waypoints in Frenet frame. Finally those Frenet waypoints will be converted back into Cartesian frame and send to simulator for execution.

4.1 Behavior Planning

The entry point of the module is in **ptg.cpp**. The behavior planning part is together with trajectory generation. See **ptg.cpp:754**. Notice that **GenerateTrajectoryPy** is only used for debugging and developing purpose. **GenerateTrajectoryCpp** is should always be used.

In the same file, starting from line 541, the behavior planning begins. It takes in the current ego configuration and all the other tracked vehicle information from tracker and decides the longitudinal and lateral behaviors.

Longitudinal Behaviors

1. Cruising, when there is no other vehicle in the lane or the vehicle is far away
2. Following, when there is leading vehicle and the distance to it is neither too far nor too close
3. Stopping, when there is leading vehicle and the vehicle is too close

Lateral Behaviors

1. Lane keeping
2. Left lane changing
3. Right lane changing

For every lane, behavior planning logic will generate a set of longitudinal behaviors and lateral behaviors.

For instance, if the vehicle is in the center lane, and there are no other vehicles at all, it might generate the behavior like,

1. Lane 0 (left lane), lon behavior: cruising, lat behavior: left lane changing
2. Lane 1 (ego lane), lon behavior: cruising, lat behavior: lane keeping
3. Lane 2 (right lane), lon behavior: cruising, lat behavior: right lane changing

Another example would be, ego is on lane 0, there is no other vehicle on lane one, but there is a vehicle not too far on lane 1, it might generate a behavior like,

1. Lane 0 (ego lane), lon behavior: cruising, lat behavior: left lane changing
2. Lane 1 (right lane), lon behavior: following, lat behavior: lane keeping

After each behavior is determined (no matter it's longitudinal or lateral), there will be a corresponding 1D trajectory generated for it. And for each lane, eventually there will be a combination of best longitudinal and lateral trajectory being chosen as the best 2D trajectory will is drivable for the ego.

The 1D trajectory generation can be found in class **PolynomialTrajectoryGenerator::Impl** and the combination logic can be found as **PolynomialTrajectoryGenerator::Impl::GetOptimalCombination**. The details about how to generate a 1D trajectory is described as below in the next section.

4.2 Jerk Minimizing Trajectory (JMT)

The polynomial implementation can be found in **math.h** and the trajectory calculation logic can be found in **jmt.cpp**.

4.2.1 1D Case

For longitudinal behavior like following, and all lateral behaviors, because of final configuration is clear which means we have full constraints. We can solve the trajectory solved as below.

From basic physics, we have the 1D trajectory (position) equation $s(t)$ w.r.t. time t as,

$$s(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5 + \text{H.O.T.}, t \in [t_i, t_f]$$

and in a more compact form,

$$s(t) = \sum_{i=0}^{\infty} \alpha_i t^i$$

By differentiating this equation we can easily also find the velocity $\dot{s}(t)$, acceleration $\ddot{s}(t)$, $\dddot{s}(t)$. By stacking these quantities in a vector $\boldsymbol{s}(t)$, we can define a 1D configuration of a vehicle. When the initial configuration $\boldsymbol{s}_i$ as $\boldsymbol{s}(t_i)$, final configuration $\boldsymbol{s}_f$ as $\boldsymbol{s}(t_f)$ and time $t_\delta = t_f - t_i$ is given, the question is that can we find a trajectory which can satisfying the these given constraints?

Here, the initial configuration is $\boldsymbol{s}_i = \begin{bmatrix} s_i & \dot{s}_i & \ddot{s}_i \end{bmatrix}^T$, the final configuration is $\boldsymbol{s}_f = \begin{bmatrix} s_f & \dot{s}_f & \ddot{s}_f \end{bmatrix}^T$, Moreover, starting time we have $t_i = 0$ such that $t_f = t_\delta$. When presenting the trajectory in this polynomial form, we need to solve for all α_i. Furthermore, in order to find a comfortable trajectory

for passengers, we also need to find a trajectory which can minimizes the total squared jerk as the objective function,

$$\int_{t_i}^{t_f} \dddot{s}(t)^2 \mathrm{d}t$$

One can show that, in order to find the minimizer, we need $\frac{\mathrm{d}^n s}{\mathrm{d}t^n} = 0, \forall n \geq 6$, which means $\alpha_6, \alpha_7, ... = 0$.

Now the 1D JMT $s(t)$ becomes,

$$s(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5$$

We have 6 coefficients (6 tunable parameters) we need to solve. By further differentiating this equation to find the equations for velocity $\dot{s}(t)$ and $\ddot{s}(t)$ explicitly,

$$s(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5 \tag{3}$$
$$\dot{s}(t) = \alpha_1 + 2\alpha_2 t + 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4 \tag{4}$$
$$\ddot{s}(t) = 2\alpha_2 + 6\alpha_3 t + 12\alpha_4 t^2 + 20\alpha_5 t^3 \tag{5}$$

now We can easily see that 3 boundary constraints and 6 unknowns, which is unsolvable. However, we can further exploit that because we know that the trajectory should start from $t_i = 0$, we can immediately solve for the first 3 parameters by,

$$\alpha_0 = s(0) \tag{6}$$
$$\alpha_1 = \dot{s}(0) \tag{7}$$
$$\alpha_2 = \frac{1}{2}\ddot{s}(0) \tag{8}$$
$$\tag{9}$$

Right now we have 3 equations and remaining unknown which we can build up a linear system and solve it. We right now can substituting $t = t_f$ in and get,

$$s_f = s_i + \dot{s}_i t + \frac{1}{2}\ddot{s}_i t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5 \tag{10}$$
$$\dot{s}_f = \dot{s}_i + \ddot{s}_i t + 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4 \tag{11}$$
$$\ddot{s}_f = \ddot{s}_i + 6\alpha_3 t + 12\alpha_4 + 20\alpha_5 t^3 \tag{12}$$

We can rewrite it in a matrix form as,

$$\boldsymbol{b} = \begin{bmatrix} s_f - (s_i + \dot{s}_i t + \frac{1}{2}\ddot{s}_i t^2) \\ \dot{s_f} - (\dot{s}_i + \ddot{s}_i t) \\ \ddot{s_f} - (\ddot{s}_i) \end{bmatrix}$$

$$\boldsymbol{A} = \begin{bmatrix} t^3 & t^4 & t^5 \\ 3t^2 & 4t^3 & 5t^4 \\ 6t & 12t^2 & 20t^3 \end{bmatrix}$$

$$\boldsymbol{x} = \begin{bmatrix} \alpha_3 & \alpha_4 & \alpha_5 \end{bmatrix}^T$$

Here $t = t_f$. And $\boldsymbol{x}$ can be solved easily by any linear solver, here for the sake of simplicity use inversion.

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

Note that, even though we can almost precisely define the final configuration, we can not figure out the exact time for the best trajectory before hand. Because of this, we will have a list candidate t that as input and generate a set of trajectories for later validation. More concretely, in the implementation, we typically use ranges like $\Delta t \in [1.0, 5.0]$. For lateral planning, not only time is sampled in this way, but also a small set of lateral offset as well for increase the chance of finding collision free path. Typically, $\Delta d \in [-0.05, 0.05]$ where Δd is the offset from the target lane center.

4.2.2 1D Case w/o Full Constraints

For behaviors like cruising and stopping, we typically don't care about where the ego vehicle will end up with as long as the result trajectory can satisfy the final constraint of velocity and acceleration.

Which means we only have 2 equations but we want to solve to 6 parameters. For the first 3 parameters, the computation if the same. However for the later 3 parameters, since we only 2 equations, we can not solve for 3 unknowns. By exploiting the transversality condition, we can set $\alpha_5 = 0$, which means we have 2 equations and we are solving 2 unknowns α_3, α_4.

In this case,

$$\boldsymbol{A} = \begin{bmatrix} 3t^2 & 4t^3 \\ 6t & 12t^2 \end{bmatrix}$$

$$\boldsymbol{b} = \begin{bmatrix} \dot{s_f} - (\dot{s}_i + \ddot{s}_i t) \\ \ddot{s_f} - (\ddot{s}_i) \end{bmatrix}$$

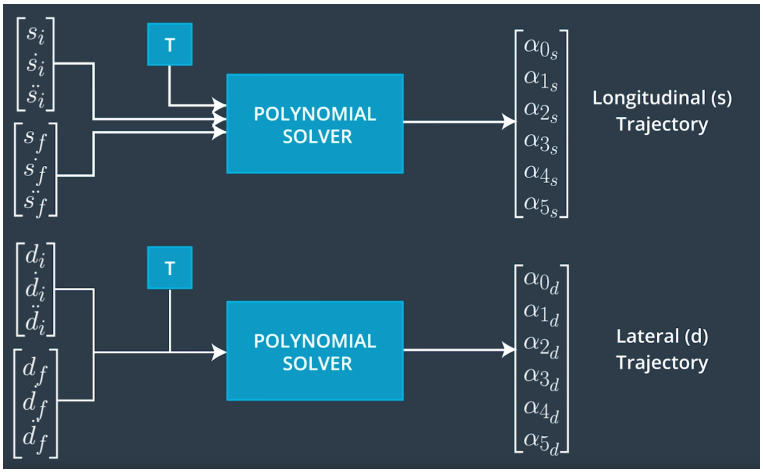

Figure 2: 2D JMT overview. Image Courtesy: Udacity CarND

and we can still solve it by,

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

Finally, after the calculation, because we might ended up using a very large execution time in order to find a valid trajectory like 6, 7, or even 8 seconds depending on the parameters set by user. This will result in hundreds of waypoints in the end since the interval between waypoints are 0.02 seconds. So a time clipping operation is done at the end of calculation. See **JMT::Solve_5DoF**.

4.2.3 2D Case

The trajectory is computed in the longitudinal axis s and lateral axis d Frenet frame. The strategy is that we can compute the 1D JMT in each direction and then combine them together. Here we can see the illustration 8.

4.2.4 Trajectory Validation

The validation code for 1D case can be found in function **JMTTrajectory1d::Validate**. The basic idea is to evaluate the configuration generated $\forall t \in [t_i, t_f]$, and check at that instant time, whether the velocity, acceleration or jerk is out of bound or not. This is binary decision which means as

long as on of the condition is unfulfilled, the trajectory is invalidated. Some goes for **JMTTrajectory2d::Validate**.

4.2.5 1D Trajectory Cost Computation

For every 1D JMT trajectory generated above, there will be a set of cost functions evaluating its configurations inspired by [1]. The implementation can be found in **JMTTrajectory1d::ComputeCost**.

4.2.6 Optimal Combination

This step is the final step for trajectory planning. It will find the optimal combination for each lane. For every lane, even though there might be multiple longitudinal behaviors, we indistinguishably put them into the evaluation process and pick the best one. Note that for each longitudinal behavior there will be only one trajectory be selected (currently the fastest one), instead of letting multiple trajectories for each behavior go through. For instance, for lane 1, 2 longitudinal behaviors and 1 lateral behaviors will be taken into consideration, which means the number of combinations will be only 2, and we pick the best one.

The reason for this strategy is simple, because it's efficient in computation and fast in speed. Since all of the invalid trajectories are hard filtered out, so it's safe and reasonable to do so.

4.2.7 Collision Checking

The implementation can be found in **collision_checking.cpp**. Given vehicle kinematics and trajectory, we can evaluate both up to the trajectory time, and see if the distance between 2 points are too close or not.

Since in order to do the collision checking properly we need at least size of the vehicle and we can overlap 2 rectangles representing the 2 vehicles and see their penetration using GJK, etc. However since these information are not given and it might be an overkill for this project and miss the actual important part which is the trajectory generation, a simple trick is used here.

The collision model is simplified for the vehicles, it's a circle instead of a more realistic long rectangle. Because of this, when ego is driving in the middle lane with 2 vehicles on each of the left right lane, there is a big chance that the 2 collision circles will cover a big portion of the middle lane. It will cause the collision checking to fail constantly and stop the ego. Because of this, only the car on the lanes to be planed on will be used for collision

checking. For instance, if we are planning for ego lane, say it's lan 1, then only vehicles on lane 1 will be checked and all other vehicles will be ignored. By doing this we can avoid the failure mentioned above.

4.2.8 Waypoints Evaluation

This step is to evaluate the best trajectory from previous step into a set of waypoints in Frenet frame, and the converting all of them into Cartesian frame.

4.2.9 Caching

The trajectory will be cached to computed the expected starting configuration for the next round of planning. See **system.cpp**.

5 Implementation

Implementation can be found in `https://github.com/kunlin596/PathPlanning`.

References

[1] Moritz Werling et al. "Optimal trajectory generation for dynamic street scenarios in a frenet frame". In: *2010 IEEE International Conference on Robotics and Automation*. IEEE. 2010, pp. 987–993.