Path planning for self-driving cars in a virtual highway

Madhu Dev · Just now · 8 min read

> *This project illustrates the concept of behavior planning and decision-making for autonomous cars executed as part of Udacity Nanodegree for Self-Driving cars.*

Snapshot from the simulator

. . .

This project in my experience is the most challenging task of the entire course. The reasons for that is amply clear as this involves programming the highest decision making module of an autonomous vehicle and the challenges become pretty evident. My Blog Post will describe my implementation in C++, key points of the project and how my implementation addressed those points.

Goal of the project

The goal of the project to make a ego car drive around a virtual highway in dense traffic by adhering to following constraints for minimum 4.32 Miles.

- No collision at any time with other vehicles
- Maximum speed of 50 MPH (~ 80 KMH)
- Maximum acceleration of 10 m/s^2
- Maximum jerk of 10 m/s^3
- Vehicle cannot be in between lanes for more than 3 seconds
- Vehicle cannot go outside the 3 lanes of the highway
- Vehicle cannot drive on the wrong side of the highway

My implementation of the planning code should output the trajectory of planned path to the term 3 simulator provided by Udacity, which will then visualize the ego behavior and output dynamic parameter like velocity, acceleration, jerk etc.

Architecture of the Planning Module

Typical Path planning subsystem for autonomous vehicles consist of prediction, trajectory planning and behavioral planning. The execution timeline and flow of Information between the modules is depicted in the picture below.

Information flow between planning and motion system (Source:Udacity)

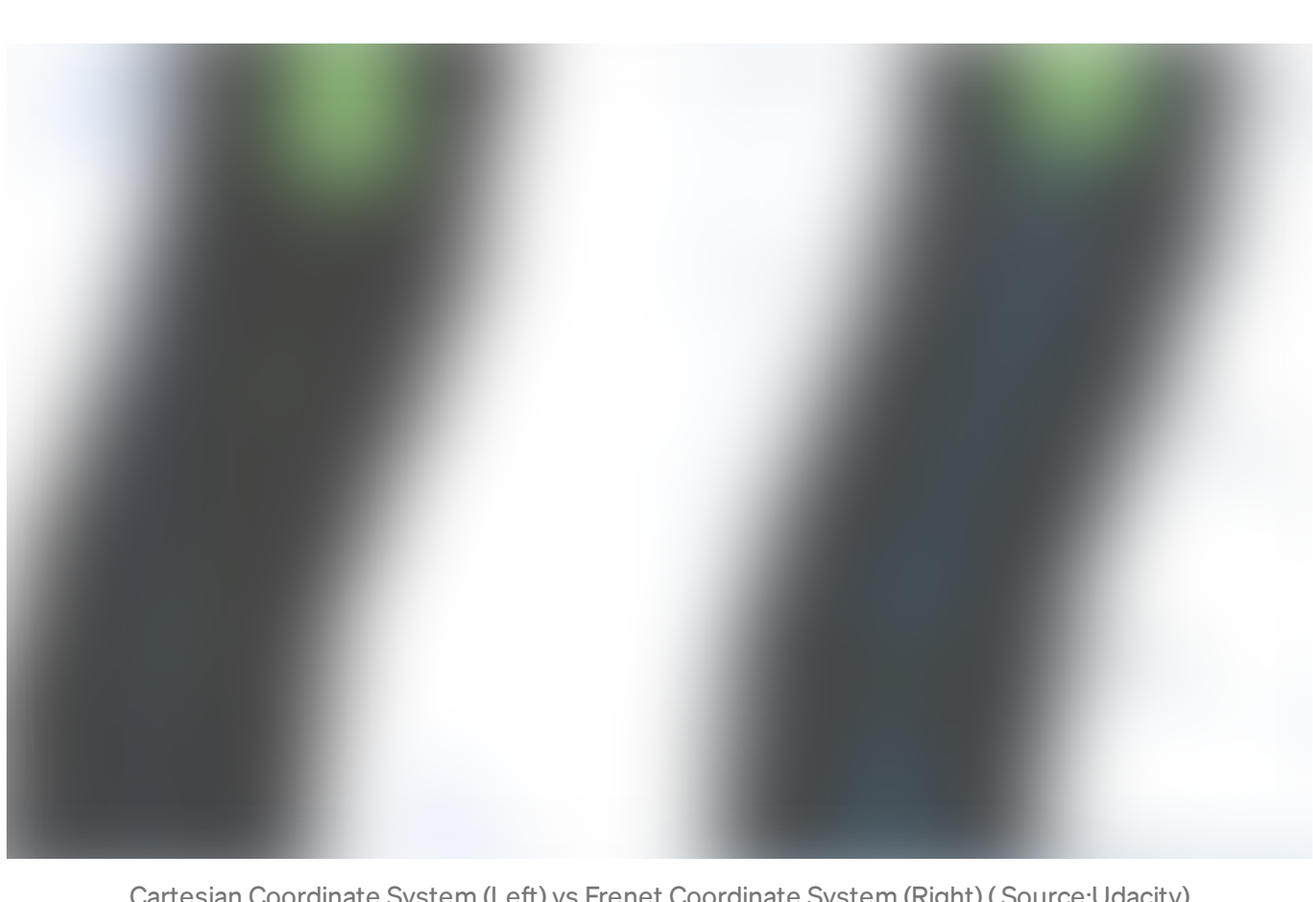
The function of individual layers are briefly discussed below.

- ***Motion Control:*** *its* responsible for moving the car and following a *reference* trajectory as closely as possible. This layer operates with the fastest time scale
- ***Sensor Fusion:*** responsible for merging sensor outputs (e.g. RADAR + LIDAR).
- ***Localization:*** responsible for locating the vehicle with as much precision as possible on the map, and where other entities (e.g. other cars) are with respect to our vehicle
- ***Trajectory:*** responsible for computing trajectories based on a set of constraints (e.g. speed, distance, lane, jerk, etc.)
- ***Prediction:*** responsible for identifying and predicting near-future changes in the scene based on vehicle's current trajectory, other vehicles' trajectories, and elements on the (e.g. traffic lights).
- ***Behaviour:*** responsible for collating in all information from lower layers and decide on future state to transition along with the trajectory to follow.

Lets look at key concepts and how they are implemented in my code.

Frenet Coordinate System

The traditional Cartesian coordinate system is used to represent position of a car on a road or map. While roads are mostly curvy, it becomes mathematically difficult to describe common driving behaviors. For example , it is not easy to interpret in Cartesian coordinate system if two cars are in same lane. So Self-Driving cars use additional coordinate system, Frenet Coodrinate system. In this system, *s* coordinate represents the distance along the road (longitudinal displacement) and d coordinate represent side-to-side position on the road (lateral displacement). Consequently, Frenet coordinate system can better represent the lane system of roads.



Cartesian Coordinate System (Left) vs Frenet Coordinate System (Right) (Source:Udacity)

In this project, the simulator takes in position data in Cartesian coordinate system. However it can output position in Cartesian and Frenet Coordinate system. The interoperability and transformation between the two coordinate systems will be discussed in section Trajectory Planning.

Prediction

A prediction module uses a map and data from sensor fusion to generate predictions for what all moving objects in vicinity likely to do. The ultimate goal of this module is to provide possible trajectories that the moving objects may probably follow. These trajectories along with their corresponding probabilities are then considered by behavior planning model to decide the behavior of ego car.

This module is heavily simplified in this project by using a Linear point model or constant velocity model for all the vehicles around the ego car. Given the duration for which the prediction is required, the new position of all the vehicles are calculated. In this case, only one trajectory is calculated with maximum probability of 1.The code for this functionality is available in *generate_predictions()* of the *class::Vehicle*

Trajectory Planning

Trajectory planning module creates a safe trajectory for the ego car to follow. A Trajectory is series of configurations of vehicle position over a short time horizon. Usually a motion planning algorithm creates the trajectory considering many factors like configuration space, start and final configuration, constraints like physics of vehicle, traffic, map, jerk minimizing, acceleration and velocity limits etc.

The basis of trajectory planning in this project is the map of highway given as text file "highway_map.csv". It is a cloud of points with each point described as both in cartesian and Frenet coordinate system. Format is as follows [X,Y,s,dx,dy]. The getXY helper function calculates the X and Y value given s and d in frenet coordinate system. A part of this map is isolated using the last 2 points in previous path and three more points selected over 90 m range. These five points are transformed from map coordinate system to vehicle coordinate system and interpolated using open source Spline function. The end result will be a spline starting from last position in previous path extending over 90 m along the track. The lane to follow will be provided by behavior planning module. The code for this entire functionality can be found in *calculate _map _XYspline_for_s()* in *MapPath.cpp*

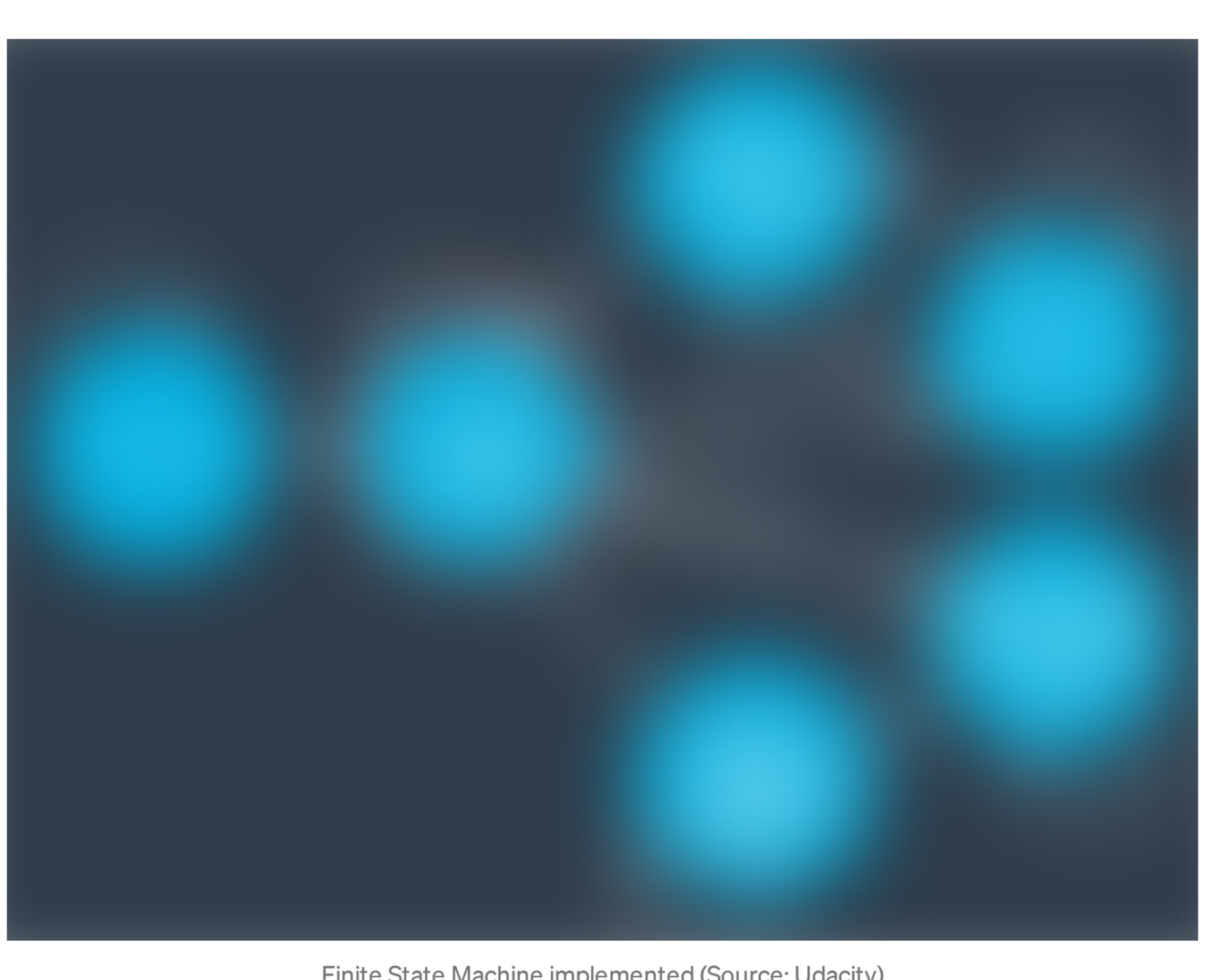
To facilitate smoother lane changes, the first selection point should be quite far from the last 2 points of previous path. This controls the lane change trajectory. If the points are too close, then a sharp lane change will be the result with collision with vehicle ahead. Hence the selection of points is different for lane changes and for cases with no lane changes.
With help of the resulting spline and the reference velocity computed by behavior module, the piece of spline is discretized such that two successive points will not have more than 0.5 m spacing to limit the velocity to 50 MPH or 22.3 m/s. This is based on the fact that the simulator executes each movement in 20ms. The discretized points are again transformed back to map coordinate system and fed to the simulator. The code for this functionality can be found *generate_map_path_with_traffic()* in *PathGenerator.cpp*

Behavior Planning

The top most decision making module of Self-Driving cars makes the final decision on the next step for the ego vehicle based on the map, route and predictions. The concept of finite state machine is used select safe behaviors to execute, and design optimal, smooth paths and velocity profiles to navigate safely around obstacles while obeying traffic laws.

In my implementation, the behavior planning module evaluates different states stated below with help of cost functions and selects the trajectory corresponding to that state with minimal cost.

The different Finite states considered for this project are shown in the picture below.



Finite State Machine implemented (Source: Udacity)

- **Keep Lane:** In this state the ego vehicle will just follow the same lane with constant velocity
- **Prepare for Lane Change Left :**In this state the ego vehicle continues on the same lane but with intended lane as a left lane. The intended lane helps in calculations to check if lane change to left is feasible. For example, Left lane available, car too close in left lane
- **Prepare for Lane Change Right:** In this state the ego vehicle continues on the same lane but with intended lane as a Right lane. The intended lane helps in calculations to check if lane change to right is feasible. For example, right lane available, car too close in right lane
- **Lane Change Left :** In this state the ego vehicle executes the actual lane change from current lane to final lane on left
- **Lane Change Right :**In this state the ego vehicle executes the actual lane change from current lane to final lane on right

Behavior planning implementation is available in *Vehicle.cpp* . The function *successor_states()* creates all these states. All the above states are coded in the functions keep_lane_trajectory(), prep_lane_change_trajectory(), lane_change_trajectory(). These functions after checking for their respective conditions will call the function *get_kinematics()* to calculate new position, velocity and acceleration. All these trajectories are then evaluated in test_func() and trajectory with minimal cost is returned.

Cost Functions

Cost function are mathematical functions that promotes or hinders specific behaviors of ego vehicle. The various trajectories or states discussed above will be evaluated and assigned a value using the below mentioned cost functions. These cost values are then multiplied with their user given respective weights to calculate the final weighted cost value for each trajectory under consideration. Finally the ego vehicle will follow the trajectory with least value.

- ***goal_lane_cost*** : This function imposes higher cost on trajectories with vehicle in first or third lane and zero cost if the vehicle is in middle lane. Being in middle is efficient, as ego car has option to change to first or third lane during heavy traffic. It reduces the chances that ego car get stuck in traffic and hence possibility of accident.
- ***inefficiency_cost*** : This function imposes higher cost on trajectories with intended and final lane that have traffic slower than the vehicle's target speed. The function promotes lane change to maintain vehicle speed.
- ***lane_change_safety_cost*** : This function imposes higher cost on trajectories that attempt lane change when speed difference between current lane and goal lane is less than 7 Kmph. This limits potential for accidents during lane change in heavy traffic
- ***lane_chane_safe_dist_cost*** : This function imposes higher cost on trajectories that attempt a lane change when front cars in both goal lane and current lane travel at same speed and distance ahead of ego car .This prevents waggling of ego cars

All the cost functions are implemented in *Cost.cpp*. The function *calculate_cost()* calculates weighted mean of the individual cost values. From programming perspective, cost functions replace the traditional if-else statements used in programming. Also they can induce more dynamic behavior and can react to situations that are not foreseen during programming.

Conclusion

After all the effort in programming different modules, it was a joy to see the virtual car make multiple laps without a error. It was a awesome unforgettable experience. The ego car is programmed conservatively for more safety, stability, smooth lane changes than efficiency or aggressive maneuvers to complete a lap quickly. The ego car consistently traveled over 4.32 Miles without any mistake and once completed 28 Miles without any incident. Of course, the ego vehicle does encounter some problems in heavy slow moving traffic on rare occasions, especially when other cars make erratic lane and speed changes.

Full video of ego car completing 4.32 miles around the track is provided here.

Video of ego driving around the track for 4.32 miles

The fact that the simulator requires set of X and Y points to move the ego vehicle has certain consequences. Firstly it was difficult to control the acceleration and jerk. Secondly , for smooth lane changes, jerk optimized trajectories using Quintic Polynomial could not be integrated into the calculations since they operate on Frenet coordinate system.

My Object Oriented implementation of the planning module allows plenty of room to expand the project by adding more cost functions for efficient drive , deriving more functions from existing ones to handle rare traffic incidents. In the future versions, i plan to parameterize many controls which are currently running on hard coded values. The source code of this implementation can be found here.

Self Driving Cars · Motion Planning · Trajectory Planning · Finite State Machine

More from Madhu Dev
Follow
Simulation Specialist, Avid Programmer, Robot and A/E-drive enthusiast