CarND-Path-Planning-Project

Goal: to safely navigate around a virtual highway with other traffic that is driving +-10 MPH of the 50 MPH speed limit

One key point of the project is to design a smooth navigation trajectory. First, it read map waypoints from a CSV data file. I put three waypoints 30m, 60m, and 90m ahead of ego car in Frenet coordinate and transform to global coordinate with the help of map waypoints from CSV data file. Fig. 1 illustrates the three waypoints up to 90 meters ahead the car.

Highway lane has width of four meters. I used “(lane index) * 4 + 2” as the center of lane in Frenet coordinate (“d”). Lane index counts from the double-yellow line (left-most) as zero. Add_way_points() implements the function to add three waypoints.

Then I created a spline by implantation from Tiny Kluge https://github.com/pkelchte/spline adding two previous points plus the three waypoints for smooth transition.

I generated total of 50 points for the trajectory. Besides the previous points, I calculated target distance of 30m in x-axis with spine(30) in y-axis. Target distance magnitude is square root of x value square plus y value square. By dividing target_distance with multiplication of velocity and sample time with meter/s mile/hour conversion, I get N count of ticks.

Target distance = √ (30 * 30 + spline(30) * spline(30))
N = target_distance / (sample_time (0.02 sec) * velocity / 2.24)

Each tick on x-axis is equal to target_distance/N. With a loop, I added each tick with its corresponding spline(x) value. Finally, I applied translation and transformation to make the trajectory in global coordinate. Add_addl_points_to_total() implements this function.

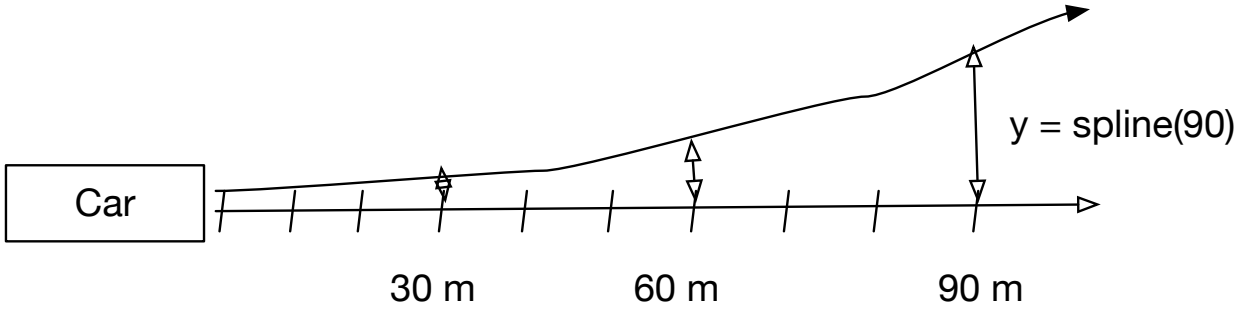

Fig. 1 Waypoints and Spline

I used fusion data to detect distance from our car. When the distance between ego car and the car in front of it is less than safe distance 30m, ego car starts to slow down. It starts to look for lane change if safe. First, we look at left lane if ego car is not at the left most lane. If there are safe distances for both front and behind cars and the front car is faster than slow car in front of ego car on the current lane, then it changes lane. The navigation trajectory described in the prior paragraph calculates the new 50 points trajectory. If left lane is not available, we apply the same algorithm for right lane if ego car is not in the right-most lane. Change_lane() and is_lane_safe() implement the algorithm.

Fig. 2 shows a screenshot from simulator. It ran for more 10 minutes, 6.69 miles without incident.

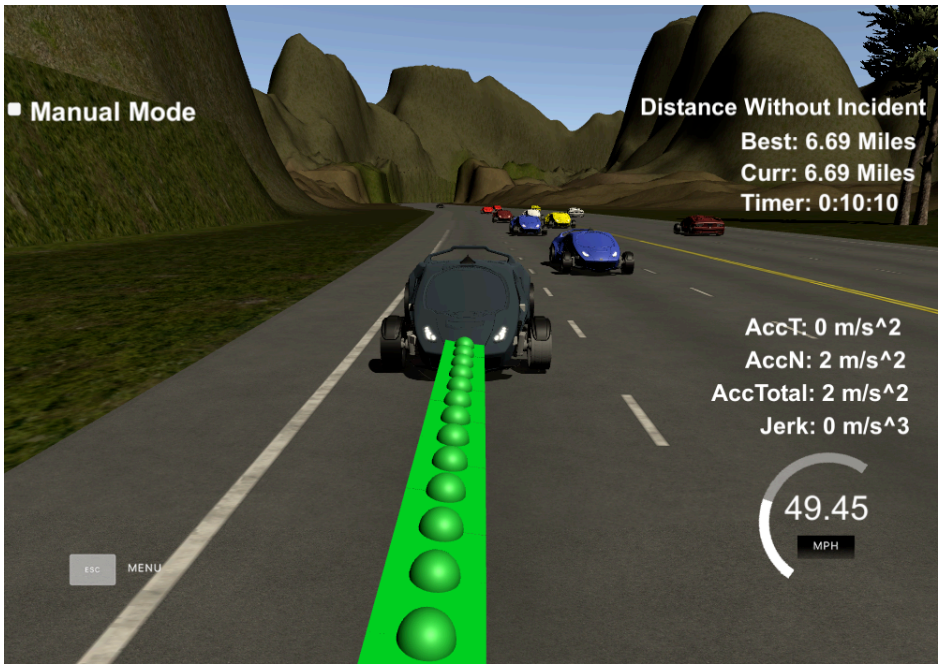

Fig. 2 Screenshot from simulator

Implementation Detail

I create a C++ class Planner to generate vectors of (next_x_vals, next_y_vals) for simulator to drive the car. Below shows code in the main message handling routine from socket.

```
// Create vectors of (next_x_vals, next_y_vals)
vector<double> next_x_vals;
vector<double> next_y_vals;

 // Use planner to figure next points
Planner myPlanner(previous_path_x, previous_path_y, car_x, car_y, car_s, car_yaw);
myPlanner.get_next_points(map_waypoints_x, map_waypoints_y, map_waypoints_s,  end_path_s, sensor_fusion, next_x_vals, next_y_vals);

// Send (next_x_vals, next_y_vals) to simulator
msgJson["next_x"] = next_x_vals;
msgJson["next_y"] = next_y_vals;
```

C++ class Planner
Public Methods:
"**get_frenet_points**" is a test function using Frenet-coordinate and map waypoints to calculate next points in global coordinate. It takes map waypoints (x,y) and s as input parameters. (next_x_vals, next_y_vals) are input and output parameters in global coordinate. (next_x_vals, next_y_vals) are used by simulator to navigate the car.

void **get_frenet_points**(vector<double>& map_waypoints_x, vector<double>& map_waypoints_y, vector<double>& map_waypoints_s, vector<double>& next_x_vals, vector<double>& next_y_vals)

“**get_next_points**” is the primary function of the class to get planned next 50 points for simulator to drive the car. It has the same input and output parameters as get_frent_points(). It uses private functions to smooth navigation and change lane.

void **get_next_points**(vector<double>& map_waypoints_x, vector<double>& map_waypoints_y, vector<double>& map_waypoints_s, double end_path_s, vector<vector<double>>& sensor_fusion, vector<double>& next_x_vals, vector<double>& next_y_vals)

map_waypoints_x, map_waypoints_y, map_waypoints_s: map waypoints (x,y) and s, input only
end_path_s: s value of the end path, input only
sensor_fusion: data of all cars on the road close to our self-driving car, input only
next_x_vals, next_y_vals: (x,y) values in global coordinate, input and output

Private Methods:
“**add_prev_pts**” is a utility function to add previous points into (next_x_vals, next_y_vals). It also adds initial points to (x_vals, y_vals) from previous points to smooth the path and updates reference values of x, y, and yaw angle.

void **add_prev_pts**(vector<double>& x_vals, vector<double>& y_vals, double& x_ref, double& y_ref, double& yaw_ref, vector<double>& next_x_vals, vector<double>& next_y_vals)

x_vals, y_vals: vector of (x,y) for planned path, input and output
x_ref, y_ref, yaw_ref: reference values of (x,y) and yaw angle. Initially, the reference (x,y) is location of the car, input and output
next_x_vals, next_y_vals: vector of (x,y) for simulator to navigate the car, input and output

“**check_car_ahead”** is a utility to check whether there is a safe distance from the front car in the same lane ahead, returning true if too close.

bool **check_car_ahead**(double end_path_s, vector<vector<double>>& sensor_fusion, int sample_count)

end_path_s: s value of the car or the end of prior path, input
sensor_fusion: vector of vector of double, data if all cars in the road, input
sample_count: number of previous points to calculate distance ahead or behind, input

"**change_lane"** is a utility function to change lane if feasible. It uses **is_lane_safe** to decide a particular lane is safe to change to. Here I uses a simply algorithm by checking left lane first and then right lane. Normally, we want to pass a car by the left lane.

bool **change_lane**(double end_path_s, vector<vector<double>>& sensor_fusion, int sample_count)

end_path_s: s value of the car or the end of prior path, input
sensor_fusion: vector of vector of double, data if all cars in the road, input
sample_count: number of previous points to calculate distance ahead or behind, input

"**is_lane_safe"** is a utility function to change a particular lane is safe to change to for cars in front and behind.
bool **is_lane_safe**(int idx, double end_path_s, vector<vector<double>>& sensor_fusion, int sample_count)

idx: index of the lane to change to. It starts zero from the leftmost lane next to double yellow lines and increases one by one to right, input only.
end_path_s: s value of the car or the end of prior path, input only
sensor_fusion: vector of vector of double, data if all cars in the road, input
sample_count: number of previous points to calculate distance ahead or behind, input only

"**add_way_points"** is a utility function to add waypoints into (x_vals, y_vals).

void **add_way_points**(vector<double>& x_vals, vector<double>& y_vals, vector<double>& map_waypoints_x, vector<double>& map_waypoints_y, vector<double>& map_waypoints_s)

x_vals, y_vals: vector of (x,y) for planned path, input and output

map_waypoints_x, map_waypoints_y, map_waypoints_s: (x,y) and s of map waypoints, input only

“**add_addl_points_to_total**” is a utility function to add more points using spline to make total of 50 for simulator.

void **add_addl_points_to_total**(vector<double>& next_x_vals, vector<double>& next_y_vals, double x_ref, double y_ref, double yaw_ref, int count)

x_vals, y_vals: vector of (x,y) for planned path, input and output
x_ref, y_ref, yaw_ref: reference (x,y) and yaw to transform and translate to global coordinate, input only

“**transform_to_car_coord” is a** utility to perform translation and rotation from global coordinate to car coordinate.

void **transform_to_car_coord**(vector<double>& x_vals, vector<double>& y_vals, double x_org, double y_org, double yaw)

x_vals, y_vals: vector of (x,y) for planned path, input and output
x_org, y_org, yaw: reference (x,y) and yaw to transform and translate to car coordinate, input only

Private Variables:
There are two internal states, m_lane for lane number and m_ref_vel for velocity in miles/hour.

```
static int m_lane; // lane number starting from double yellow lines as zero
static double m_ref_vel; // reference velocity miles/hour
```

I also keep some states from simulator.

```
vector<double>& m_previous_path_x;
vector<double>& m_previous_path_y;
double m_car_x;
double m_car_y;
double m_car_s;
double m_car_yaw;
```

A state variable spline is used across private functions.

```
tk::spline m_spline;
```

Future Work

I use simple algorithm for lane pass and state management. It needs more sophisticate algorithm to handle corner cases with smoothness in real world.