PROJECT PLANNING PROJECT

Krishnan Ramaswamy

Table of Contents

1. Objective

The objective of this document is to describe the functional blocks of path planning project and discuss code snippets. Also, this document provides the summary of observations made and list the suggestion for further improvements.

2. Path Planning Foundation

Computing optimal path for Self-Driving vehicle involves multiple modules working together on sensory data and highway map data. Path planning is a motion planning problem. The motion planning problem states that :

Given configurations space, start configuration and goal, constraints such as obstacles, map and traffic, find the best path in dynamic environment.

Start configurations is given to us via localization and sensors. The key components of solving motion problem involves foundations from Search, Prediction, Behavior of other vehicles and pedestrians and trajectory generation.

Search

- **discrete** path and **continuous** path planning.

- Motion planning problem find the best optimal path for an object to reach the end point using minimum cost path
- **continuous** path planning can be considered as serious of discrete motion planning
- A* is a variant of search algorithm which is very efficient and uses heuristics guess to find the optimal path. Free form navigation is the heuristic function used in self driving car.
- Dynamic programming is alternative method to find the best optimal path. It provides technique to find optimal path for multiple end locations
- If the environment is stochastic, then we need to plan for every position

Prediction

- Model based and Data driven prediction. Hybrid approach is used as well
- A prediction module uses a map and data from sensor fusion to generate predictions for what all other **dynamic** objects in view are likely to do.
- Here is the input and output sample

<table>
<tr><td></td><td></td></tr>
<tr><td>


```
"timestamp" : 34512.21,
    "vehicles" : [
        {
            "id"  : 0,
            "x"   : -10.0,
            "y"   : 8.1,
            "v_x" : 8.0,
            "v_y" : 0.0,
            "sigma_x" : 0.031,
            "sigma_y" : 0.040,
            "sigma_v_x" : 0.12,
            "sigma" : 0.03,
        },
        {
            "id"  : 1,
            "x"   : 10.0,
            "y"   : 12.1,
            "v_x" : -8.0,
            "v_y" : 0.0,
            "sigma_x" : 0.031,
            "sigma_y" : 0.040,
            "sigma_v_x" : 0.12,
            "sigma_v_y" : 0.03,
        },
    ]
}
```


</td><td>


```
{
    "timestamp" : 34512.21,
    "vehicles" : [
        {
            "id" : 0,
            "length": 3.4,
            "width" : 1.5,
            "predictions" : [
                {
                    "probability" :
0.781,
                    "trajectory"  : [
                        {
                            "x": -10.0,
                            "y": 8.1,
                            "yaw": 0.0,
                            "timestamp":
34512.71
                        },
                        {
                            "x": -6.0,
                            "y": 8.1,
                            "yaw": 0.0,
                            "timestamp":
34513.21
                        },
                        {
                            "x": -2.0,
                            "y": 8.1,
                            "yaw": 0.0,
```


</td></tr>
</table>

	"timestamp": 34513.71 },

Behavior Planning

The responsibility of Behavior Planning module is to suggest states which are Feasible, sage, legal and efficient by taking input from Map, Route and Predictions

FSM is used in making decisions on transitions. Not all transitions are valid and legal. Self-Driving vehicle domain have its own set of state and valid transitions. Later sections and code snippet will discuss about this more.

In summary, One way to implement a transition function is by generating rough trajectories for each accessible "next state" and then finding the best. To "find the best" we generally use **cost functions**. We can then figure out how costly each rough trajectory is and then select the state with the lowest cost trajectory.

For highway planning, FSM is fine but for more complex planning such as in urban or inner-city road, FSM is not appropriate

Trajectory Planner

The responsibility of Trajectory planner is to take input from Behavior Planner and provide collision free trajectory that is optimal

Putting it all together

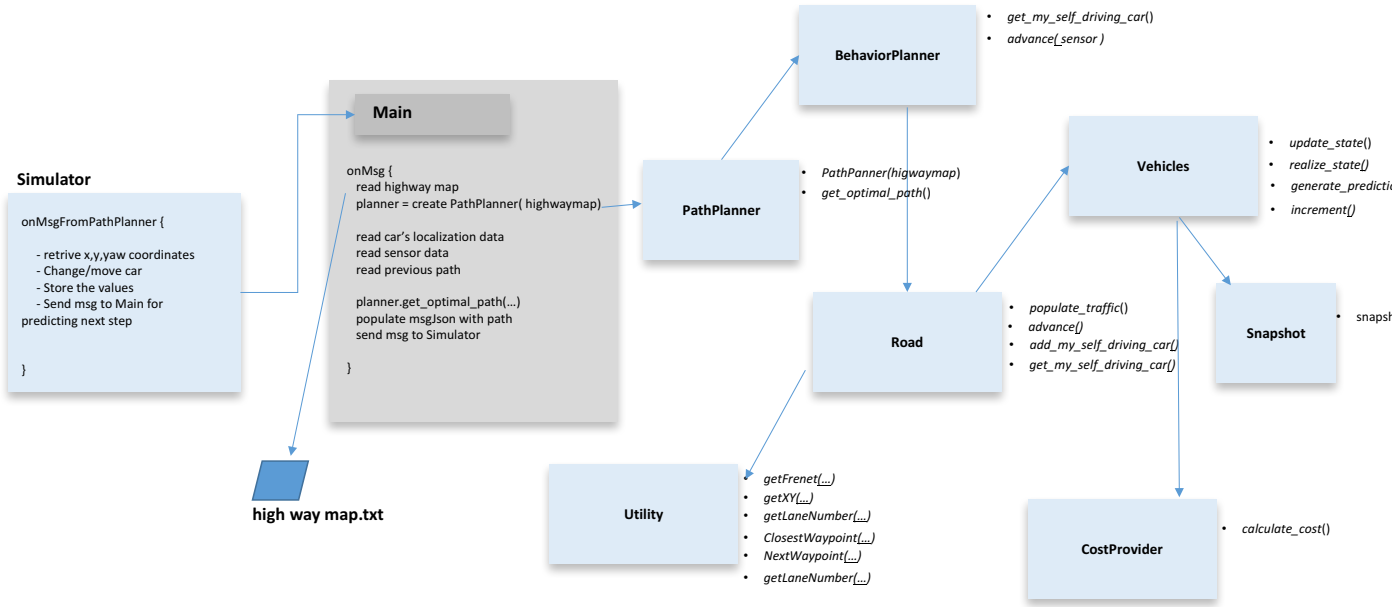
There are many motion planning algorithms.

- Combinatorial methods which deals with Small elements and do not scale well
- Potential Field algorithms consider the obstacles as anti-gravity elements and may stuck in local minima
- Optimal Control is a dynamic method, optimizing the cost function using Numerical optimization method. It is difficult to incorporate all other vehicles's behavior

- In Sampling based method, tot all part of free space needs to be explore and explored ones are kept in graph structure Probabilistic graph search algorithms such as RRT, RRT*, PRM etc

3. Functional blocks

Following diagram provides high level overview of Path Planning components and its relationship. The responsibilities of each object and code snippet is given in the subsequent sections.



4. Classes and Objects

4.1. Main

Step 1)

The responsibility of Main program is to read the waypoints from highway_map.csv and initialize PathPlanner Object. The data consists of a series of data points, each as a global map x/y Cartesian coordinate pair, along with a Frenet coordinate pair of s (distance along the road) and d (distance from centerline). The "d" portion of this coordinate pair is curious in that it is given as an x/y unit vector pair "dx" and "dy". This vector pair is later converted into a unit vector which can be used to calculate distance from the centerline.

```
vector<double> map_waypoints_x;
vector<double> map_waypoints_y;
vector<double> map_waypoints_s;
vector<double> map_waypoints_dx;
vector<double> map_waypoints_dy;
```

../data/highway_map.csv

```
784.6001 1135.571 0 -0.02359831 -0.9997216
815.2679 1134.93 30.6744785308838 -0.01099479 -0.9999396
844.6398 1134.911 60.0463714599609 -0.002048373 -0.9999979
875.0436 1134.808 90.4504146575928 -0.001847863 -0.9999983
905.283 1134.799 120.689735412598 0.004131136 -0.9999915
934.9677 1135.055 150.375551223755 0.05904382 -0.9982554
964.7734 1138.318 180.359313964844 0.1677761 -0.9858252
995.2703 1145.318 211.649354934692 0.3077888 -0.9514547
1025.028 1157.81 243.922914505005 0.3825578 -0.9239317
1054.498 1169.842 275.754606246948 0.3815603 -0.9243439
1079.219 1180.179 302.548864364624 0.3191902 -0.9476907
1102.047 1185.857 326.072883605957 0.1833147 -0.9830543
1127.149 1189.116 351.385223388672 0.09871602 -0.9951157
```

```
PathPlanner planner = PathPlanner(map_waypoints_x, map_waypoints_y, map_waypoints_s,
map_waypoints_dx, map_waypoints_dy);
```

Step 2)

Next, the Main program waits for events from Simulator and read the current car position and previous path - as below

```
h.onMessage(……) {

// Main car's localization Data
double car_x = j[1]["x"];
double car_y = j[1]["y"];
double car_s = j[1]["s"];
double car_d = j[1]["d"];
double car_yaw = j[1]["yaw"];
```

```
double car_speed = j[1]["speed"];

// Previous path data given to the Planner
auto previous_path_x = j[1]["previous_path_x"];
auto previous_path_y = j[1]["previous_path_y"];
// Previous path's end s and d values
double end_path_s = j[1]["end_path_s"];
double end_path_d = j[1]["end_path_d"];

// Sensor Fusion Data, a list of all other cars on the same side of the road.
auto sensor_fusion = j[1]["sensor_fusion"];

    //cont..
   }
```

Step 3)

Next, the Main program invokes PathPlanner to estimate the optimal path

```
vector<double> path_plan = planner.Solve(car_data, sensor_fusion, previous_path_x, previous_path_y, end_path_sd);
```

Step 4) The output from the planner is a vector of x and y Cartesian coordinates, representing the path the vehicle is to follow, which are passed back to the simulator by updating msgJson with next x and y location for the car to move.

```
//.. add (x,y) points to list here, points are in reference to the vehicle's coordinate syst
// the points in the simulator are connected by a Green line
for (size_t i = 0; i < path_plan.size(); i++) {
  if (i % 2 == 0) {
    next_x_vals.push_back(path_plan[i]);
  } else {
    next_y_vals.push_back(path_plan[i]);
  }
}

msgJson["next_x"] = next_x_vals;
msgJson["next_y"] = next_y_vals;

auto msg = "42[\"control\","+ msgJson.dump()+"]";

ws.send(msg.data(), msg.length(), uWS::OpCode::TEXT);
```

4.2. PathPlanner

PathPlanner object is responsible for

Taking waypoint information and storing it in local variables for later reference
Utilize BehaviourPlanner to determine the best next state for my-self-driving-vehicle. The next state will contain information in Frenet coordinate.
Transform Frenet coordinates to map coordinate and then to car local coordinate system

The only information passed into BehaviourPlanner is the sensor fusion data and the vehicle's Frenet s-coordinate values

We need the sensor fusion data obviously, because it contains information about the state of the other vehicles on the road. But we only care about the vehicle's current s-value, because all the other values for the vehicle will be computed by the planner, but it is very important that we know the current s-value.

Once the behavior planner is done processing, all the information needed for generation of the path will be available. At this point, the path generation method is called and the data returned to main.cpp for output back to the simulator

GeneratePath()

first gets the information about the my-self-driving-car vehicle; what lane it is supposed to be in, its s-value, its velocity and acceleration, its target speed, and its state

The code essentially works by taking a starting reference x/y position for the car, and it's current steering angle or yaw value

```
double ref_x = this->car_x;
double ref_y = this->car_y;
double ref_yaw = deg2rad(this->car_yaw);
```

It then finds the previous position of the car, either by using a simple extrapolation based on the car's steering angle

```
if (this->previous_path_size < 2) {
  double prev_car_x = this->car_x - cos(deg2rad(this->car_yaw));
  double prev_car_y = this->car_y - sin(deg2rad(this->car_yaw));
  ...
}
```

...or by getting the information from the previous path, and finding the steering angle from those

```
ref_x = this->previous_path_x[this->previous_path_size - 1];
ref_y = this->previous_path_y[this->previous_path_size - 1];

// and second to last end point
```

```
double ref_x_prev = this->previous_path_x[this->previous_path_size - 2];
double ref_y_prev = this->previous_path_y[this->previous_path_size - 2];
ref_yaw = atan2(ref_y - ref_y_prev, ref_x - ref_x_prev);


// and second to last end point
double ref_x_prev = this->previous_path_x[this->previous_path_size - 2];
double ref_y_prev = this->previous_path_y[this->previous_path_size - 2];
ref_yaw = atan2(ref_y - ref_y_prev, ref_x - ref_x_prev);
```

These points describe a path tangent to the car or to the previous path's endpoint respectively, forming the first point in the new path being generated. The points in both cases are pushed onto a couple of lists which are used in the path generation process.

We then find the Frenet d-coordinates of the lane the my-self-driving-car vehicle is in, and then extrapolate a series of three x/y coordinates which follow the waypoints, but spaced 30 meters apart. These are also placed on the

```
int lane = getLaneFrenet(this->car_lane, 0);
vector<double> next_wp0 = getXY(this->car_s + SPLINE_SPACING, lane, this-
>map_waypoints_s, this->map_waypoints_x, this->map_waypoints_y);
vector<double> next_wp1 = getXY(this->car_s + (SPLINE_SPACING * 2), lane, this-
>map_waypoints_s, this->map_waypoints_x, this->map_waypoints_y);
vector<double> next_wp2 = getXY(this->car_s + (SPLINE_SPACING * 3), lane, this-
>map_waypoints_s, this->map_waypoints_x, this->map_waypoints_y);

ptsx.push_back(next_wp0[0]);
ptsx.push_back(next_wp1[0]);
ptsx.push_back(next_wp2[0]);

ptsy.push_back(next_wp0[1]);
ptsy.push_back(next_wp1[1]);
ptsy.push_back(next_wp2[1]);
```

At this point, we have a series of five global x and y coordinates which describe a path from some given starting position to some ending position, which were given in Frenet s and d coordinates, then transformed to x/y Cartesian coordinates.

At this point, we need to rotate the points to the my-self-driving-car vehicle's local coordinate system. This is done mainly because the technique being used to generate the curve of the path (cubic spline interpolation via spline.h) has problems when the spline runs in a vertical orientation rather than horizontal, and the x-coordinates line up. So we first rotate the points to a horizontal orientation, perform the spline interpolation, then rotate the coordinates back.

So first, we rotate, then create the spline

```
for (int i = 0; i < ptsx.size(); i++) {
  // shift car reference angle to 0 degrees
  double shift_x = ptsx[i] - ref_x;
```

```
  double shift_y = ptsy[i] - ref_y;

  ptsx[i] = (shift_x * cos(0 - ref_yaw) - shift_y * sin(0 - ref_yaw));
  ptsy[i] = (shift_x * sin(0 - ref_yaw) + shift_y * cos(0 - ref_yaw));
}

// create a spline
tk::spline s;

// set (x,y) points to the spline
s.set_points(ptsx, ptsy);
```

From this point, we start to do something curious. We begin to generate our path by first pushing onto our path vector information from the previous path. This path information is given to us from the simulator, and references the coordinates of the previous path that haven't been "consumed" by the my-self-driving-car vehicle as it travels along. You can think of the my-self-driving-car vehicle as being like "pac-man", eating the dots ahead of it (path coordinates), but not all the dots may be eaten before it needs to extend the path

```
for (int i = 0; i < this->previous_path_size; i++) {
  path.push_back(this->previous_path_x[i]);
  path.push_back(this->previous_path_y[i]);
}
```

We then need to calculate the proper spacing of the points so that we travel at our desired velocity. We do this by calculating a distance value based on the spline spacing of 30 meters

```
double target_x = SPLINE_SPACING;  // horizon 30 meters, 0.6s (30 * 0.02)
double target_y = s(target_x);
double target_dist = sqrt((target_x * target_x) + (target_y * target_y));
```

We then fill in the remainder of our path points, with the x-coordinate value being adjusted based on the desired velocity and other values to adjust the spacing between the coordinates. The resulting coordinates are then rotated and transformed back to the global representation, and pushed onto the path vector

```
for (int i = 1; i <= FILLER_DIST - this->previous_path_size; i++) {
  double N = (target_dist / (SECS_PER_TICK * this->car_ref_vel / MPH_TO_MPS));
  double x_point = x_add_on + (target_x) / N;
  double y_point = s(x_point);

  x_add_on = x_point;  // shift to next point

  double x_ref = x_point;
  double y_ref = y_point;
```

```
  // rotate and translate points back to global coordinates
  // to reverse earlier translate/rotate to local car coordinates
  x_point = (x_ref * cos(ref_yaw) - y_ref * sin(ref_yaw));
  y_point = (x_ref * sin(ref_yaw) + y_ref * cos(ref_yaw));

  x_point += ref_x;
  y_point += ref_y;

  path.push_back(x_point);
  path.push_back(y_point);
}
```

4.3. BehaviourPlanner

BehaviourPlanner object is responsible for

a) Creating Road object
b) Populating traffic information in the Road object using the other vehicles' position data provided by the sensor
c) Work with Road object in estimating the next step

Attributes

Road object (1)

Methods

```
Constructor() {

        road = Road()
        road.add(START_LANE,0)
}

advance() {

        this->road.populate_traffic(sensor_fusion);
        this->road.advance(my-self-driving-car_s);


}
```

4.4. Road

Road object is responsible for

a) Representing all vehicles on the road

b) Predicting where the vehicle will be

Methods

populate_traffic() {

```
Clear of all vehicles except self-driving car
repopulate the vehicle list from the sensor fusion data

for (int i = 0; i < sensor_fusion.size(); i++) {
  int oah_id = static_cast<int>(sensor_fusion[i][0]);  // unique id for car

  double oah_x = sensor_fusion[i][1];   // x-position global map coords
  double oah_y = sensor_fusion[i][2];   // y-position global map coords
  double oah_vx = sensor_fusion[i][3];  // x-component of car's velocity
  double oah_vy = sensor_fusion[i][4];  // y-component of car's velocity
  double oah_s = sensor_fusion[i][5];   // how far down the road is the car?
  double oah_d = sensor_fusion[i][6];   // what lane is the car in?

  int oah_lane = getLaneNumber(oah_d);

  // compute the magnitude of the velocity (distance formula)
  double oah_speed = sqrt(oah_vx * oah_vx + oah_vy * oah_vy);

  Vehicle vehicle = Vehicle(oah_lane, oah_s, oah_speed);
  vehicle.state = "KL";

  this->vehicles.insert(std::pair<int, Vehicle>(oah_id, vehicle));
}
```

advance() {

For every timeslice from the simulator, a variety of predictions about the vehicles on the road is generated, provided the vehicle is on the track

```
map<int, vector<vector<double>>> predictions;

map<int, Vehicle>::iterator it = this->vehicles.begin();
while (it != this->vehicles.end()) {
    int v_id = it->first;

    Vehicle vv = it->second;
```

```
            // if the lane is < 0 then vehicle is not on track, so skip
            if (vv.lane > -1) {
              predictions[v_id] = vv.generate_predictions(PREDICTIONS_HZ);
            }

            it++;
        }
```

After predictions, all vehicles are "incremented"; what this means is that each of their positions is updated according to their current data (position, acceleration, etc).

```
it = this->vehicles.begin();
while (it != this->vehicles.end()) {
  int v_id = it->first;

  bool is_my-self-driving-car = (v_id == my-self-driving-car_key);

  ticks++;
  if (ticks > MAX_TICKS) {
    ticks = 0;
    if (is_my-self-driving-car) {
      it->second.s = my-self-driving-car_s;  // sync up my-self-driving-car s-value to
simulator s-value
      it->second.update_state(predictions);
      it->second.realize_state(predictions);
    }
  }

  it->second.increment();

  it++;
}



}
```

```
add_my_self_driving_vechile() {


        used by the constructor in the behavior planner class to add the my-self-
        driving-car vehicle to the list of vehicles on the "road" - aka, the road
        vehicle vector list. There isn't too much to note about this method; it
        basically instantiates a vehicle with the my-self-driving-car vehicle
        information, and inserts it into the vehicle list on the road object

        add_my_self_driving_vechile (int lane_num, double s) {
          Vehicle my-self-driving-car = Vehicle(lane_num, s, 0, 0, true);
```

```
        this->vehicles.insert(std::pair<int, Vehicle>(this->my-self-driving-car_key,
      my-self-driving-car));
        }


        }
```

4.5. Vehicle

Vehicle object is responsible for

a) representing individual vehicles on the road, including both the self-driving vehicle, and other simulated traffic vehicles.

b) representing location (in Frenet coordinates), velocity, acceleration, and "state", which is represents what the vehicle is currently doing or should be doing next

CS - "constant speed", which is only utilized at the instantiation of a vehicle
KL - "keep lane", which means the vehicle needs to stay in its current lane
PLCL & PLCR - "prepare for lane change left/right", which means the car needs to find a gap in the vehicles on its left or right, depending on which lane it intends to change lanes to
LCL & LCR - "lane change left/right", which means the car needs to change lanes to the left or right

The simulated other vehicles are all configured as "constant speed

The state instead is only used by the self-driving vehicle representation, to inform the system to update the trajectory as needed, and ultimately to alter the resulting path for the planner.

Methods

Constructor() {

```
Vehicle::Vehicle(int lane, double s, double v, double a, bool is_my-self-driving-car)
{
    this->lane = lane;
    this->s = s;
```

```
    this->v = v;
    this->a = a;
    this->state = "CS";

    if (is_my-self-driving-car) {
      // configure speed limit, num_lanes, max_acceleration,
      // max_decceleration, and is_my-self-driving-car for my-self-driving-car vehicle
      this->target_speed = MY-SELF-DRIVING-CAR_MAX_VELOCITY;
      this->lanes_available = NUM_LANES;
      this->max_acceleration = MY-SELF-DRIVING-CAR_MAX_ACCEL;
      this->max_decceleration = MY-SELF-DRIVING-CAR_MAX_DECEL;
      this->is_my-self-driving-car = true;
    } else {
      this->max_acceleration = 0;
      this->max_decceleration = 0;
    }

    this->cost = Cost();


When a vehicle is instantiated, information about the vehicle is passed in to the
constructor regarding its current lane, s-value (Frenet coordinate), velocity,
acceleration, and whether the my-self-driving-car vehicle is the one being
instantiated or not.

If the my-self-driving-car vehicle is being instantiated, then a variety of other
class variables are configured using values from the configuration data

In addition, the vehicle object is flagged as being the my-self-driving-car vehicle;
this flag is necessary because other processes in the system need to know when the my-
self-driving-car vehicle is being processed

} // end of construction

update(int lane, double s, double v, double a) {

  this->lane = lane;
  this->s = s;
  this->v = v;
  this->a = a;
}


update_state() {

wrapper for getting the next state of the my-self-driving-car vehicle based on
predictions of   what the other vehicles are doing

this->state = this->get_next_state(predictions);

}
```

get_next_state() {

gets the next state for the self-driving vehicle based on predictions of what the other vehicles are doing

```
  vector<string> states = {"KL", "PLCR", "LCR", "PLCL", "LCL"};

  // remove states that are impossible to get to from the current state
  string state = this->state;

  if (this->lane == 0) {  // left-hand lane
    states.erase(states.begin() + 4);
    states.erase(states.begin() + 3);

    // if in KL state then only allow
    // transition to PLCR, not LCR
    if (state.compare("KL") == 0) states.erase(states.begin() + 2);

  } else if (this->lane == 1) {  // middle lane
    // if in KL state, only allow transition
    //  to PLCR or PLCL, not LCR or LCL
    if (state.compare("KL") == 0) {
      states.erase(states.begin() + 4);
      states.erase(states.begin() + 2);
    }
  } else if (this->lane == this->lanes_available - 1) {  // right-hand lane
    // if in KL state, only allow transition
    // to PLCR, not LCR
    if (state.compare("KL") == 0) states.erase(states.begin() + 4);

    states.erase(states.begin() + 2);
    states.erase(states.begin() + 1);
  }

  vector<string> new_states;
  vector<double> new_costs;

  for (int i = 0; i < states.size(); i++) {
    string state = states[i];

    // two trajectories - the current trajectory, and the proposed state trajectory
    vector<Snapshot> trajectories = this->trajectories_for_state(state, predictions, TRAJECTORIES_HZ);

    double cost = this->cost.calculate_cost(*this, trajectories, predictions);

    new_states.insert(new_states.end(), state);
    new_costs.insert(new_costs.end(), cost);
  }
```

For instance, if it is in the farleft lane, there isn't a reason to allow it to go any further left, so those states are removed from the list of states it can potentially transition to. Furthermore, if the my-self-driving-car vehicle is currently in a "keep

lane" state, it isn't allowed to transition directly to a "lane change" state, but must first be in a "prepare lane change" state instead (this is a safety measure, because in the "lane change" state, no checking is done to see if the lane being changed to is clear of vehicles in order to safely change lanes; the "prepare lane change" state however does check for this).

Once it has a list of states which can be used, for each one of those states it prepares a set of trajectories and calculates the cost for each one of those trajectories, and builds a list of the potential new states and their associated costs

}

min_cost_state() {

determines which state for the my-self-driving-car vehicle is best to transition to, based on the best (lowest) cost score which essentially loops thru the costs and states, and finds whichever one in the list has the lowest cost, and returns the associated state for that lowest cost

}

trajectories_for_state() {

simulates all states available from current state of my-self-driving-car vehicle, and develops trajectories for each potential state

This method takes the potential state, the predictions of what the other vehicles on the road are doing, and a horizon time, and uses those values to project out where the my-self-driving-car vehicle will end up at, based on it's current state, if that proposed state is utilized. It does this by simply simulating what the vehicle will do over the time horizon, given the initial state of the vehicle

```
// remember the current state of ego vehicle
Snapshot current = Snapshot(this->lane, this->s, this->v, this->a, this->state);

// build a list of trajectories
vector<Snapshot> trajectories;

// save the current state for the initial trajectory in the list
trajectories.insert(trajectories.end(), current);

// ...pretend to be in the new proposed state
this->state = state;

// perform the state transition for the proposed state
this->realize_state(predictions);

for (int i = 0; i < horizon; i++) {
  // update the velocity and acceleration of ego out to the horizon
  this->increment(1.0, true);
}

// save the trajectory results of the proposed state
trajectories.insert(trajectories.end(), Snapshot(this->lane, this->s, this->v, this->a, this->state));

}
```

realize_state() {

based on the set my-self-driving-car vehicle state, updates the trajectory motion for the state.

It is basically a wrapper method, which takes the previously generated predictions, and based on the current state of the vehicle it calls one of several different realizer methods, which ultimately carry out the update of the my-self-driving-car vehicle's state

}

realize_constant_speed() {

keeps the my-self-driving-car vehicle at a constant speed by setting the acceleration to zero
}

realize_keep_lane() {

alters the acceleration only of the my-self-driving-car vehicle, based on what vehicles ahead are doing
}

realize_lane_change() {

alters the my-self-driving-car vehicle's current lane to either the left or right; also updates the acceleration for the new lane state

It is passed a directional flag ("L" or "R") for the direction the my-self-driving-car vehicle should change lanes to, which is then used to determine a delta offset from the current lane, which is then updated. A helper function is called to prevent the lane value from going below zero or over the maximum number of lanes, and the acceleration is matched for the new lane

}

max_accel_for_lane() {

alters the my-self-driving-car vehicle's acceleration, and thus speed, based on whether cars are slowing down or speeding up ahead of the my-self-driving-car vehicle

This method operates by first calculating an acceleration for the vehicle based on what the current velocity is and what the target velocity needs to be at each time step this method is called. Eventually, the delta will converge to zero

}

realize_prep_lane_change() {

prepares the my-self-driving-car vehicle for a lane change, by looking for a suitable gap to the left or right of the my-self-driving-car vehicle, depending on which lane is being changed to. This in a manner somewhat similar to what is going on in previous methods, with some differences. The two major differences is that it doesn't change the lanes, but it does check the lane that is going to be changed to by the my-self-driving-car vehicle. It also checks to the immediate side and behind the my-self-driving-car vehicle, and not ahead of it.

}

```
realize_prep_lane_change(map<int, vector<vector<double>>> predictions, string
direction) {
  // prepping for a lane change, so based on which lane is
  // going to be changed to, set a delta offset to represent
  // the new lane
  int delta = 1;

  if (direction.compare("L") == 0) {
    delta = -1;
  }
```

}

generate_predictions() {

generates a list of lane and s-values for a given vehicle over a set time horizon, to allow the my-self-driving-car vehicle a "prediction" of where a "non-player" car will be after the time period (or anywhere along the period), based on the known beginning state

}

increment() {

This method calculates the next s-value and velocity of the vehicle for a given time delta (dt), based on its current velocity and acceleration

}

5. Observation

Observed that My-self-driving-car drives properly without any incidents for up to 50.00/hour (MY-SELF-DRIVING-CAR_MAX_VELOCITY in config.h)

Observed that My-self-driving-car did not exceed Max Acceleration and Jerk

There is no collision for most part and the My-self-driving-car stayed in its lane, except for the time between changing lanes

My-self-driving-car is able to change lanes smoothly

In traffic, My-self-driving-car vehicle will typically follow the flow of the traffic, maintaining a healthy distance from the lead vehicle to prevent collisions. If the traffic slows down in front of the vehicle, it will slow down as well, then ramp its speed up again after the situation passes.

The my-self-driving-car car in this implementation is not a perfect. I struggled for days trying to get the system to be more aggressive in lane changes and passing, and what I generally got for my troubles was a car that ran others off the road. So after lengthy experimentation and parameter tweaking, I settled on what you see in the project submission.

In my testing, I have found that when the rare collision occurs in following situations

a) The My-self-driving-car vehicle changes lanes, but isn't moving fast enough as

another vehicle moving much faster in the lane being changed to approaches. In the event the lane change isn't completed in time, a collision can occur if the other car doesn't brake in time

b) More insidious is the fact that there are cars in the simulation that seem to act like "reckless drivers" at times. These cars will sometimes swerve in front of other cars or the my-self-driving-car vehicle, so close as to not allow the my-self-driving-car vehicle time to avoid a collision.

I have also seen these "reckless driver" vehicles cause collisions ahead of the my-self-driving-car vehicle, and I have seen vehicles from the opposite side veer into the right-hand lanes as well.

It doesn't appear that when this occurs that these vehicles are in the sensor data - at least, my car and other cars don't appear to acknowledge they are anywhere in front of them. I don't know for certain if this is absolutely true, as these incidents are very rare occurrences, and are difficult to test for.

There always seems to be a maximum of 12 cars at any one time on the patch, but some of those cars aren't visible, and are assigned a lane number of -1. You can see this patch effect if you zoom out from the my-self-driving-car vehicle and flatten the view out toward the horizon of the road; you will then see the cars in the distance pop in and out of the view as the fall off the edges (and get recycled).

In short, I believe, My-self-driving-car is able to drive without any major incidents for most part.

6. References

- Path planning project though not a big project in terms of number of lines of code, it is difficult to implement all the code from scratch.
- Udacity provided basic framework and code snippets through lessons but that was not sufficient enough for me to take up and implement rest of the code needed for completion.
- I relied on other Engineer's suggestions from Udacity forums and Slack. I am thankful to all the people who shared code snippets and suggestions.
- Though I have reused other people's suggestions, I believe the code structure and code optimization that I have implemented is unique to me
- Finally, I believe, I have attempted to describe the components and interactions between components as much possible