Path Planning Project

Jean-François Vanreusel

Overview

In this project, we implement a **Path Planning** model to guide our car on a track. The goal of this project is to complete at least one loop on the track without causing any violation (speed limit, lane violation, collision) and by keeping the ride comfortable with smooth accelerations and decelerations.

In this project, we implement a behavior planner (i.e. decision model on what to do) and a path planner to execute our decision smoothly. We also need to develop a prediction model (using sensor fusion and localization data) to predict where other cars will be at the time our car makes its move.

Deliverables

Our project is composed of the following files:

- main.cpp
- path.cpp and path.h
- vehicle.cpp and vehicle.h
- highway_map.csv
- **Path Planning Project.pdf** – You're reading it ☺.

Path.cpp (and .h) include the Path class and relevant methods.
Vehicle.cpp (and .h) include the Vehicle class and relevant methods.

Map and trajectory

To ensure that our vehicle remains on the track, we received a map composed of waypoints data about the track. This data is included in the data/highway_map.csv file. We used this data to convert global coordinates to Frenet coordinates (s,d) and vice-versa.

Because we were only provided 181 waypoints, we decided to generate additional waypoints using a spline. This is included in the Path::define_spline method. We generated a new point every 50 centimeters.

The spline only needs the be created and run at the beginning of our application. This saves processing time.

Finally, we observed some issues when we were completing the first lap and discovered that our spline didn't "close the loop" on the track. To remedy to this problem, we added the following point to the highway data:

784.6001 1135.571 6945 -0.02359831 -0.9997216

6945 represents the maximum length of the track, which means that it's also at the same position as the first point (s=0). This additional waypoint could have been added programmatically too.

Path Planning Model

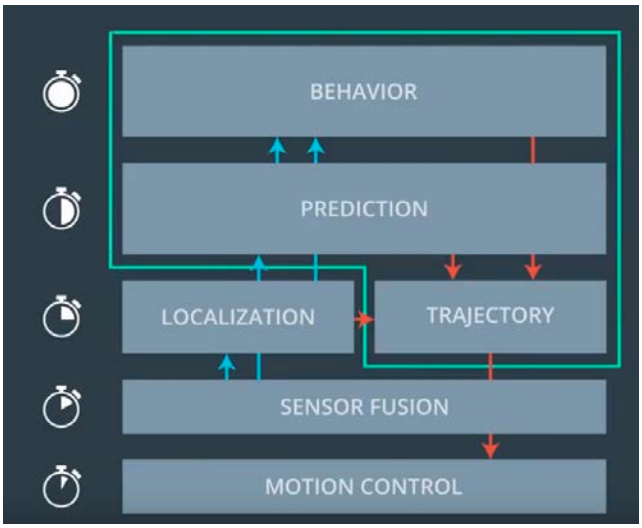


Figure 1: Path Planning architecture

Our path planning architecture includes various layers:

1. **Behavior planner:** This component decides on the best next move for the car. This is included in the Vehicle class.
2. **Prediction model:** This component predicts where the cars will be in the future and is critical to avoid collisions. It leverages our car localization data as well as the sensor fusion data. This is included in the main.cpp file.
3. **The Path planner:** This component creates the optimal path based on the instruction it gets from the behavior planner.
4. The **Motion Control** component is handled by the simulator after we submit our path.

Behavior Planner

The purpose of the behavior planner is to decide the next best move for the car. To support this model, we developed a small state machine model (in the vehicle.cpp file).

Our state machine is composed of 4 states: Keep Lane (KL), Keep Lane but Slow Down (KLSD), Lane Change Left (LCL) and Lane Change Right (LCR).

When we are in “KL”, we can stay in “KL” state or go to “KLSD”. We can also change lane towards the left (“LCL”) if we are not on the first lane or towards the right (“LCR”) if we are not on the right most lane.

```
void Vehicle::state_machine(vector<string>& possible_states) {

    //returns the possible states based on the current state

    //KL: Keep Lane at max speed (50 mph for this project)
    //KLSD: Keep Lane but Slow Down (to stay away from car in front)
    //LCR: Lane Change Right
    //LCL: Lane Change Left

    if ((this->state.compare("KL") == 0) || (this->state.compare("KLSD") ==
0)){
        possible_states = {"KL", "KLSD"};
        if (lane == target_lane) {
            if (target_lane !=0) {possible_states.push_back("LCL");}
            if (target_lane != lanes_available - 1)
{possible_states.push_back("LCR");}
        }
    } else if (this->state.compare("LCL") == 0) {
        possible_states = {"KL"};    //we could possibly add LCR and KLSD
    } else if (this->state.compare("LCR") == 0) {
        possible_states = {"KL"};
    } else possible_states = {"KL"};
```

Similarly, if we are in the “KLSD” state, we can reach the “KL”, “KLSD”, “LCL” or “LCR” states. We set stricter constraints when we are in the “LCR” or “LCL” states. Those states can only be followed by “KL” state to be safe. In other words, we can’t change two lanes immediately – or quickly change lane towards the right and then towards the left.

To determine the optimal state, we simulate the state of the car after we make our move. To do so, we create a copy of our vehicle (ego) in the vehicle::update_state method.

We developed three key functions that realize the “moves”:

- **Realize_keep_lane**
- **Realize_keep_lane_slow_down**
- **Realize_lane_change**

Each of those functions uses the past information on the car and predicts the state of the car in the future based on its predicted position, and velocity.

Note that we use **Frenet** coordinates (s,d) to apply to state changes as they are independent of the road curvatures.

As part of the Behavior Planner, we simulate each possible state and pick the best one possible. To select the best possible state, we developed 3 cost functions:

- **Vehicle::keep_lane_cost_function –** This function returns the lowest value possible (0) unless our car starts to be too close from a car ahead of us. In that case, we set the cost to 500 to encourage the behavior planner to pick another option – such as slowing down or changing lane.
- **Vehicle::change_lane_cost_function –** This function measures the feasibility of a lane change. We only want to change lane if our current lane doesn't allow us to reach our optimal speed. If we have room to execute a lane change, our cost is low. Otherwise, it is set to a high value because it would cause a collision.
- **Vehicle::slow_down_cost_function –** If we can't keep our current speed in our lane and can't change lane, then we have no other choice than slowing down.

Our algorithm picks the state that provides the lowest cost function.

The output of our Behavioral Planner is a target speed and lane for our path planner.

Path Planner

The path planner code is included in the path.cpp file. It is composed of three key methods:

Path Init method

This init method takes parameters provided by the behavioral planner such as the target speed and lanes. It also takes other key data provided by the simulator: **previous_path_x, previous_path_y.** Previous_path_x and Previous_path_y include points (from the previous path) that haven't been processed by the simulator yet. Those points are important to generate the new path. We want to "mesh" the end of the previous path with the new path, so we don't create sudden velocity or acceleration changes (jerk).

Path Create method

The create method creates the new path based on the suggested velocity and lane. The method starts by creating a **spline** with the latest information known on the car. If previous_path_x and previous_path_y include enough points, we take the last two points of previous_path as the starting point of our spline. We then add 3 points at further distance to define the general "curve" of our desired trajectory. We created two types of trajectories. One when the car stays on the same lane and another one when the car switches lane. This avoided acceleration/jerk issues when the car switched lanes. We use 3 "future" points to define our trajectories/splines as seen below:

```
if (trunc(ref_d/4) != trunc(current_d/4)) { // if lane change
    trajectory_points = {60,75,90};
} else {
    trajectory_points = {30,60,90};
}
```

```
for(int i = 0; i < 3; i++) {
  next_s = ref_s + trajectory_points[i]; // we pick a point 30 meters away
  next_d = ref_d; //we go to ref_d
  xy_coord = convert_sd_to_xy(next_s, next_d);
  path_x.push_back(xy_coord[0]);
  path_y.push_back(xy_coord[1]);
}
```

Once we have our spline, we can apply it to generate the new trajectory towards the expected destination and match our speed criteria.

Note a key function (highlighted in the code) used to generate the path is the conversion function from the S,D space into the X,Y space. This ensures that our car stays on the track.

Path Generation

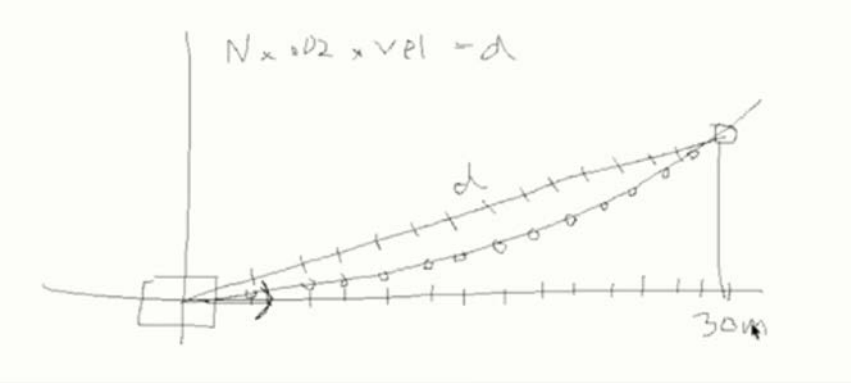
The path is first generated by using the previous_path points that haven't been processed by the simulator yet.

```
for (int i = 0; i < path_size; i++) {
        next_x_vals.push_back(previous_path_x[i]);
        next_y_vals.push_back(previous_path_y[i]);
    }
```

Then, we start adding points to our trajectory by using the spline that we created above. This was the trickiest part of the project as far as I am concerned.

```
    double target_x = 30;   //we create a path from our current
position x to 30 meters further.
    double target_y = x2y(target_x);
    double target_distance = sqrt((target_x)*(target_x) -
(target_y)*(target_y));
```

First, we decide to work with the car coordinate system. So, the car is at the origin. We then pick a point at x=30 and compute its y equivalent by using the spline that we defined above. We can compute the distance using Pythagoras theorem.



Now that we have the distance and know that we want to maintain a target speed, we can generate points on our trajectory. The simulator processes points every 20 milliseconds (0.02 sec) – so we need to determine how many points are needed to reach our target speed.

```
double x_add_on = 0;

  //we generate points for our trajectory such that it includes 50 points
  for (int i=0; i < 50 - path_size; i++) {
    //compute the number of N's needed to
    double N = (target_distance/(time_interval*target_speed/2.24)); //divided by 2.24 because of mph to m/s speed conversion.
    double x_point = x_add_on + (target_x)/N;
    double y_point = x2y(x_point);

    x_add_on = x_point;
    double x_ref = x_point;
    double y_ref = y_point;

    //convert point from car coordinate system back to global coord system
    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;

    next_x_vals.push_back(x_point);
    next_y_vals.push_back(y_point);
  }
```

Predictions

Another key aspect of our application is its ability to predict where other cars will be in the future. This is critical information to master if we don't want to collide with other cars on the road.

We created the predict function (in main.cpp) which takes in (as input) the sensor fusion data and the time delay in which we want to predict the future. The predict function generate a predictions hash/map with the state of each car in the future (time now + time_delay).

Results

We successfully completed multiple laps on the track without violations. Here is a screenshot where the car drove on the track for 10 miles in 13 minutes.

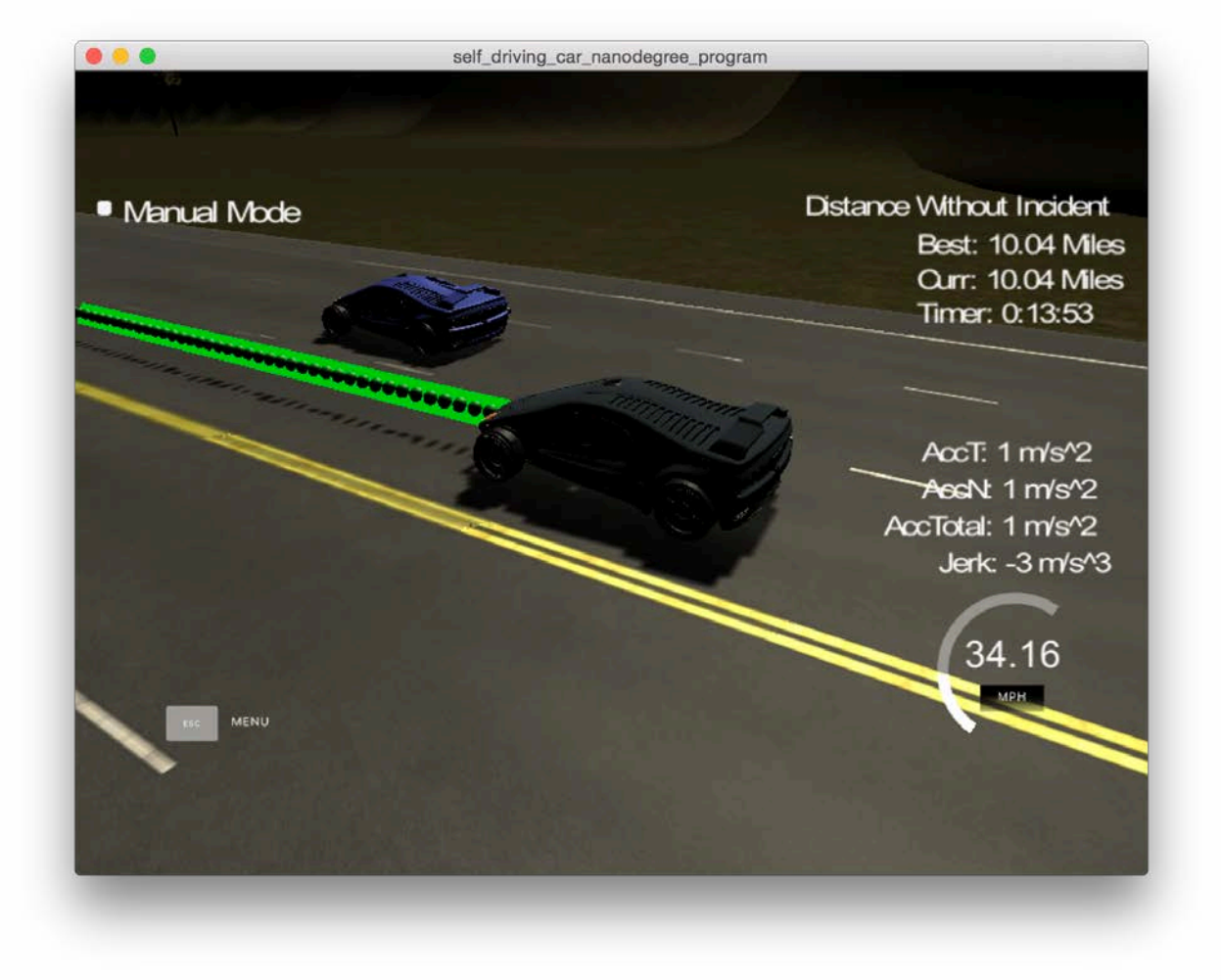


Figure 2: Car driving around the track for 10 miles

Known issues & Areas for improvement

A few areas can be improved in our implementation:

- If the car drives on the right lane towards the middle of the track, it sometimes gets a lane violation. We couldn't observe it visually. The position (defined by d) is still well below 12, but maybe the back of the car crosses the lane. So, I "hacked" the "d" coordinate when the car drives on the right lane: ref_d = (2 + 4 * ego.target_lane)*0.98;
- Our changing lane algorithm ensures that no car is on our target lane. However, the speed decreases a bit in some situations which increases the risk to be hit by a faster car on that lane.