Path Planning Project

Udacity Self Driving Car Nano Degree, Term 3, Project 1

Project

The goal of this project is to design a Path Planner in C++ and drive a car on a 3-lane highway with traffic, in the simulator environment. The Path Planner and the simulator communicate with each other using WebSocket. The simulator provides the car telemetry and Sensor Fusion data to the Path Planner. The Path Planner analyzes the Sensor Fusion data to determine the position of other cars surrounding the ego car to determine if it is necessary and safe to change lanes. The Path Planner also adjusts the speed of the ego car based on the ego car's current speed and the position and speed of other cars. The Path Planner provides the lane and velocity information to the simulator and the simulator applies these parameters to the ego car. In addition, the Path Planner provides a set of path points to the simulator to keep driving the ego car.

Project Steps

- Make the ego car to move and stay in its current, center lane.
- Make the ego car to stay in the center lane and adjust its speed to avoid colliding with the car ahead of it.
- Analyze the complete Sensor Fusion data to determine where the ego car and other cars would be in the future.
- If it is safer, make the ego car change lanes to move around a slow moving car.
- If it is safer, make the ego car to come back to the center lane.

Code Organization

Since I will be refereeing to code snippets, I am providing brief overview of the code structure.

- main.cpp
  Started with the repo version and enhanced it to meet the Rubric requirements.

- behaviorPlanner.h & behaviorPlanner.cpp
  The BehaviorPlanner class is defined in these files. The BehaviorPlanner class is algorithmic in nature. It doesn't have any data members and so, doesn't remember any data. It is responsible for analyzing the Sensor Fusion data and determining if there are cars within unsafe distance ahead and on the left and right lanes.

Valid Trajectories

- **The car is able to drive at least 4.32 miles without incident.**
  I was able to successfully test multiple times with the car driving incident free for more than 4.32 miles.

- **The car drives according to the speed limit.**
  The "Getting Started" section of the Path Planning project specifies a speed limit of 50MPH. The following code snippet defined the maximum speed as 49.5 MPH and makes sure that speed is not exceeded.

  File: behaviorPlanner.h,
  line #: 15
  ```
  const double MAX_SPEED = 49.5;
  ```

  File: behaviorPlanner.cpp, Function: BehaviorPlanner::determineLaneAndVelocity(),
  line#: 112-113
  ```
  if( ref_velocity < MAX_SPEED ) {
    ref_velocity += ACCEL_STEP;
  }
  ```

- **Max acceleration and jerk are not exceeded.**
  The "Getting Started" section of the Path Planning project specifies the limit of both the acceleration and jerk as 10 m/s^2. The following code snippet defines the acceleration step as 0.224 which is equivalent to 5 meters per second square.

  File: behaviorPlanner.h,
  line #: 16
  ```
  const double ACCEL_STEP = 0.224;
  ```

  File: behaviorPlanner.cpp, Function: BehaviorPlanner::determineLaneAndVelocity(), |
  line#: 112-113
  ```
  if( ref_velocity < MAX_SPEED ) {
    ref_velocity += ACCEL_STEP;
  }
  ```

  When increasing or decreasing the acceleration, the constant ACCEL_STEP is used.
  This makes sure that the maximum acceleration is not exceeded and jerk is minimized.

- **Car does not have collisions.**
  The flags, other_cars.ahead, other_cars.left and other_cars.right are set to "true" if there are cars within 30 meters . This is done in the following code snippet.
  File: behaviorPlanner.cpp, Function: BehaviorPlanner::areCarsPresentInLanes()
  Line #: 40 – 52

```
if( other_lane == ego_lane ) {
  other_cars.ahead |= other_s > ego_s &&  other_s - ego_s <
                SAFE_DISTANCE;
} else if( other_lane - ego_lane == -1 ) {
  other_cars.left |= ego_s - SAFE_DISTANCE < other_s &&  ego_s +
                 SAFE_DISTANCE > other_s;
} else if( other_lane - ego_lane == 1 ) {
  other_cars.right |= ego_s - SAFE_DISTANCE < other_s && ego_s +
                SAFE_DISTANCE > other_s;
}
```

  In the following code snippet, lane change is executed only when there are no cars in adjacent lanes within a safety distance. If not able to execute a lane change, then the speed is reduced. This is how the Path Planner avoids collisions.
  File: behaviorPlanner.cpp, Function: BehaviorPlanner::determineLaneAndVelocity()
  Line #: 73 - 97

```
if( other_cars.ahead ) {

  if( !other_cars.left && lane > 0 ) {

    // There is a car ahead. There is a left lane and
    // there is no car in the left lane.
    // Move to the left lane
    lane--;

  } else if( !other_cars.right && lane != 2 ) {
    // There is a car ahead. There is a right lane and
    // there is no car in the right lane
    // Move to the right lane.
    lane++;

  } else {
    // There is a car ahead. There are cars in both the
    // left lane and the right lane. Nowhere to go.
    // Reduce the speed.
    ref_velocity -= ACCEL_STEP; //5m/s/s(meter per second square)

  }
} /* if( other_cars.ahead ) */
```

- **The car stays in its lane, except for the time between changing lanes.**

  When there is no car ahead of the ego car, there is no need to change lane. Under this condition, the ego car's lane number is checked. If it is not in the center lane (lane = 1), then the ego car is safely moved to the center lane. The following code snippet captures this behavior.
  File: behaviorPlanner.cpp, Function: BehaviorPlanner::determineLaneAndVelocity()
  Line #: 104 - 109

```
if( lane != 1 ) {
  if( ( lane == 0 && !other_cars.right ) ||
      (lane == 2 &&!other_cars.left ) ) {
    // Safe to move to the center lane.
    lane = 1;
  }
}
```

- **The car is able to change lanes.**
  The car is successfully able to change lanes as needed based on Sensor Fusion data.
  Please refer previous bullet sections for detailed description and code snippets.

Reflection

As a starting point, I fully watched the code walk through video. The walk through covered a lot of project requirements with plenty of details and how to implement them in code. I mirrored my Path Planner design based on how it was done in the code walk through. I enhanced main.cpp with additional functionality as required by the project rubric.

To decide whether it is necessary to change lanes (i.e., slow car ahead of the ego car), as well as to determine if it is safer to do a lane change, the sensor fusion data is analyzed. Based on this prediction, action is taken to change lanes (i.e., safe to change lanes) or stay in the same lane (cars in adjacent lanes) and reduce the speed. This implementation is covered in detail in the previous section, Valid Trajectories.

The code walk through video did not cover all aspects of lane changing bit hinted how it could be done. I debated between using the Finite State Machines approach and using a very simple approach hinted by the code walk through. Finally, I decided to keep things simple and developed on the ideas discussed in the code walk through. This worked well.

To generate paths, the Path Planner uses the past path points, car coordinates, and the destination lane as determined by the BehaviorPlanner (please refer previous section, Valid Trajectories).

The last two points from the previous path and 3 distant points at 30m, 60m, and 90m are used to initialize the spline calculation (main.cpp, lines 304 – 356). To keep the calculations simple, the coordinates are transformed to the local car coordinates using shift and rotation (main.cpp, lines 362 – 370). To make sure there is no jerk and to maintain a smooth continuity, the past trajectory points are copied as part of the new trajectory (main.cpp, lines 383 – 387). Finally, the path points calculated using spline are converted to the global coordinates (main.cpp, lines 414 – 415) before being sent to the simulator.