Highway Path Planning Project - Reflection

Udacity Self Driving Car Nanodegree

Goals

In main goal of this project is to safely navigate around a virtual highway with other traffic and handling the below mentioned points.

- Stay within your lane unless Lane change is required
- Change lane when a slow-moving vehicle is going ahead and the nearby lane is free
- No collisions should occur, during lane changing or lane keeping.

Constraints:

1. 50 MPH **speed** limit
2. **Acceleration** (change in velocity less than +- 10 m/s) less than 10 m^2/s
3. Jerk (change in acceleration less than +-10 m^2/s) less than 10 m^3/s
4. Total distance travelled without any instance should be more than **4.32 Miles**

Inputs:

- Car's localization data
- sensor fusion data
- sparse map list of waypoints around the highway.

File Structure:

Major files that are frequently used in this project are

- **main.cpp** – Handles all the tasks like path planning, navigation and sending messages to GUI.
- **main.h** – holds structures and user defined helper functions to support functions in main.cpp.
- **helpers.h** – Provided header files, consisting of all basic supporting functions.
- **spline.h** – predefined library file to perform interpolation from a graph. Alternative for polynomial fitting.

helpers.h Fucntions:

- **hasData** – Provides all necessary details about the vehicles present in the map and the map details
- **getFrenet** – Transform points in cartesian coordinates to Frenet coordinates.
- **getXY** – Transform points in Frenet coordinates to cartesian coordinates.

Main.h:

- struct **AllRoadVehicleDetails** – All information related to the neighbouring vehicles obtained from the map and sensor fusion.
- struct **Vehicle** - Placeholder to hold all necessary vehicle data.
- **sigmoid** () – Mathematical Sigmoid function, converts range between 0 to 1

Implementation Details:

Newly added functions in main.cpp and their descriptions are described here. The same was added to the implementation in the main.cpp file for better understanding.

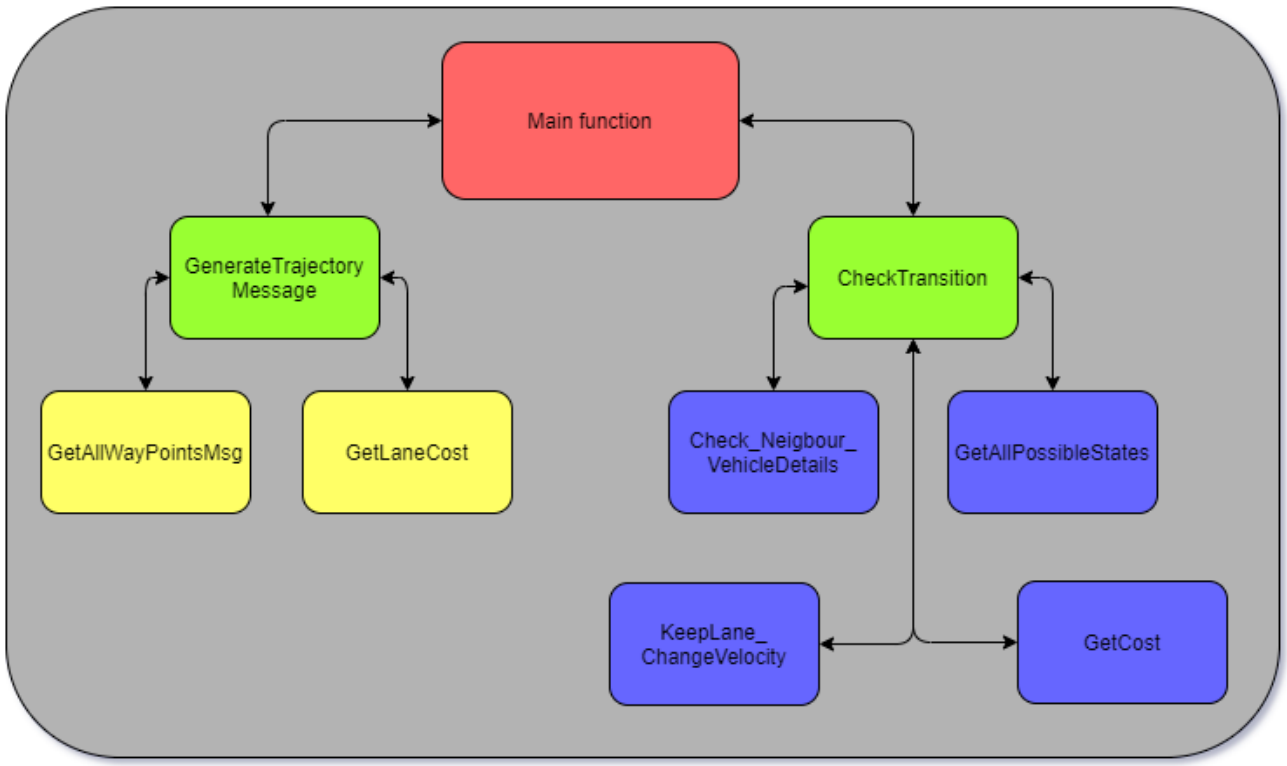

Fig.1 High Level Architecture Diagram of main.cpp

CheckTransition:

Handles various state transition of ego vehicle. Get optimal Lane (d value) and Velocity of ego vehicle from all the available details.

1. if there is a vehicle ahead at a distance within 30 m, try lane change or reduce velocity to prevent collision.
2. Choose lane to which ego vehicle has to move, based on the cost function based on traffic.
3. If unable to change lane due to vehicle ahead and behind the other lanes,
   a. try to reduce speed constantly if distance to collide is greater than 25m
   b. reduce speed based on the distance to collide dynamically using spline library
4. If no vehicle ahead maintains target speed.

return: optimal velocity and lane to be travelled

GenerateTrajectoryMessage:

1. Generate optimal trajectory based on lane cost.

2. Generate 3 trajectories with +- 0.2 to lane values.
3. Choose the optimal trajectory based on deviation between the expected and actual lane position in frenet coordinates.
4. **return**: Json message with optimal way points.

Check_Neigbour_VehicleDetails:

Update information of the neighbouring vehicles in the map based on the information from the sensor fusion.
return : structure with all details listed in AllRoadVehicleDetails

GetAllPossibleStates:

Get all possible lane transitions from current state.
return : Possible lane state

GetCost:

Calculate the cost of parameters based on the distance to the nearest vehicle in each lane

1. if 2 lanes are available during changing lanes (ex. when travelling in middle lane)
2. Choose the lane in which the ongoing vehicle is far away from ego vehicle. (less traffic)
3. This helps in reducing successive lane changes
4. High cost if vehicle is too close, low cost if vehicle is far away.
5. Cost is normalized to 1.0
6. calculate cost only if lane change in that lane is possible; no vehicles within +-40m from ego vehicle
7. Maximum cost if lane change not possible

return : cost based on nearest vehicle distance

KeepLane_ChangeVelocity:

calculate the velocity of ego vehicle in Keep Lane state.

1. deceleration is based on the distance between the vehicle going ahead of ego vehicle in same direction.
2. Construct a graph between distance to collide and factor of change in velocity (deceleration).
3. Interpolate at get the values of rate of change in velocity factor based from the graph.
4. Steps 2 and 3 is executed using spline.h library.

return : Updated Reference velocity

Implementation Flow-Chart:

The complete algorithm of the implementation is shown in the below flow chart in Fig.2

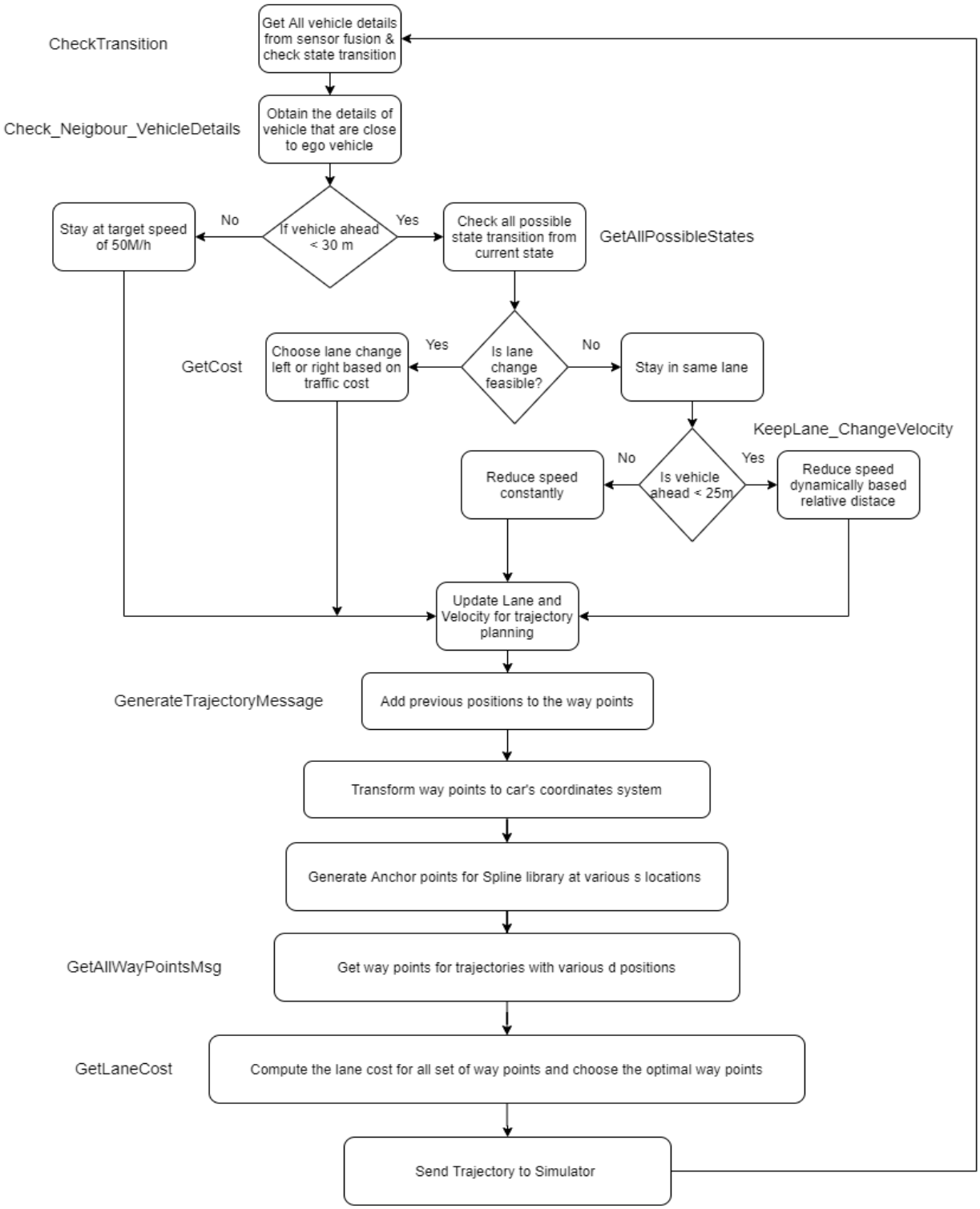

Fig.2 Implementation Algorithm flow chart

Results:

- The vehicle is able to achieve all the basic goals.
- It travels around the entire highway distance of 4.32miles without any incident
- The vehicle is capable of changing lanes (both left and right lanes) whenever needed
- The car does not experience acceleration or jerk above the specified level.
- No collisions with other vehicles

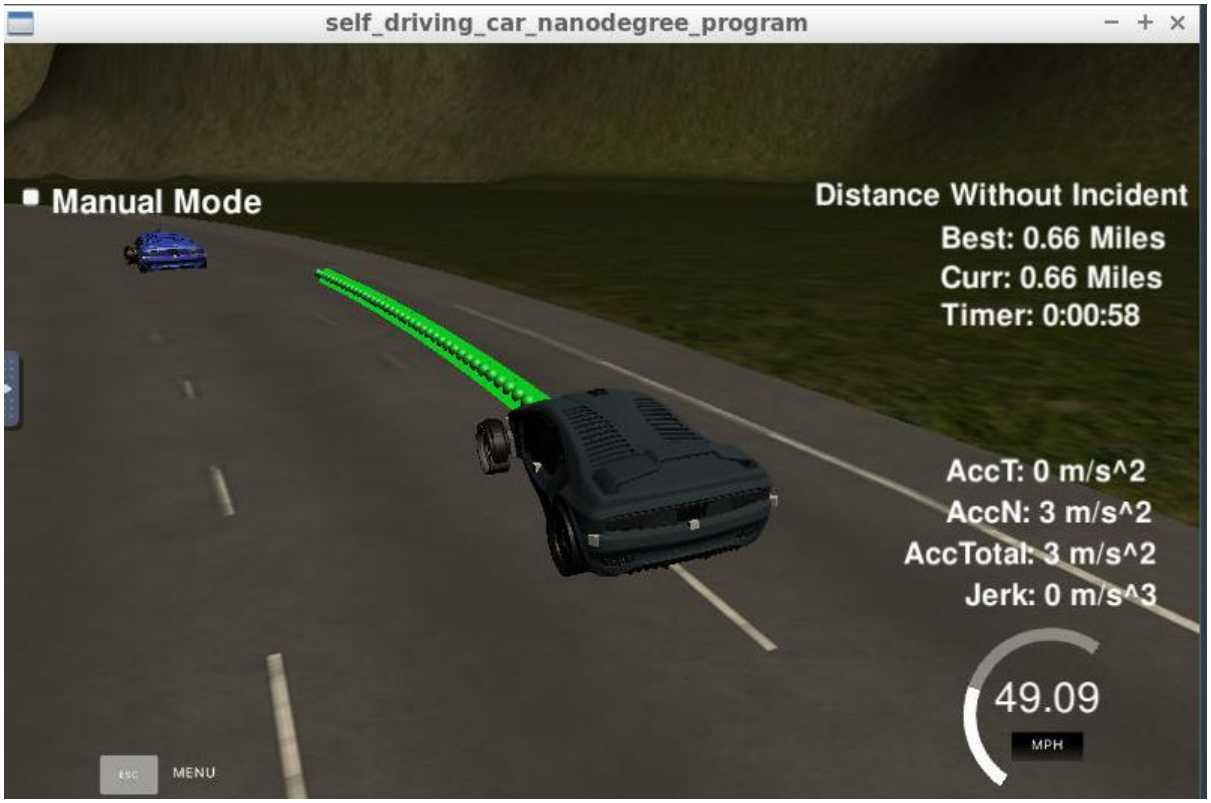

Fig.3 Lane change Right

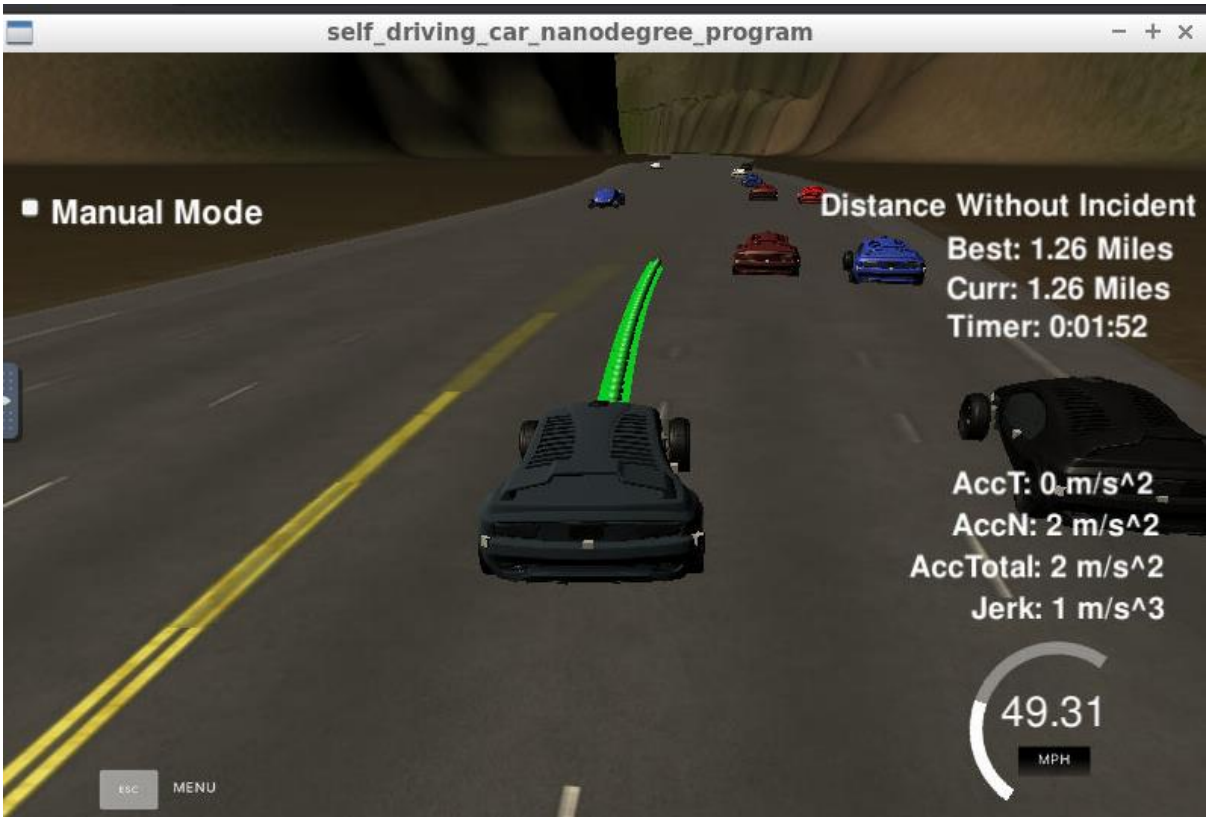

Fig.4 Lane change Left

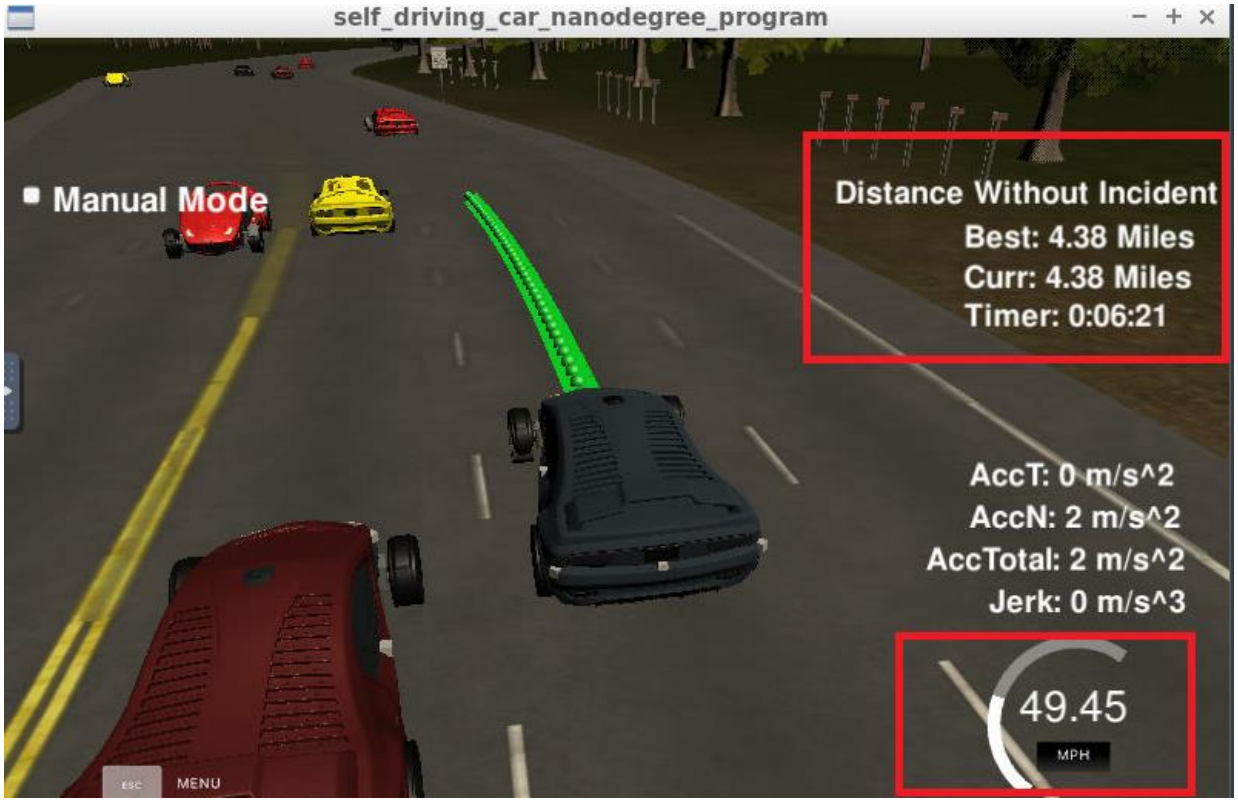

Fig.5 Total Distance Travelled and Target Velocity

Future Woks:

- Instead of developing a cost-based algorithm, Behavioral Cloning based path planning same as project 4 can be implemented. This captures the decisions taken by a human driver and tries to behave in the same way.
- More cost functions and new state transitions shall be added to improve the performance the path planner.
- Consideration of unacceptable behaviour from other vehicles shall be handled in the algorithm.
- PID controller-based trajectory planning shall be implemented in order to avoid sudden oscillations and to reduce steady state error.