Motion Planning and Decision Making for Autonomous Vehicles

REVIEW | HISTORY

Meets Specifications

Congratulations, you have passed the project successfully by meeting all the specifications very well. 🎉🎉

My solution runs great, but for whatever reason there seems to be a ghost vehicle matching exactly the ego which the ego-vehicle "glides" through — this seems like a saved state? Are you aware of what might be causing this?

This could be because the simulator driver persists until explicitly killed and older vehicle still shows when a run is re-initiated.

As far as the code review goes, I would specifically like advice on improving the formulas for distance, ideas for the motion controller, and real suggestions regarding parameter values and other TODOs.

Great work here. The formula for distance calculation is looking good and goal calculations are perfect too.
All the required files have been submitted, great work in understanding on how to create spirals paths for a vehicle and successfully implement the functions in project. The program can be compiled and run without any error. 👏🏻

```
[ 54%] Linking CXX executable spiral_planner
[100%] Built target spiral_planner
(venv) root@1b6aa8878c65:/home/workspace/nd013-c5-planning-starter/project/starter_files#
```

Good job in implementing required functionalities in behavior planner, cost functions, motion planner and vehicle profile generator cpp files very nicely, and also working on all TODOs that are required for this project very well! The choice of planning parameters has also been wisely done and you could create an optimal path given the mathematical code for generating the cubic spirals. 🤞🏻

Reference

- Decision-Making for Automated Vehicles Using a Hierarchical Behavior-Based Arbitration Scheme - This article is talking about how behavior based automated vehicles are built.
- Motion Planning Among Dynamic, Decision-Making Agents with Deep Reinforcement Learning - A very nice read about agents in deep learning for decision making.
- Planning and Decision Making for Aerial Robots - This book is mainly about aerial robots and decision making.

Implementation of Decision Making Framework

✓ Completed this TODO in behavior_planner_FSM.cpp:

TODO - Lookahead: One way to find a reasonable lookahead distance is to find the distance you will need to come to a stop while traveling at speed V and using a comfortable deceleration.

Very nicely done!

Your formula for lookahead distance works great!

Lookahead: One way to find a reasonable lookahead distance is to find the distance you will need to come to a stop while traveling at speed V and using a comfortable deceleration. ✅

Reference

There is a research going on using particle filtering for motion and decision making in vehicles, please find an interesting article around it here.

✓ Completed the following TODOs in behavior_planner_FSM.cpp:

1. TODO - Goal behind the stopping point: put the goal behind the stopping point (i.e the actual goal location) by "_stop_line_buffer". HINTS: remember that we need to go back in the opposite direction of the goal/road, i.e you should use: ang = goal.rotation.yaw + M_PI and then use cosine and sine to get x and y.
2. TODO - Goal speed at stopping point: What should be the goal speed?
3. TODO - Goal speed in nominal state: What should be the goal speed now that we know we are in Nominal state and we can continue freely?
   Remember that the speed is a vector.
4. TODO - Maintain the same goal when in DECEL_TO_STOP state: Make sure the new goal is the same as the previous goal. That way we keep/maintain the goal at the stop line.
5. TODO - It turns out that when we teleport, the car is always at speed zero. In this the case, as soon as we enter the DECEL_TO_STOP state,
   the condition that we are <=_stop_threshold_speed is ALWAYS true and we move straight to "STOPPED" state. To solve this issue (since we don't have a motion controller yet), you should use "distance" instead of speed. Make sure the distance to the stopping point is <=P_STOP_THRESHOLD_DISTANCE. Uncomment the line used to calculate the distance
6. TODO - Use distance rather than speed: Use distance rather than speed...
7. TODO - Move to STOPPED state: Now that we know we are close or at the stopping point we should change state to "STOPPED"
8. TODO - Maintain the same goal when in STOPPED state: Make sure the new goal is the same as the previous goal. That way we keep/maintain the goal at the stop line.
9. TODO - Move to FOLLOW_LANE state: What state do we want to move to, when we are "done" at the STOPPED state?

✅ auto ang = goal.rotation.yaw + M_PI;
✅ goal.location.x += _stop_line_buffer std::cos(ang);
✅ goal.location.y += _stop_line_buffer std::sin(ang);
✅ goal.velocity.x = 0;
✅ goal.velocity.y = 0;
✅ goal.velocity.z = 0;
✅ goal.velocity.x = _speed_limit std::cos(goal.rotation.yaw);
✅ goal.velocity.y = _speed_limit std::sin(goal.rotation.yaw);
✅ goal.velocity.z = 0;
✅ goal = _goal;
✅ auto distance_to_stop_sign = utils::magnitude(goal.location - ego_state.location);
✅ if (distance_to_stop_sign <= P_STOP_THRESHOLD_DISTANCE) {
✅ _active_maneuver = STOPPED;
✅ goal = _goal;
✅ _active_maneuver = FOLLOW_LANE;

Reference

Some very interesting topics are being worked upon on different vehicle types and behavior planning being one of them, you may read one good paper here about it.

Path Planning using Cubic Spirals

✓ Completed this TODO in planning_params.h:

TODO - Num of paths (goals): Chose a value for "P_NUM_PATHS"

Good choice: `5`, your chosen value is working good in simulator !👍

Reference

You may enjoy going through this read about using two cubic spirals for path planning.

✓ Completed the following TODOs in motion_planner.cpp:

1. TODO - Perpendicular direction: ADD pi/2 to the goal yaw (goal_state.rotation.yaw)
2. TODO - Offset goal location: calculate the x and y position of the offset goals using "offset" (calculated above) and knowing that the goals should lie on a perpendicular line to the direction (yaw) of the main goal. You calculated this direction above (yaw_plus_90). HINT: use std::cos(yaw_plus_90) and std::sin(yaw_plus_90)

- Perpendicular direction: ADD pi/2 to the goal yaw (goal_state.rotation.yaw) ✅
- Offset goal location ✅
  Great job again!

✓ Returned a discretized spiral that allows for a successful collision checking.

Completed this TODO in planning_params.h:

TODO-num of points in the spiral: choose a value for "P_NUM_POINTS_IN_SPIRAL"

_Reminder: Deciding the number of points used to discretize will have a huge impact on checking for collisions and visualization.

Too many points and we'll pay a high computation penalty and the visualization will be terribly slow.
Too few and we might miss detecting collisions and we will teleport large distances making it visually not natural._

With the calculations being used, the current set value of spiral points 25 works well! ✅

Reference
I am sharing one more reference research paper here for your reading purpose and it talks about dynamic driving in autonomous vehicles. Hope you like it!

Trajectory Generation

✓ Completed this TODO in velocity_profile_generator.cpp

TODO - Calc distance: Used one of the common rectilinear accelerated equations of motion to calculate the distance traveled while going from
v_i (initial velocity) to v_f (final velocity) at a constant acceleration/deceleration "a". HINT look at the description of this function

Great work here! Calc distance ✅

Reference
A good article about motion planning with trajectory optimization can be read here.

✓ Completed this TODO in velocity_profile_generator.cpp

TODO - Calc final speed: Calculate the final distance.

Calc final speed ✅

Trajectory Selection

✓ Completed this TODO in cost_functions.cpp

TODO - Circle placement: Where should the circles be at? The code below is NOT complete. HINT: use CIRCLE_OFFSETS[c], sine and cosine to
calculate x and y

Calc Circle placement✅

✓ Completed this TODO in cost_functions.cpp

TODO - Distance from circles to obstacles/actor: How do you calculate the distance between the center of each circle and the obstacle/actor

Distance from circles to obstacles/actor✅

The calculation of the distance between center of each cirlce and the obstacle/actor is correctly done.

✓ Completed this TODO in cost_functions.cpp

TODO - Distance between last point on spiral and main goal: How do we calculate the distance between the last point on the spiral (spiral[n-1])
and the main goal (main_goal.location). Use spiral[n - 1].x, spiral[n - 1].y and spiral[n - 1].z Use main_goal.location.x, main_goal.location.y and main_goal.location.z

Distance between last point on spiral and main goal✅

You evaluated each trajectory generated against an objective function and selected the "best" one. The goal of this objective function is to score poor paths that are in collision OR too close to static obstacles and to score high paths that end up closer to the center-line of the global path.

DOWNLOAD PROJECT

RETURN TO PATH