A* Path Following

Lab 6 | Team 3 | 04/26/2024

Introduction

Nicholas

In this report, we detail our approach to developing a high-level navigation algorithm to create a path on a given map and drive the robot using that path. The algorithm only needs a start point, an end point, and a map to work. The algorithm first plots a path on the map to reach the end from the start, avoiding any obstacles along the way. Then, it drives the robot following this path until it reaches the end, keeping track of its position along the path to navigate. Such a navigation algorithm is crucial to the development of an autonomous driving algorithm, as without it the robot is unable to come up with high-level driving.

This algorithm is made up of four other algorithms working in tandem: a path-planning algorithm, a driver, a localization algorithm, and a safety controller. We drew on and improved the Monte Carlo Particle Filter and the Safety Controller from our previous projects. We will focus on the implementation of the A* Path Planning Algorithm and our Pursuit Controller Trajectory Follower for this report.

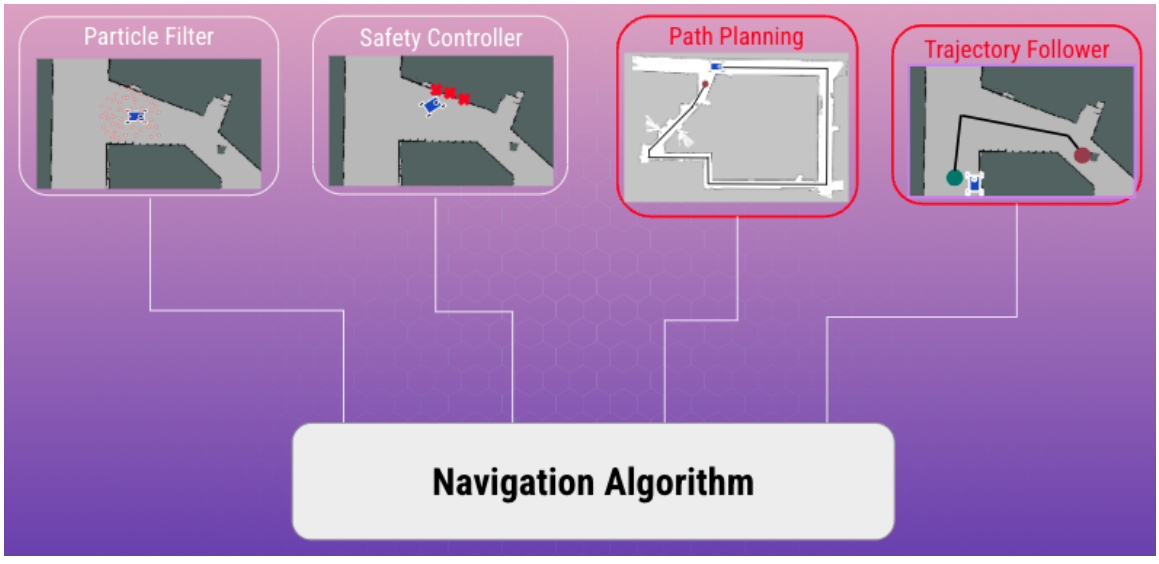

Figure 1: **The four individual algorithms that together make up the navigation algorithm.** The Monte Carlo Particle Filter and the Safety Controller have been adapted from our previous projects. We will focus on the implementation of the A* Path Planning Algorithm and our Pursuit Controller Trajectory Follower for this report.

Technical Approach

Autumn, Nicholas, Lennie, Michael

Our implementation is split into two modules: path planning and path following. In the **Initial Setup**, we go over the relevant data acquired from the racecar that the implementation needs. The **Technical Approach** section explains the theory behind the implementation and clarifies how each building block contributes to driving the car. Finally, the **ROS Implementation** section illustrates specific design choices we made.

Initial Setup:

For the Pure Pursuit Trajectory Follower, the ROS node is initialized by executing the trajectory_follower.py script. The node subscribes to the odometry data from the specified topic, /pf/pose/odom, and the PoseArray representing the planned trajectory from /trajectory/current.

The Trajectory Planner node is initialized using the trajectory_planner.py script. It subscribes to the occupancy grid map, goal pose, and initial pose topics, enabling it to receive environment information and user-specified navigation goals. It also declares parameters for configuring topics related to map, start pose, and end pose topics.

Path Planning:

We chose to use the A* algorithm in order to plan our trajectory. The A* algorithm is a state-space search algorithm. As compared to sampling-based algorithms like RRT*, A* is slower and more difficult to tune, but returns more sensible and closer-to-optimal solutions. A* was chosen as for this lab, we were confident we could design an implementation of A* that would be able to plan a path fast enough to race while maintaining the stronger optimality of search-based algorithms.

Being a state-space search algorithm means that A* relies on the engineers' transformation of the map into a configuration space and the discretization of that space into points. Thankfully, the

map is already represented as a discretized grid called an OccupancyGrid in ROS, so we used that as a way to represent the map. However, the OccupancyGrid's points are not represented in the same coordinate frame as the Start Pose and End Pose. Therefore, we need to do a series of transformations in order to get the inputs and outputs at various stages into the right. A visualization of these transformations can be found below.

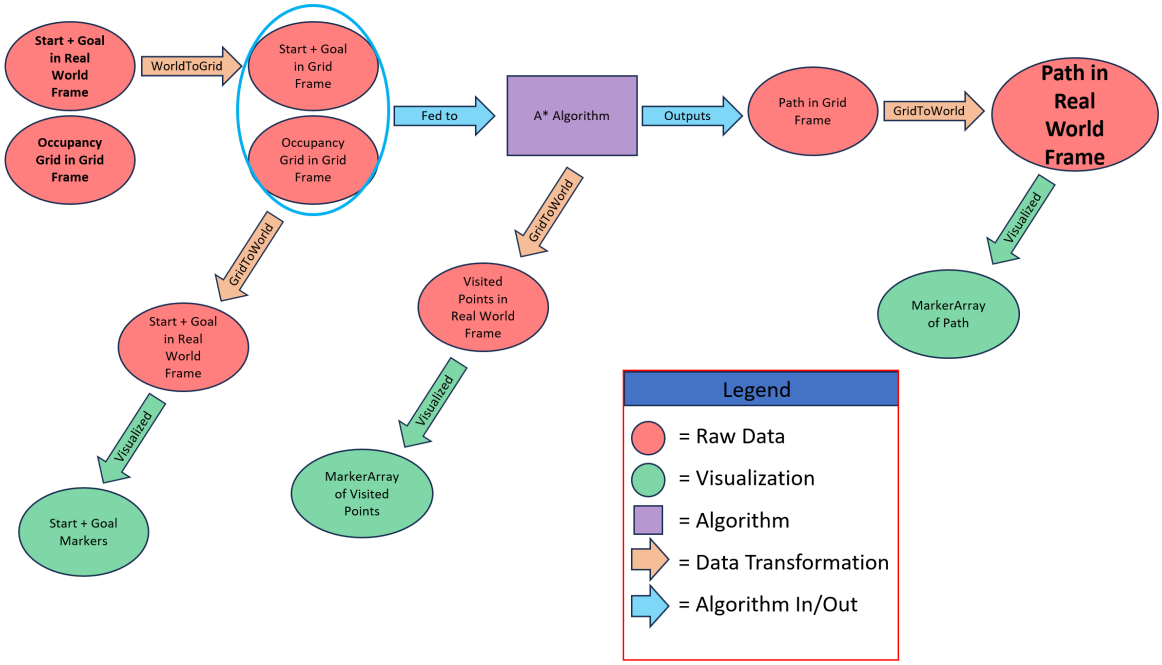

Figure 2: Conceptual visualization of the data pipeline for the path planning algorithm. The nature of the map and poses being in different frames means that the data needs to be transformed repeatedly. The data is first transformed into the map frame for use with the A* algorithm, then transformed back into the pixel frame in order to be visualized and used as coordinates on the actual map.

$$\mathbf{p}' = (\mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}) \cdot \mathbf{p} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 3: Matrix transformations for getting from grid to world frames. The translation, rotation, and scaling parameters were obtained directly from the OccupancyGrid, and we only had to express it in the matrix form. To make our code more easy to read and use, we implemented helper functions to do the converting.

In addition to these transformations, a filter was also applied to the map OccupancyGrid to add tolerance for walls. The A* algorithm will generate paths that run directly alongside walls and cut corners very closely in an attempt to minimize the total distance of its path. While this may be okay in simulation, driving too close to walls or corners is dangerous for the actual robot. Therefore, a dilation filter was used to thicken the walls in the map that robot receives. Doing so allows the A* algorithm to generate optimal paths with respect to the thicker walls, which will leave some extra space when the path is visualized on the unfiltered map. A visualization of the original map and filtered map is shown below.

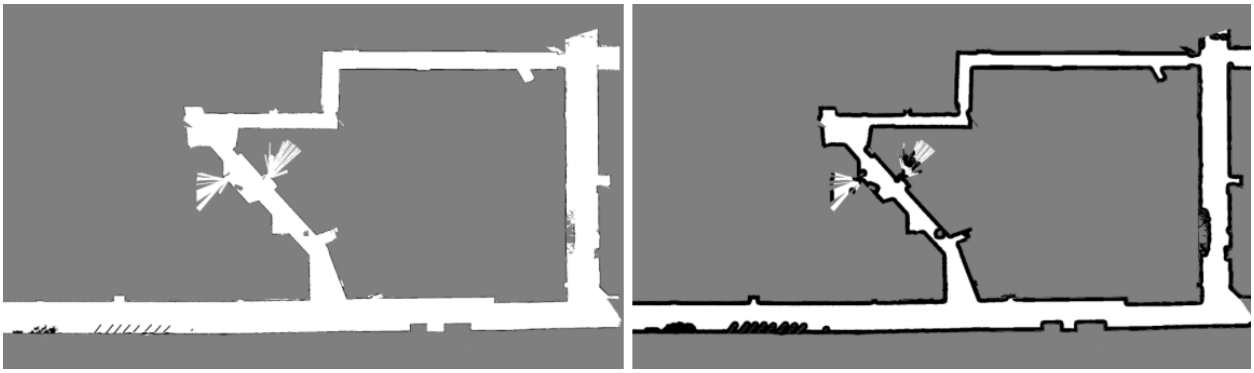

Figure 4: Original map (left) compared to filtered map (right). Applying a dilation filter using disks of radius 10 px increases the thickness of the walls on the map. Both images are acquired directly from the map OccupancyGrid, where unoccupied locations are shown in white, occupied locations are shown in black, and unknown locations are shown in gray.

We chose to represent our trajectory as a piecewise representation, which utilizes a series of discrete poses to define the trajectory. As trajectories can be represented using a PoseArray, this approach offers simplicity and ease of implementation, essentially connecting the dots between consecutive poses. By doing so, the robot can follow the trajectory by sequentially navigating from one pose to the next.

While the piecewise representation method is straightforward and intuitive, there are limitations, particularly in scenarios involving curved trajectories. Since this representation is discrete, the resulting trajectories may not exhibit smoothness, especially around curves. This lack of smoothness can lead to suboptimal performance, as the robot's motion may appear jerky or imprecise. Additionally, abrupt changes in direction between consecutive poses can introduce issues such as overshooting or oscillations, particularly at higher speeds. Despite these performance concerns, the simplicity and implementation ease of the piecewise representation outweighs the need for smoothness.

We debugged our algorithm in a variety of ways. First, we checked that our transformation matrix for WorldToGrid was the inverse of GridToWorld by performing both transformations on the start and goal markers and then plotting those points on the map. If the point appears where it is supposed to on the map, that means that the transformations invert each other correctly. Next, we used WorldToGrid on the start pose in various places. We checked if the grid cell was occupied in a location where the real-world location was either a wall, the normal floor, or out of bounds. This tells us whether or not the WorldToGrid transformation was working properly; if the cell was considered “occupied” when the cell shouldn’t be, that meant that the transformation was broken and needed to be fixed. With both of these tests, we can confirm that the transformations are working as they should, and can safely use them to transform from one frame to the other. Finally, we tested A* by visualizing the set of points that it has visited, and seeing how it progresses. This allowed us to both see how long it takes to reach a point and the decisions it makes to draw a path.

Path Following:

The trajectory follower algorithm is designed to facilitate the autonomous navigation of a robot along a predefined trajectory. It subscribes to the robot’s odometry and trajectory to receive information about the pose and the planned trajectory. A visualization of the data pipeline for the path follower is shown below.

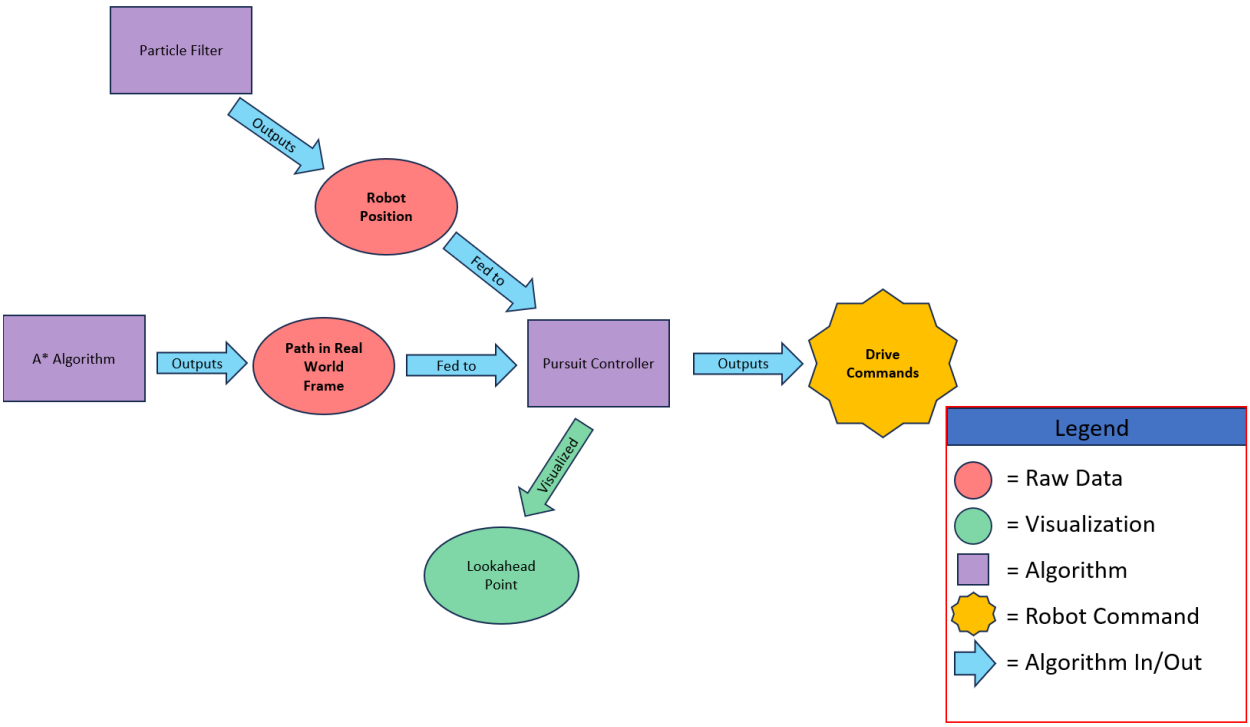

Figure 5: **Conceptual visualization of the data pipeline for the trajectory follower algorithm.** The pursuit controller takes in the output of the particle filter and the A* algorithm, and outputs drive commands following the path given. We also visualize the lookahead point as a debugging tool.

Based on this information, it calculates a control command to steer the robot and control its speed along the trajectory. Within pure pursuit, the primary challenge was determining the lookahead distance, and finding the lookahead point.

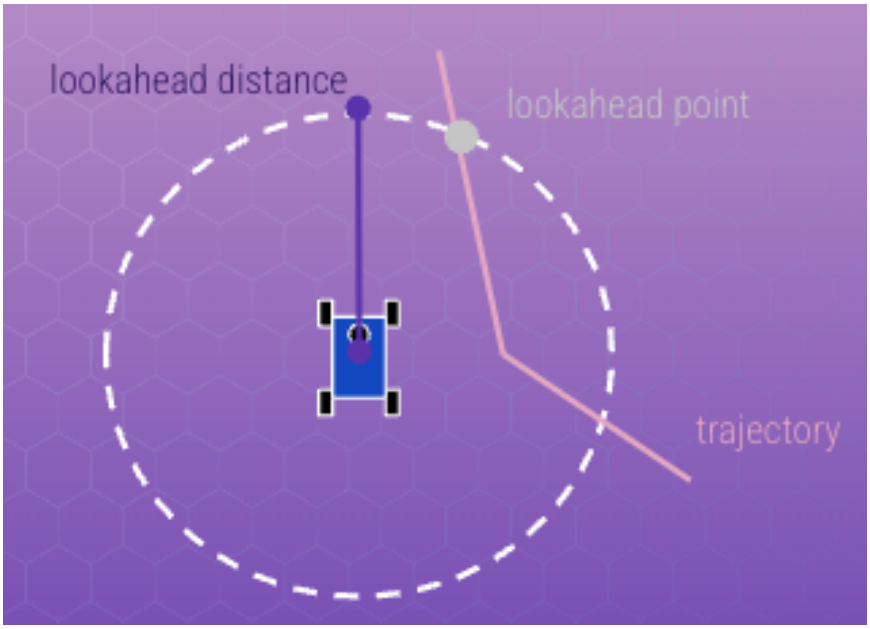

Figure 6: **Conceptual visualization of the relationship between the robot's lookahead distance, trajectory and lookahead point.** The image shows the radius is of the circle surrounding the robot is known as the lookahead distance, and the lookahead point being the intersection between the trajectory the circle created from the lookahead distance.

The lookahead distance plays a crucial role in determining the behavior of the trajectory follower. Within the algorithm, the lookahead distance is dynamically adjusted based on the curvature of the trajectory segment currently being traversed. When the robot driving through straighter segments of the trajectory, the lookahead distance is set to be relatively large value, typically 0.4 meters. This allows the robot to anticipate turns and adjust its trajectory accordingly. Then, when the robot approaches a turn, the lookahead distance is reduced to 0.1 meters. This is done to ensure tighter control and smoother navigation around curves.

Based on the lookahead distance which is the radius around the robot, the lookahead point is then calculated, determining the target point on the trajectory towards which the robot navigates. Similar to the lookahead distance, the lookahead point is also dynamic, as it's based on the current pose of the robot and the characteristics of the current trajectory segment. First the trajectory is divided into segments defined by consecutive points, for each segment, the distance between the robot's current position and the line connecting the segment's endpoints is calculated. The distance calculated will be used to measure the robot's proximity to the trajectory.

From these distances, the minimum distance, indicating the closest distance to the trajectory, will be used to calculate the lookahead point. To do this, the robot's current pose is projected onto the segment and extends a line from the projection point by a distance equal to the lookahead distance. The point where this line intersects the trajectory segment becomes the lookahead point.

ROS Implementation:

The Pure Pursuit Trajectory Follower and Trajectory Planner are two integral components of an autonomous navigation system implemented in ROS. The Trajectory Planner (tajectory_planner.py), generates an optimal trajectory for the robot to follow. The resulting trajectory, represented as a PoseArray message, is published to the /trajectory/current topic for the Pure Pursuit Trajectory Follower.

The Trajectory Follower (trajectory_follower.py), handles the driving commands including steering the robot along a planned path. It initializes by subscribing to the odometry data from the topic specified by the 'odom_topic' parameter, /pf/pose/odom, provided from the particle filter and the planned trajectory from /trajectory/current, represented as a PoseArray. Based on this data (odometry data and the planned trajectory) the Pure Pursuit algorithm to calculate steering commands. The node then publishes the lookahead point to the /lookahead_pt topic for visualization in RViz and computes a distance error metric for an accuracy evaluation, published to the /distance_error topic.

Experimental Evaluation

Michael, James

In order to properly evaluate the performance of the algorithms described in **Technical Approach**, several metrics were created and tested, both in simulation and on the actual robot. The **Simulation Evaluation** section will focus on the optimality of our A* and Pure Pursuit algorithms in simulation, and the **Robot Evaluation** section will discuss how these algorithms performed once implemented on the actual robot. Both sections will have a strong emphasis on the metrics used during the experimental evaluation of our algorithms.

Simulation Evaluation

The first metric used for simulation evaluation characterized the performance of our A* algorithm. As part of the lab, three example trajectories - trajectory_1.traj, trajectory_2.traj, and trajectory_3.traj - were provided. To test the performance of our A* algorithm, we compared the path distance of the example trajectories to the path distance obtained from our implementation of A*. Since the example trajectories each provided optimal paths between two points on the map, this metric was a valuable way to determine the accuracy of our algorithm. An illustration of our A* algorithm performance is shown below.

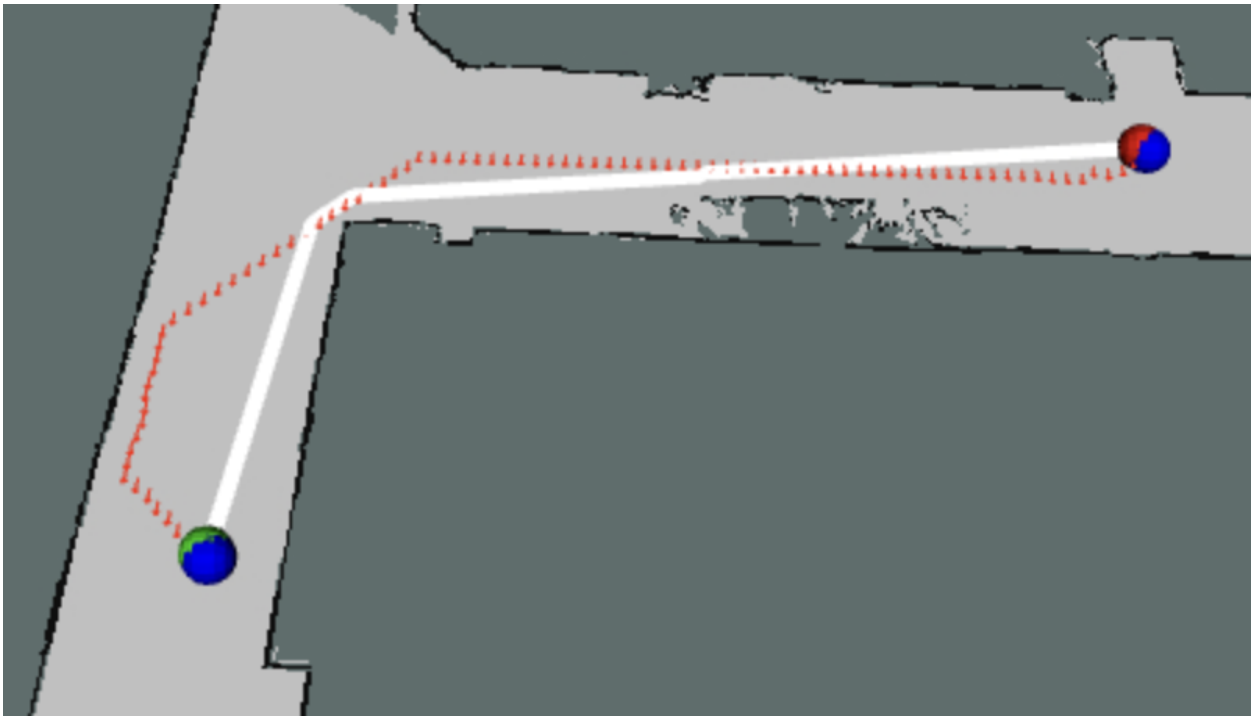

Figure 7: Evaluation of A* algorithm performance. When run alongside trajectory_2.traj, our A* algorithm produces a path that is only slightly longer in terms of total distance. The distance of trajectory_2.traj is 25.09 m, and our algorithm generates a path that is 27.73 m.

The second metric used for simulation evaluation focused on the optimality of our pure pursuit controller. With a perfectly ideal pure pursuit controller, the robot would not diverge from the planned path throughout its entire trajectory. So, in order to quantify the performance of our pure pursuit algorithm, we created a distance error metric to measure the distance between the robot and the planned path as it traveled along the desired trajectory. Throughout several simulation tests, the robot would perfectly follow the planned path while driving straight, but during turns, it would slightly deviate from its intended location on the path. We attempted to mitigate this error

by lowering the lookahead distance during turns, but even with this strategy, the control algorithm still had slight error.

A rqt graph was used to visualize this distance error metric, as seen below.

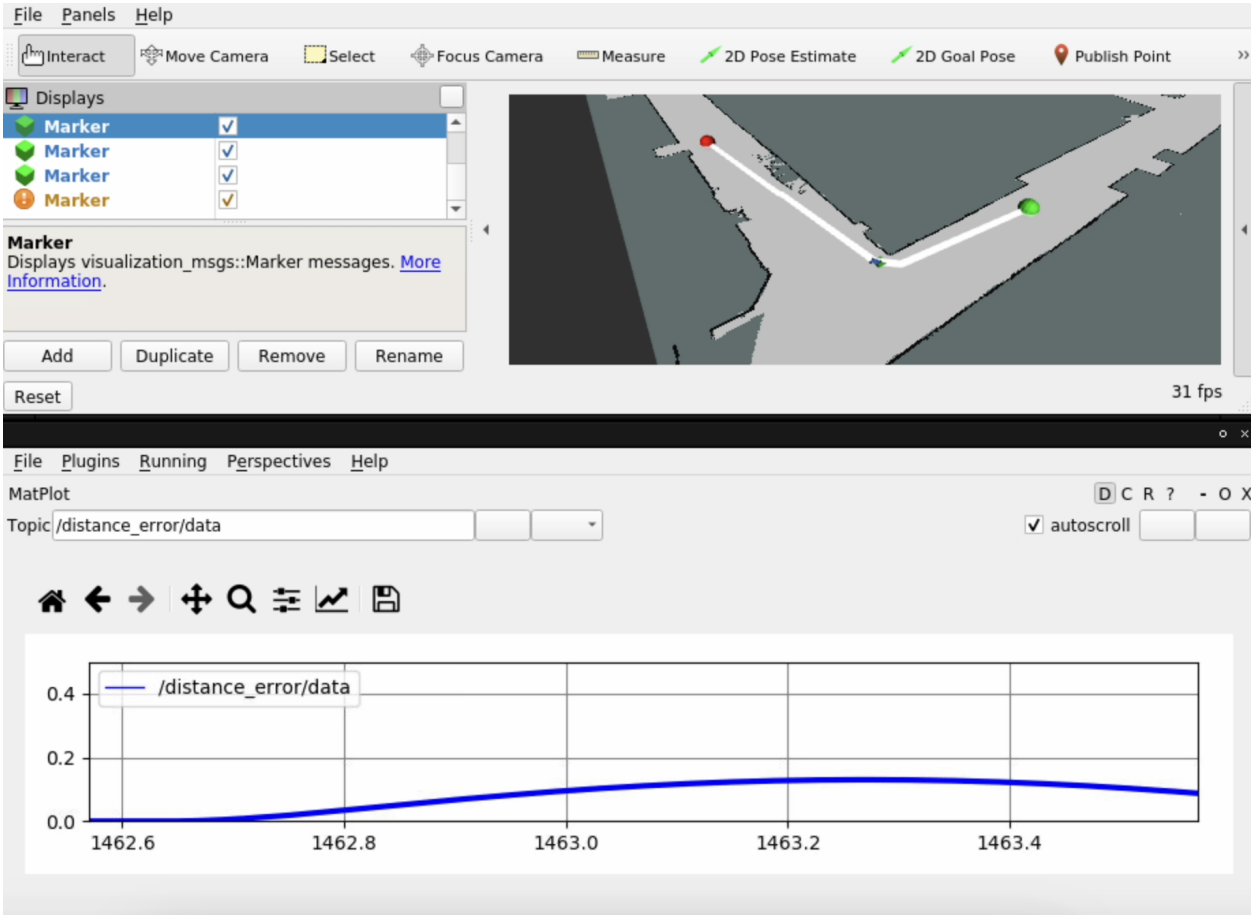

Figure 8: Illustration of distance error metric during a turn. The distance error metric, which measures the distance between the robot and the planned path, briefly spikes during a turn. During the straight segments of the path, the distance error remained constant at 0. However, during a turn, the error will slightly increase to ~ 0.2 m before returning back to 0.

Robot Evaluation

There were numerous issues with robot implementation including map loading failures, planner not receiving poses, particle filter problems, and code compatibility issues. Several solutions were tried to address the problems including rewriting launch files and parameters to fix issues that were suspected to be from incorrect topics or run order. For example, our particle filter having misshaped arrays for its particles, trajectory planner not receiving a map, and the map

being unable to be loaded from the launch command. There were additional problems such as a drive command being inputted and reflected on RVIZ, but the robot remaining stationary. Despite having one successful run with a trajectory being built and the robot following the line, the robot movement was extremely unsteady. Further tests also led to more inconsistent results as the trajectory planner was unable to start up again with the node not receiving start and goal poses. The code is currently being iterated on to reduce lag and possibly eliminate these on board problems.

Conclusion

Lennie

In this lab, we achieved a working A* path finding algorithm and pure pursuit controller that our racecar can use to find the shortest path to a destination in a known environment. Using a modularized approach, we integrated algorithms implemented in the current and previous labs: path planning, pure pursuit trajectory following, localization, and safety controller.

The path planner notably makes a tradeoff between computational efficiency and solution optimality. It's important for quickly deciding the path the car should follow, especially in the final challenge. The path following algorithm: the lookahead distance is different for when the robot is driving in a straight line versus a tight corner. With both of these algorithms, implementing visualizations at various points allowed for fine-tuning and general comprehension.

Looking ahead, we are focused on improving our teamwork as well as improving the debugging process. One improvement that was extremely helpful was the inclusion of a brief guide on how to run the code and visualize in RViz. In the future, documenting the process and keeping everyone on the same process will help us greatly in the final challenge. It will be exciting to finally integrate all the parts of the lab including the wall follower and visual servoing.

Personal reflections

Autumn:

Navigating through this lab proved to be a daunting task, marked by numerous hurdles that strained both our collective morale and individual contributions. I personally struggled to keep up and contribute significantly since I was gone both weekends and was slammed with midterms during the week in between. We ended up pulling many late nights and even an all-nighter to get the code working, but despite our tireless efforts we are still facing a lot of errors. I think one of our biggest issues was communication and organization. In retrospect, I think our issues stemmed not only from the technical challenges of this lab but also from the shortcomings in our team dynamics and communication. Everyone was really busy these last couple weeks, thus making it hard to coordinate and get things done efficiently. Looking back, I think it would have helped if we all knew what everyone else was working on and their technical approaches before the presentation and report. That way, if someone had to step out, someone else could easily step in without too much trouble. We need to work on keeping each other in the loop and making sure we're all on the same page, especially when things get hectic.

James:

This lab was an challenging experience due to technical difficulties in debugging, solving issues with the algorithms, and implementing it on the robot. Better planning and communication would have helped mitigated these issues. It could have helped us pace out and distribute the work to manage the technical problems that were rising up. Additionally, robot implementation proved to be remarkably difficult particularly due to our previous localization package not performing well with our path planning package. These issues with previous code implementation made me realize the importance of designing well thought out code with futureproofing. It is likely that previous code will be revisited to be streamlined and iterated on to help prevent issues such as this from occurring again.

Lennie:

In this lab, I learned about implementing path planning with ROS. I have previously coded A* in previous classes before, but integrating it into the ROS framework was a challenge. I had trouble figuring out the conversions between the map and grid spaces as well as when each should be used. Additionally, optimizing the code was also interesting. Looking back, improvements in team dynamics could have helped our team be more efficient with our time given the amount of things we had to get right in such a short amount of time.

Michael:

In this lab, I learned a lot about search algorithms and their implementations in ROS. From a fundamental standpoint, I was familiar with several different search algorithms prior to this lab, but A* was one of the algorithms that I had never worked with. I enjoyed learning about its overall structure and its usage for robot path planning. I also enjoyed working with our algorithms in simulation and fine-tuning different map filters to generate optimal real-world paths. However, implementing our algorithms on the robot proved to be incredibly challenging. Even after several long nights of debugging, we still struggled to test our path planning algorithms on the robot. In hindsight, we likely could have structured our time better as a team to prevent this last-minute scramble, but even so, I think we just drastically underestimated the challenges that we would face during the robot debugging phase (and the amount of time it would take). Even with the struggles that we had leading up to the briefing, I was happy with how we presented our work on this lab during the briefing.

Nicholas:

In this lab, I learned about the importance of modeling our solution to a problem very carefully. Very often in this lab, I found problems that couldn't be solved by simply tuning some parameters. We had to find ways to optimize the underlying algorithm and come up with our own unit tests to make sure that the algorithm was working the way we expected it to.

In order to make sure we captured everything about the problem as possible, I started the lab by making a list of all the design choices we would have to make, and all the metrics we would have to come up with to test our algorithm. I figured by doing that, I would be able to make the lab easy to break down and compartmentalize. Not so! The areas that ended up being challenging were a lot more difficult to predict than I thought, and we had to amend our preliminary scheduling time and again. I learned that engineering plans have to be amended early and often, as the unexpected is very common in robotics. I also learned that my plans will have to be a lot more nuanced in the future. We need to capture exactly how we will implement certain parts and build unit tests for our programs, as debugging is by far the biggest time consumer and frustration-inducing part of our work. It is also a great way to demonstrate how we have planned out our work to our professors and employers!

As for communication, I found high-level diagrams and conceptual breakdowns to be helpful for the team as well as for our audience. Initially, I thought those breakdowns would be redundant for teammates who are in the thick of the implementation, but it turned out to be quite useful as we ended up referring to those diagrams repeatedly throughout our implementation.

In short, I learned that high-level planning and schematic generation is a lot more difficult than it looks, even when you spend several hours in advance trying to put it all together! It is best to try to update them as needed and keep everyone on the same page; after all, some things cannot be anticipated.