Fine Obstacle Avoidance Using Laser Scan and Novel A* Algorithm

Shiva Sreeram, Neil Janwani

1 Introduction

All robots and systems are subject to a variety of limitations. When in a restrictive environment, the robot will have to contend with the sensor readings being less reliable. As such, additional algorithms and methodologies have to be applied in order to handle the extreme edge cases that will appear due to accumulated error. This project seeks to address this topic by sending a two-wheel drive robot equipped with encoders, a gyro, and an Intel Realsense camera (shown in Figure 1 below) through a myriad of pathways consisting of unknown obstacles while avoiding collisions and ultimately arriving at a goal point in a map.

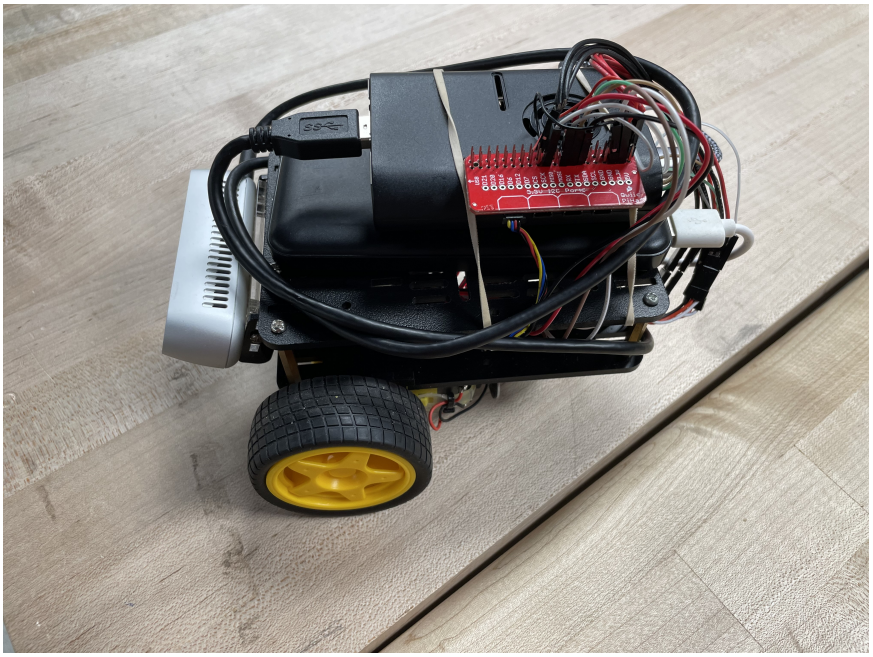

Figure 1: The robot utilized in this project.

2 Approach

2.1 Wheel Control and Odometry

We begin by briefly going over some of the lower level functionality; the wheel control node reads in the encoders and gyro to determine the current state of the robot while also receiving velocity commands to determine what the desired state of our robot is. Due to the amount of noise that can come in from these readings, we also apply a basic filter to keep it smooth. We then apply proportional feedback control to send PWM commands to the motors to ensure our bot arrives at the desired position. It then publishes the actual state the wheels are in (their velocities) as well as the θ reading from the gyro. We can see the accuracy of the wheel states from the plot of our week 3 goals:

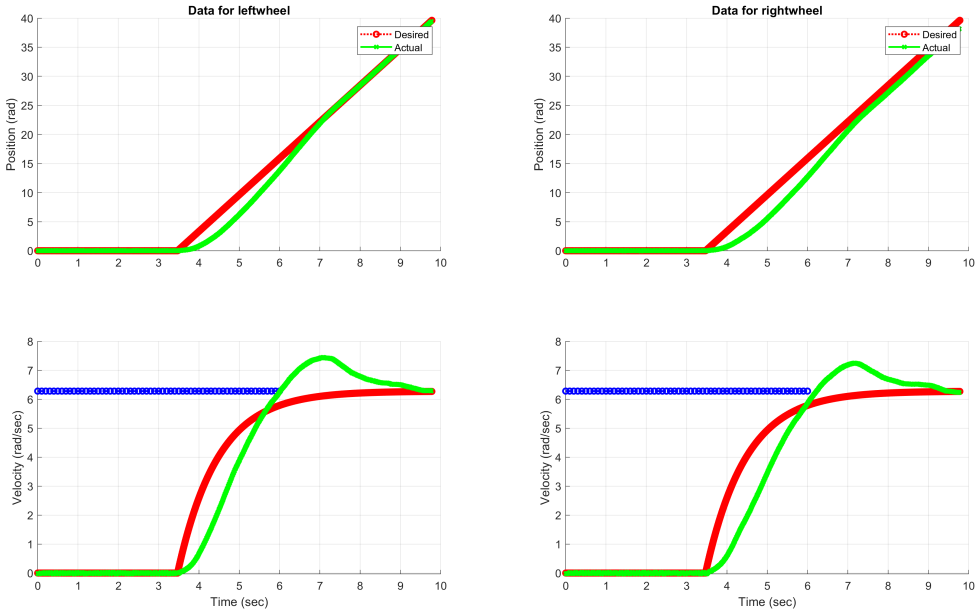

Figure 2: These images were included in our Week 3 goals. We see that the desired velocity and position defined by the blue line's input velocity were achieved.

The odometry node is at a higher level than wheel control and effectively handles the basic work required to place the robot into an RViz visualization. Reading in the wheel state and gyro reading that are published by the wheel control node, this node then publishes the ROS point, quaternion, and twist in an odometry frame so that we are able to see the bot's actual movement align in the visualization. This node as well as wheel control are not unique implementations but are fundamental building blocks to the implementations ahead.

2.2 Localization

Given that we are navigating through a large map, the use of encoders will not be completely sufficient due to the potential of wheel slippage and due to error accumulation during gyroscope w_z integration. Due to these issues, after a bit of exploration the robot's location in the map would be completely lost. To account for this, we apply localization. This node primarily handles the relationship between the odometry and map frames and updates this as necessary. We created a wall point map in the grid frame (with a resolution of one inch by one inch per grid point) of the closest wall point to each point within the map via a brush fire algorithm and saved it as a csv file. We then read in the laser scan message from the camera, converting this to map then grid frame. We create a mapping of laser scan points to nearest wall points and then run a minimize least squares algorithm. This determines the delta x, delta y, and delta θ that we need to adjust the map to odom frame transform to match up with the map. A factor of one-tenth is applied to prevent too sudden of adjustments.

All of these aforementioned components are fairly standard for the localization process but there are a few unique components to our implementation. First of all, we don't even start the process until after receiving the first pose as, before that point, the process is quite pointless. Another relatively simple one is that while the robot is moving, we apply a small factor to the localization (one-twentieth) as we do not want to accidentally localize to the wrong points as the robot is spinning (when the error is highest). The most important component is that we are not localizing laser points that are too far from their corresponding wall point. This component is vital to our approach to obstacle handling which we will further discuss in its corresponding section. To further prevent unwieldy localization, we are also tracking the number of points we are using and apply the fraction of the number of points considered (in a given instance, or step, of localization) over the total number of laser scan points received. This prevents the robot from being subject to dramatic adjustments in the map while it can only see a small portion of the wall. To throttle this even more, we don't even apply localization if the number of points considered is less than fifty (which is likely when the robot can only see obstacles). Below is the localization process in action from the week 7 goals:

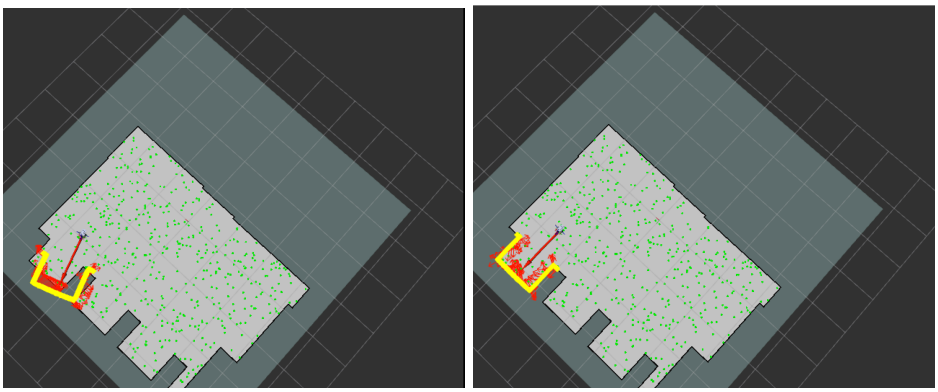

Figure 3: These images were included in our Week 7 goals. The yellow lines are the laser scan readings, the red arrows are messages we published of the direction the laser scan points were being moved (very helpful for debugging), and the green dots are PRM points from our earlier implementation (these work differently now as we will discuss in the next section. We see the robot receive an initial pose that is quite off from its actual position and after a brief amount of time, the robot is able to localize.

This process is very important as we need to start off in a very accurate position on the map as we begin to handle the obstacles. We also need to be able to readjust after we navigate through obstacles in case we partially lose our place in the map. In the figure below, we have a flowchart to dictate this node in a more digestible fashion:

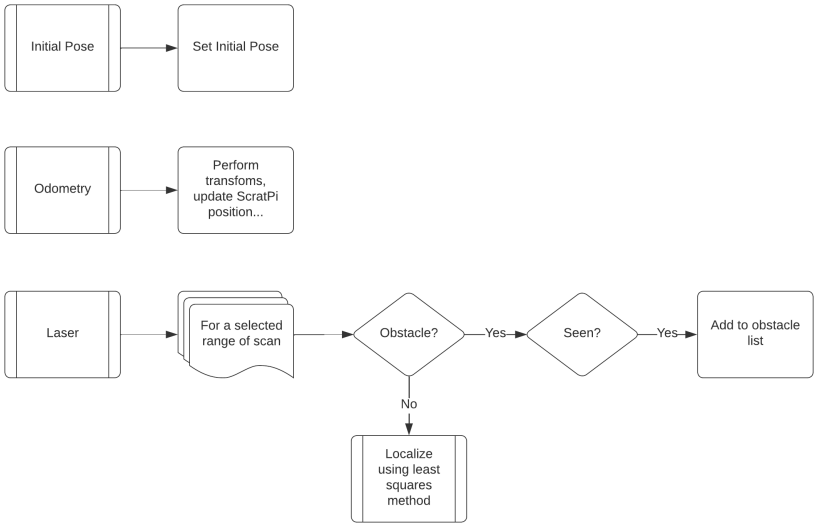

Figure 4: Flowchart of our localization node.

2.3 "Simple" Driver

2.3.1 Driver

Our whole navigation system works by setting goals and traversing to them from our current position. We have four stages which we have a state variable to manage:

1. Turning to goal: the robot turns to face the goal position.
2. Driving to goal: the robot drives to the goal though an omega factor is still applied to account for the scenario that the robot loses the heading to the goal.
3. Turning at goal: the robot turns to face the direction that the goal position defines.
4. At goal: the robot is at the goal and should no longer move until a new goal is set.

2.3.2 Planning

Our planning process can be concisely stated as a modified implementation of A* operating on PRM points. We start by generating 150 fully random points within the map (we will be adding more but this is discussed in the next section). The A* algorithm itself is the standard process; however, the connects to function and weights are quite unique. For our connects to function, we attempt to determine whether the path between two nodes gets *too close* to some object. To accomplish this task, we discretize the straight-line path between nodes v and w in 5 cm spacing. We chose this number as the robot is roughly 10 cm long, and thus by moving at 'half-increments' we can accurately tell whether a wall or obstacle is too close. While it is peculiar that we don't make 10 cm spacings, please note that we have yet to define *how close* we may get to a wall. Our bot is roughly 16 cm in width, and thus, we choose to allow our closest wall or obstacle distance to be 10 cm in either direction (from the center of the bot). By utilizing an already saved csv, which maps grid points to their closest wall, we can write this algorithm using constant time lookups, allowing the computation time to center around the *number* of obstacles inputted rather than re-computing an entirely new brushfire algorithm (which we will discuss in more detail momentarily). Using this connects to function, we may then generate a plan. However, recall that the weights for this plan are also unique, in contrast to traditional A*. While we consider the classic C_{togo}, we also define a new cost: C_{free}. C_{free} allows us to denote how *close* a given node is to a wall or obstacle. In this manner, by allowing low costs to correlate with free space in our map (we use an inverse function on nearest wall distance), we may prefer points in free space. In this way, our plan will tend to take wide corners as avoid obstacles as much as possible. In this way, we may reasonably avoid missing obstacles in our peripheral vision (as we try to take wide turns around corners). Using this generated plan, we may execute the path in the driver by dequeuing elements from the path and setting them as the next goal (the desired thetas for the waypoints in the path are set in the direction of the next waypoint). All of these components allow us to navigate an empty map just fine in addition to obstacles which we will focus on now.

2.4 Obstacles

2.4.1 Detecting

We briefly touched on this in our localization node but in order to detect an obstacle, we see if there are any localization points that are far from the walls and if these points are a meter or less away from the bot, we denote them as obstacles.

2.4.2 Remembering

Our solution for remembering is quite unique and can be seen as a very rudimentary version of SLAM where we start with a known base map but are filling it in. To do this we first only add an obstacle to a list in localize if it shows up ten times (to account for noise). We also round these obstacles to the nearest ten centimeters as a consequence of the potential variability in our location in the map versus its actual location and also to reduce the computation time it takes to consider these points. We then broadcast this from the localize node to be read in by the simple driver node. Inside simple driver, we perform a nontrivial process to update the wall point map:

1. Receive the obstacle in a callback function.
2. Convert this point to grid frame and denote the nearest point to it as itself.

3. We then apply a square "blur" out of this point of 40 centimeter width where we update the closest point to be the obstacle we just added as long as the current wall point it references is further away.

This "blur" process is to save on computation time and is a perfectly fine optimization as we are not localizing on these points and 20 centimeters is larger than the robot.

2.4.3 Replanning

In replanning, we encounter an interesting challenge. Of course, we wish to re-generate our plan the instant we know it is not valid. However, we must take into account errors in localization, driving, and the unfortunate time-delay in 'remembering' obstacles (as noted above). Thus, in reality we wish to stop on two separate conditions.

1. We see an obstacle directly in front of us (and in our path)

2. We detect that an obstacle lies somewhere along our future path

Following these conditions inevitably raises a host of race conditions, or conditions related to multi-processing (common with callback functions). Thus, we need to enable many flags (to not re-plan while we are generating a new plan, for example) in order to avoid such bugs. Note that our PRM density is not necessarily high enough to generate acceptable paths through obstacles. To circumvent this issue, we implemented a feature to 'explode' PRM nodes along points of obstacle or wall collision. For example, if we detect that some portion of our path is intersected by some obstacle or wall, we 'explode' PRM points at that intersection point. In the other case, if we get too close to an obstacle we simply wish to 'explode' PRM points around ScratPi itself (as we need to somehow circumvent the obstacle with our unique A* planning). In fact, this A* planning is exactly what makes this process efficient. Recall that we implement a weight, C_{free} that enables us to *prefer* more free space nodes rather than nodes close to obstacles. Thus, our A* implementation will actually 'bow out' the path around corners, as can be seen below:

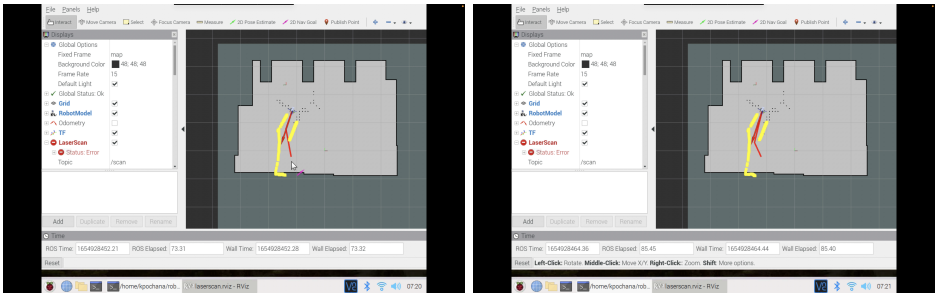

Figure 5: On the left, we have the prior plan. We note that some obstacles are seen, and thus the path 'bows out' in order to make room for the 'corner' (or so the robot currently thinks)

Finally, we include a flowchart to express this complicated, but simple, node cleanly. Of course, we lose a lot of detail here.

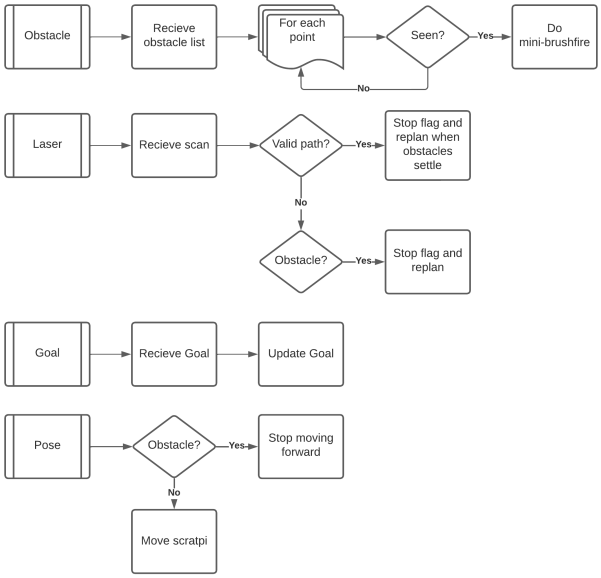

Figure 6: Flowchart representing our simple driver node

3 Results

We can see how this all comes together by going through the scenario presented in our video where we have our bot navigate through a fairly complicated setup across a large portion of the map. Below is the scenario we put our bot through:

Figure 7: The maze in the map.

We first send the robot to go back. As it continues, we see the robot add these points to the map (and also to the wall point map) as obstacles and continuously replan using our novel A* until it arrives at the first nav goal:

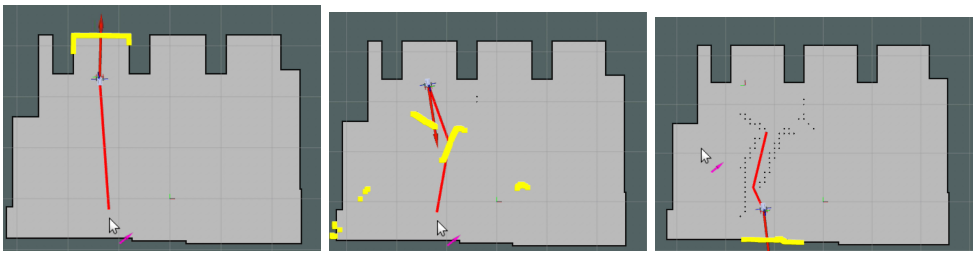

Figure 8: In the first image, the bot receives the first goal with no knowledge of any obstacles behind it. As such, the plan generates a straight path to the goal. As it traverses, we see that a replan occurs: the Simple Driver node realized that the plan would cross obstacles so it generated a new one. We see that this plan still will cross obstacles but this is due to a race condition where the obstacles are not added to the wall point map before the new plan is created. With the process of continual replanning however, we will not run into any obstacles. The robot was able to make it to the goal with the obstacles it encountered saved and the laser scan matches well with the wall of the playpen.

We are able to then navigate around the rest of the maze with two more goals applied. Throughout this process, the continual replanning with the novel A* prevented any collisions with walls or obstacles and the laser scan continued to line up well with the map due to localization. We can see these results shown below:

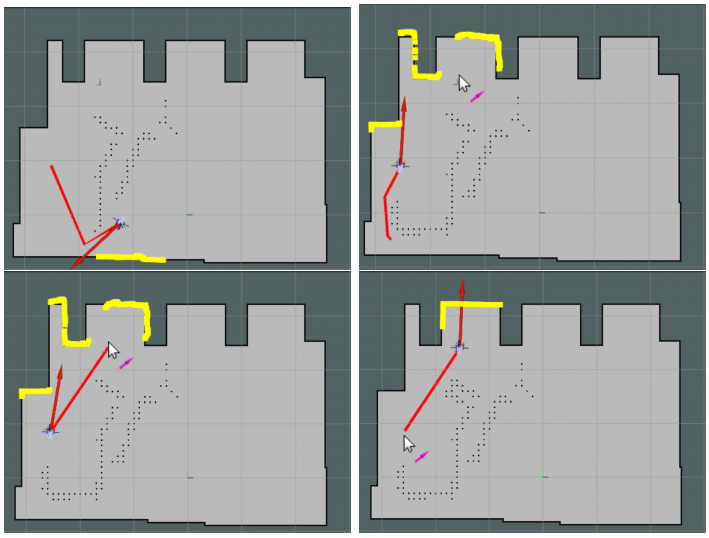

Figure 9: The first row of images define the bot receiving and arriving at the second nav goal where the second row defines the same for the third goal. We continue to see the same behavior that the bot experienced for the first goal: adding obstacles and replanning continuously along the path with localization keeping the robot aligned on the map.

Because we are remembering these points, we can now explore the map with where the novel A* accounts for all of the obstacles that have been added to the wall point map. As such new plans will require far less replanning (though we note that the initial navigation did not see all of the obstacles so occasional replanning is still necessary. We have a view of this here in the figure below:

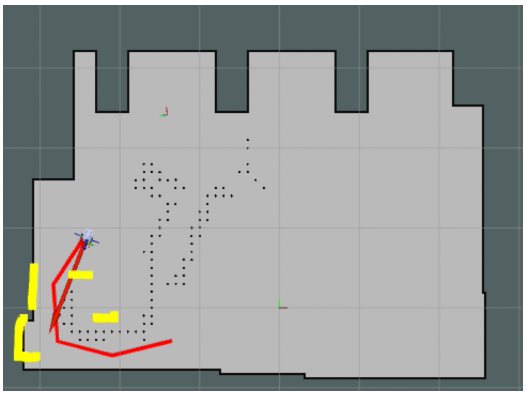

Figure 10: All of the obstacles have been remembered with few additional ones introduced. The plans generated are far less likely to require replanning.

4 Conclusion

This project demonstrates the capabilities of the robot in situations where it would be expected to fail. Utilizing the novel A* and wall point map, we were able to navigate the robot through extremely narrow spaces without running into a wall, despite the limitations with the wheel control and localization.
Speaking of the limitations with localization, an issue occurs when localizing along the straight edge of the wall, especially when objects are located along this edge. This definitely presents itself as an issue when there are no corners to localize to. Should one wish to continue this type of project, there are a lot of areas to improve in making the localization more robust.

5 Acknowledgements

Thank you very much Professor for all your help over the course of this term and throughout the school year. We really appreciate all the time you spent guiding us through robotics and dealing with our shenanigans. In addition, extend our thanks to the TAs for all their support.
We would also like to acknowledge Krishna Pochana for his work on the project in first and second term, and for his continued interest during this term. Furthermore, we would like to thank Agnim Argawal and Josh Hejna for their work during first and second term respectively.