Implementation of Simultaneous Localization and Mapping on a Differential Drive Robot

Zachary Serocki
Dept. of Robotics Engineering
Worcester Polytechnic Institute
Worcester, United States of America
zdserocki@wpi.edu

Sarah Listzwan
Dept. of Robotics Engineering
Worcester Polytechnic Institute
Worcester, United States of America
salistzwan@wpi.edu

Alexander Kraemling
Dept. of Robotics Engineering
Worcester Polytechnic Institute
Worcester, United States of America
ajkraemling@wpi.edu

Benjamin Penti
Dept. of Robotics Engineering
Worcester Polytechnic Institute
Worcester, United States of America
bjpenti@wpi.edu

***Abstract*—Particle Filter Simultaneous Localization and Mapping was implemented on a differential drive robot using the Robot Operating System (ROS) framework. These methods were implemented to allow the robot to map a static maze, then later be capable of localizing itself when placed within a random location in the maze. A* path planning was implemented to allow the robot to navigate the maze efficiently both during uninformed and informed navigation. The robot used was a TurtleBot 3, equipped with a LIDAR sensor and differential drive.**

***Index Terms*—SLAM, Pure Pursuit, Differential Drive, Particle Filter, A*, Graph Search**

I. Introduction

The TurtleBot was required to complete two distinct tasks, first exploring the maze until all frontiers were explored, while building a map. Then, using the previously constructed map, the robot was moved to a random location, and the robot was instructed via RVIZ to move to a specific location on the map. Two main SLAM packages were used to allow for informed and uninformed navigation. The Gmapping package was used during the mapping phase to construct a map using LIDAR readings, and it uses a Particle Filter to localize during motion. The AMCL package is a particle filter localization package that when given a map, determines the likely location of the robot. These packages were implemented to allow the robot to construct a map of an unknown environment, then navigate it.

II. Methods

1) The robot was moved to a random location in the maze. The robot, frontier_id, Gmapping, path_planner, and pure_pursuit nodes were initialized.
2) The robot begins constructing a map and determining its frontiers for exploration. The frontiers are ranked and the robot constructs a path to the best frontier.
3) The robot then moves to that frontier, and then repeats the process. The mapping stage is complete when the robot no longer has any viable frontiers.
4) The nodes are then deactivated, and the robot is moved to a random location in the map.
5) The robot, AMCL, path_planner, frontier_id, and the pure_pursuit nodes are then activated. When the correct location on the map is marked in RVIZ, the robot begins to determine its location in the map.
6) Once the robots location is determined to an acceptable level of certainty, the robot plans a path to the goal location and navigates there

III. Results

A. Architecture

The code base was organized into a number of nodes, each handling a distinct process. The nodes were connected via topics and services, allowing effective communication while maintaining modularity of the code (See Figure 1).

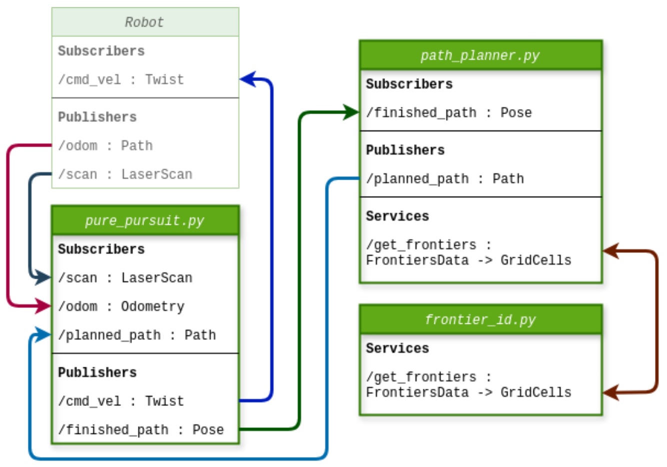

Fig. 1. Code Architecture Diagram

The Robot node publishes odometry and LIDAR Scan information to pure_pursuit, Gmapping, and AMCL. The path_planner node requests the current map using a map service. When it requests the map, it enlarges the walls of the map then requests frontiers to be identified via the get_frontiers service. The Frontier Identification Node computes the frontiers when requested by the get_frontiers service. The frontiers are computed and sorted based on quality. The path_planner node then receives these sorted frontiers and plans a path from the current location of the robot provided by the pure_pursuit node, to the goal frontier. This path is then sent to the pure_pursuit node. The pure_pursuit node navigates between these nodes by sending command velocity messages to the robot. The pure_pursuit node also keeps track of the closes object in front of it, to provide an e-stop if the robot is close to crashing. Once pure_pursuit has finished navigating its path, it sends its current location to the path_planner node to ask for a new one.

During informed navigation the robot starts in localization state in the pure_pursuit node. The pure_pursuit node will not publish its start location to begin the path planning process until the covariance of its position has dropped below a threshold. There are two covariance thresholds that are used to determine the behavior of the robot in its localizing state. The first threshold moves the robot from localization in place, to localization by spinning. Once the covariance drops below the second threshold, the robot moves into navigation state. The robot will then publish its current location, which will allow the path planning control flow to begin. Once the location has been chosen on RVIZ using the Goal Position feature, then the path will be constructed and the robot will navigate through the path.

B. Localization

The robot localizes during uninformed navigation using the odometry process model in the Gmapping particle filter. The Gmapping particle filter is used to update that frame as the wheels slip or as the IMU drifts overtime. Within the pure_pursuit node, the robot's current pose is kept in map frame, using a TransformListener. The Gmapping package publishes a transformation between the odom and the map frame to account for the shortcomings of odometry. This allows the planning of the paths to continue to occur in the map frame, even after the odometry and map frames drift over time.

During informed navigation, the robot localizes using the AMCL package. The AMCL Package publishes the most likely location of the robot as well as its covariance. Once the covariance in the robot's position is reduced to a certain level, the robot begins planning its path.

The robot begins in a localization state, when the robot localizes in place until the covariance dips below a threshold. The robot then begins spinning to continue to localize. Once the robots covariance dips below another threshold, the robot moves into its navigating state. In the navigating state, the robot plans and executes its path to a specified target.

C. Frontier Identification

The map is stored in an occupancy grid. Each cell in the occupancy grid contains a number that indicates its status. Above a threshold, the cell is occupied, below this threshold, the cell is unoccupied. If the value is -1 then the cell is unknown. The frontiers are defined by the boundary between unoccupied and unknown cells.

The frontiers are determined using edge detection in the CV2 library, a computer vision library. The unknown areas and the unoccupied ares are turned into their own separate binary masks. The unknown area is dilated by one pixel. A intersection is performed on the binary masks to find the intersection of unknown areas and the free space. The connected components are then isolated from this image using the CV2, ConnectedComponentsWithStats function. This function identifies the connected components and produces their height, width, and area. To determine the centroids of these frontiers, the width and the height of each frontiers are compared to determine which of these axes is the major axis. The dominant dimension is then used to determine the center in that axis. The array is then searched in that row/column to find the centroid (See Figure 2).

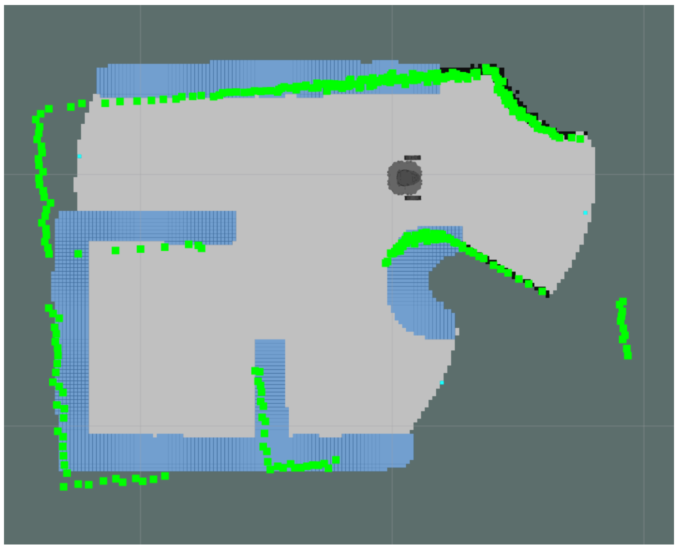

Fig. 2. Frontier Centroids (Shown in Cyan Blue)

These centroids are stored, along with the area, and passed to the sorting function. The sorting function sorts the centroids via a linear combination of the frontier lengths (areas), distance from the robots current position, density (how clustered the frontiers are), and distance from the start position. The distance from the robots current position is most heavily weighted to allow for smaller paths to be generated, making map traversal more efficient.

D. Path Planning

Path planning begins by receiving a current location of the robot from the pure_pursuit node. This indicates that pure_pursuit has completed its last path and would like a new one. The path_planner then requests a map from Gmapping, and computes the configuration space (c-space) of this map. This entails enlarging the walls of the map by the radius of the robot so the robot can be treated as a point in the center of the robot, without worry of the robots sides brushing the walls (See Figure 3).

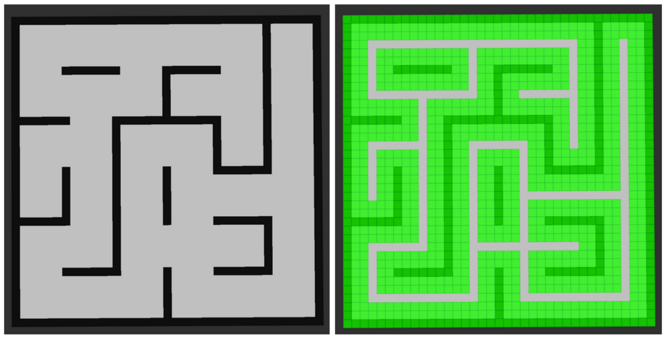
Fig. 3. Map Before and After C-Space Operation

In practice, this c-space must be adjusted to allow for consistent path planning. The walls must not be enlarged by the full radius of the robot to prevent frontiers from being lost due to inadequately large grid size. The robot is prevented from running into walls by implementing a predictive collision algorithm on the drive controller level, as well as path planning though the center of the open space. The unknown space is also enlarged by the same amount to prevent the frontiers from being located directly on the border of unknown space. This could cause the robot to run into the wall if the unknown space were a wall.

This c-spaced map is then passed to the frontier identification node, and when the frontiers are returned, path planning can commence. Given the current location and the goal location of the robot, the path can be computed. A* search is used to find the ideal path from the start to the end of the map [1]. A* search provides a fast and effective search algorithm that provides an optimal path by taking into account both cost to get to the cell and a heuristic, which in this case is the Euclidean distance from the current node to the goal. The cost was defined as a combination of the Euclidean distance from the start, as well as a cost for the distance to the nearest wall. This causes A* to plan a path in the center of the map, rather then taking simply the shortest path, which could cut close to walls and cause crashes due to small errors in mapping,localization, or pure_pursuit (See Figure 4).

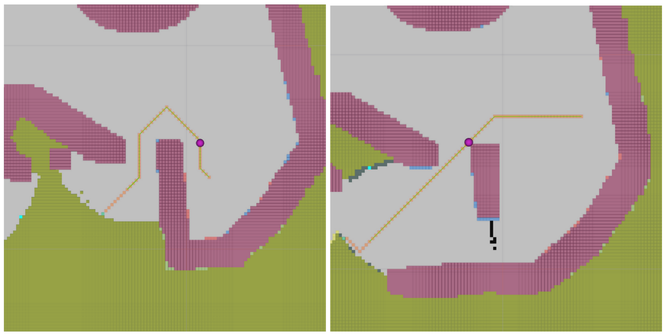
Fig. 4. A* With(left) and Without(right) Wall Avoidance Costs

This path produced by A* is then converted into global coordinates and sent to the drive controller (pure_pursuit) for actuation.

E. Drive Control

The drive controller in use is a Pure Pursuit controller. This has many advantages over a rotate-drive-rotate controller. The Pure Pursuit controller allows the robot to cut corners in the path, as well as minimize time spent turning. It also reduces the error in the final position as small turning errors in the rotate-drive-rotate can cause large errors when driving large distances.

The Pure Pursuit controller maps the path as a series of line segments. The look-ahead distance is set and the intersection of the path with a circle of that radius is driven towards by the robot. This is repeated continuously until the robot reaches the end of the path.

The Drive Controller also utilizes a predictive collision algorithm, to prevent the robot from crashing into unforeseen objects. The robot monitors the distance of the LaserScans produced in the front 120^o of the robot. If the smallest LaserScan is less than 13 cm, then the robot is automatically stopped, and it requests a new path, which requires the c-spaces to be recomputed. This allows the robot to adapt to unexpected environmental variables, or small errors in localization or mapping.

IV. Discussion

This lab project aimed to implement SLAM on a differential drive TurtleBot equipped with a LIDAR sensor to allow for navigation through a maze. Various techniques were integrated to allow this robot to functionally navigate and map in the real world, adapting to imperfect sensor readings and odometry.

A. Robot Performance

1) Mapping: The robot was able to reliably map the field given the field had consistency floor conditions, and the robot could physically fit between all obstacles in some way. This allowed the robot to navigate and map the field, until all unknown frontiers were explored. If the field contained a path that the robot could not physically fit through, the frontier would be eliminated by c-spacing and would not be explored.

The inconsistencies in odometry were compensated for by transforming the odometry frame into the map frame, and executing all planning and actuation from the map frame.

2) Informed Navigation: The robot is able to localize within 15-20 seconds in the map. Once localization has occurred and the user has chosen the desired end point on the map, the robot navigates to the end point. If the robots covariance in position rises high enough, the robot returns to localizing mode. In practice this has rarely occurred as the robot was driven slow enough to reliably localize in the map.

B. Techniques for Improved Reliability

To improve reliability in the real world, various techniques were implemented including early path exit, c-space adjustments and an collision avoidance functionality. An early path exit was implemented into the path_planner node. This prevents the full path from being sent to the robot during the mapping phase. This prevents the robot from getting too close to the unknown space, which could be a wall, and detrimental to complete the path. This early path exit allows the robot to move though most of the path, but by stopping before reaching the end, the robot will have more information about its surroundings and can then make a more informed decision on if it would like to continue in the same direction, or if all the unknown space in that area had already been mapped.

In order to prevent losing frontiers due to the inaccuracy of using a map of grid cells, we ensured our c-space was smaller than the radius of the robot. Although this does create the possibility of the robot brushing the wall, this is overcome by developing a drive controller level predictive collision algorithm.

The collision predictive alert was created as a map-independent technique for preventing collisions caused by a change in the environment, or an error in the mapping or localization. This technique allows the robot to stop and re-evaluate its environment when it comes close to running into an obstacle. Allowing the robot to adjust its path based on current conditions, and plan a better path around the obstacle.

C. Future Work

To improve this implementation, a few features could be implemented, including an improved drive controller, as well as an improved map-independent wall avoidance algorithm.

A drive controller with better tuned turning gains and improved speed, as well as a Model Predictive Path Integral Controller could be implemented [2]. This is a type of Model Predictive Control (MPC) scheme that allows for the ideal inputs to the command velocity to be determined to allow the robot to follow the path most closely. These are determined by choosing random inputs and simulating system response over a given number of steps. This process is repeated a large number of times and the input with the best response is fed into the real robot. Given the ROS node structure, this control could be implemented on a dedicated computer, providing the increased computing resources needed while still integrating into the existing structure.

To avoid walls and dynamic obsticals more effectively without having to re-plan a path, an Artificial Potential Fields approach could be used [3]. This approach simulates the robot as a particle with the same charge as the obstacles. The robot aims to seek the path of least resistance, which is as far away from the obstacles as possible. This could be used to prevent the robot from hitting the walls when there are errors in the mapping or localization, or an unexpected change in the environment. This could be integrated into the drive controller, allowing for a map-independent method for avoiding obstacles.

V. Conclusion

In this project we implemented a robot that can autonomously explore and generate a map, as well localize in that map. SLAM was implemented within the ROS framework on a TurtleBot 3 robot. The Gmapping and AMCL packages were used to generate and localize in the map without prior knowledge of the environment.

Thanks to the various techniques we implemented, our robot was able to meet all criteria, including mapping the maze without making collisions with the wall, driving back to start position, and being able to adequately localize anywhere within the map.

References

[1] P. E. Hart, N. J. Nilsson and B. Raphael, ”A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” in IEEE Transactions on Systems Science and Cybernetics, vol. 4, no. 2, pp. 100-107, July 1968, doi: 10.1109/TSSC.1968.300136. keywords: Costs;Mathematical programming;Minimization methods;Functional programming;Automatic control;Minimax techniques;Gradient methods;Chemical technology;Automatic programming;Instruction sets,

[2] W. Wu, Z. Chen and H. Zhao, ”Model Predictive Path Integral Control based on Model Sampling,” 2019 2nd International Conference of Intelligent Robotic and Control Engineering (IRCE), Singapore, 2019, pp. 46-50, doi: 10.1109/IRCE.2019.00017.

[3] Y. Li, B. Tian, Y. Yang and C. Li, ”Path planning of robot based on artificial potential field method,” 2022 IEEE 6th Information Technology and Mechatronics Engineering Conference (ITOEC), Chongqing, China, 2022, pp. 91-94, doi: 10.1109/ITOEC53115.2022.9734712. keywords: Mechatronics;Simulation;Conferences;Path planning;Information technology;Robots;Matlab;artificial potential field method;local minimum;safe distance;time distance detection method, keywords: component;Model Predictive Path Integral;Model Sampling;Robotics,

Appendix

A. GitHub Release

GitHub Release Link

B. Contributions

Zachary Serocki: Pure Pursuit Controller, wall avoidance, Localization, code architecture design, tuning and debugging
Sarah Listzwan: Frontier Identification, A* Distance from Wall, E-stop, path planning, tuning and debugging
Alexander Kraemling: Frontier prioritization, A*, pure pursuit, general debugging/ tuning, architecture design
Benjamin Penti: Localization, A*, debugging and tuning