WORCESTER POLYTECHNIC INSTITUTE
ROBOTICS ENGINEERING PROGRAM

Robot Navigation using Self Localization and Mapping (SLAM)

SUBMITTED BY

Jack Kamataris,

Diego Pena-Stein,

Kang Zhang

Date Submitted: May 1, 2024
Date Completed: April 30, 2024
Course Instructor: Dr. Rosendo

Lab Section: RBE 3002 D'24
Team Number: 8

Abstract

This lab report will cover how our team was able to build on our path planning algorithm lab and use it simultaneously with anonymous frontier detection from the point cloud map made from the lidar sensor. We will also explain how we are using g-mapping and AMCL localization practices to generate the map in real time and localize our robot in the generated map when passed statically.

Introduction

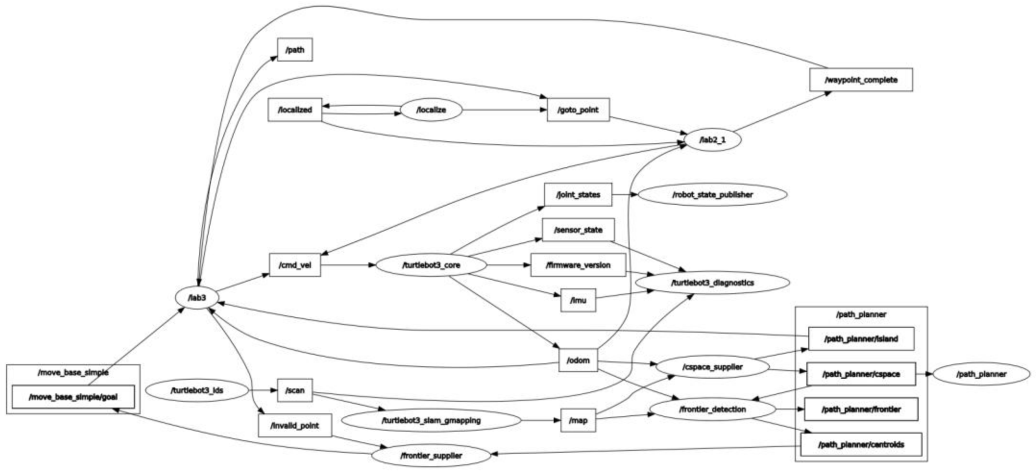
While we had working path generation and following from the previous labs, they were not up to completing the final demo. We significantly overhauled the driving, incorporating PID control, Bezier curved paths for generating waypoints, and aborting paths. These changes were made to account for continuous motion along an entire path so that our robot could traverse the map efficiently. While traversing the map, the robot would take multiple scans a second with the LIDAR sensor and create a dynamic point cloud map, which maps the walls and free space of the map. Using Gmapping and its internal SLAM algorithm, we could localize our robot while moving and staying still. Finally, using the AMCL algorithm, we could localize our robot from a static map and generate our location based on each particle's confidence in being in that position, given the current laser scans.

Methodology

1) PHASE 1: Create a map of the environment using Gmapping
   (a) Calculate the C-Space of the map
   (b) Find a frontier to explore
   (c) Plan the optimal path to the frontier
   (d) Navigate to the frontier while avoiding obstacles
   (e) Repeat until the map is completed
   (f) Save the map on file

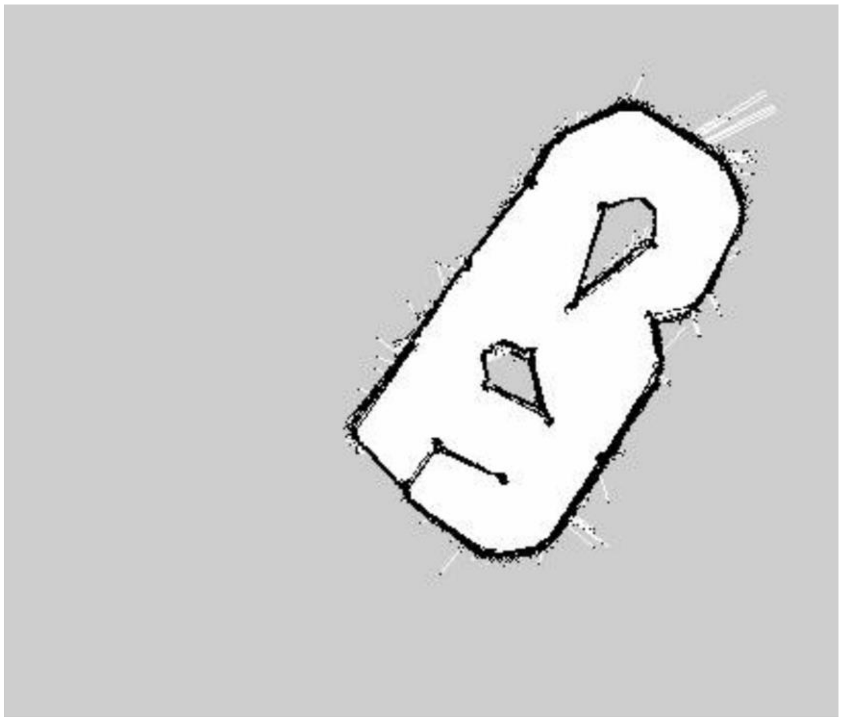
2) PHASE 2: Using the map, navigate back from where you are to the starting position while avoiding obstacles.

3) PHASE 3: Navigate to a specific goal position in the maze. The evil professor will choose the goal position



Results

Figure 1)



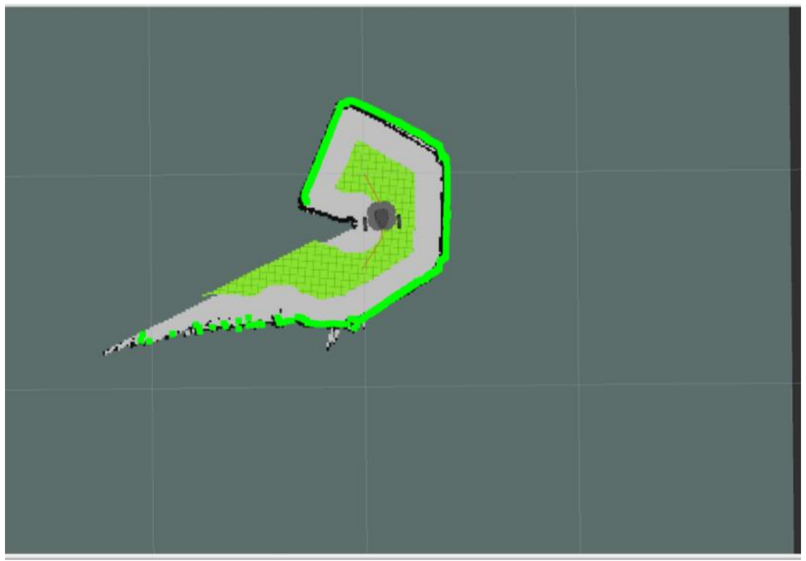


Figure 2) PGM image file generated by the Gmapping process and used for the AMCL static map.

Discussion

C-**Space**

Our team uses OpenCV to define how we can find, update, and locate the frontiers of the unexplored map. We know the frontier will be defined as a c-space that touches unexplored space. Otherwise, the robot would have already explored space and would not be considered a frontier. So we use OpenCV to edge detect the c-space colored pixels of a generated image that touch the pixels of the unexplored space. Then, with this, we blur the pixels to make the frontier larger and calculate its “image moment” and the centroid of the area. This centroid will give us the middle location on the c-space for our robot to drive to.

The dynamic map is published to a topic, as can be seen in Figure 2. The grey area is the unexplored area, white is the open space, and black is the walls. This is all you need to create the occupancy grid to generate the C-space of the robot with OpenCv. OpenCv allows us to create the C-space of the robot at a low cost by making a Gaussian blur of the walls (black pixels) and labeling all the remaining free space (white) as our walkable space. This allows our robot to update its C-space as well dynamically.

G-mapping

We use the provided G-mapping ROS library to run self localization and mapping while the robot is in the first phase. Even though the library works out of the box, we edited the parameters in the G-mapping YAML file in order to customize the behavior to our liking. The initial resolution was not as high as what we originally wanted it to be, so we increased it to half a centimeter. While our code was efficient enough to process the C-Space and A* algorithms with a resolution this high, we had issues with LIDAR artifacts at such a high resolution, so we decreased it to 1-2cm to lose scans being read through walls (This can be seen in Figure 1 as free space behind walls). In addition, we modified the SRR, SRT, STR, and STT parameters in order to prevent “drift” in G-mapping while driving, and some other update parameters to help the stability of localization.

Frontier Detection

For Phase 1, we are required to explore a map autonomously. This requires the robot to detect frontiers to explore and successfully map the environment. Our frontier algorithm used OpenCV to find where the c-space and unknown space border. We dilate c-space and unknown areas to record where they intersect to become our frontier. Additionally, to find a possible frontier point for the robot to reach, we dilate the frontier and find the intersection between the frontier and the c-space to guarantee that the end will be within the c-space. The centroid of the new frontier is calculated by finding the mean area point for the frontier. This sets the stage for finding a path to reach the frontier point.

Beizer Path Generation

For our path generation, we updated our Lab 3 path planning algorithm by reducing the number of stops and turns. Our team first generated the A* path and further optimized it by removing all intermediate points if a point can move to the farthest points without going through the c-space. Now that all the paths have been optimized to reduce the number of intermediate points, we used a smoothing algorithm to create smoothed pathing.

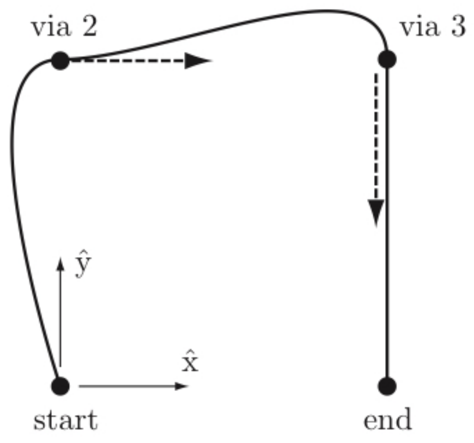


Figure X): Path generation uses the robot's pose to always point to the next point to travel.

Image Source: Lynch, Kevin M., and Frank Chongwoo Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017.

By changing the control points, we constantly manipulated the cubic Bezier curve to be tangent to the next point. This was done by calculating the 3rd control point to be colinear with the end and future points. To further manipulate how it curves, the length from the 3rd control point to the endpoint is changed to increase the turning radius of the curve to fit the robot's stable turning radius. A parametric piece-wise function is made for the robot to follow by generating each path for each point-to-point path.

PID-Controller

To move along the set points, a PID controller was made to control the robot's linear and twist speed, given its Euclidean distance and heading error to the end. This primary PID controller allows the robot to consistently correct its heading position to each point for smooth point-to-point driving. When the robot's heading error is greater than Pi/4 (90 degrees), the robot will spin in place, using a PID controller, until the heading direction is to the point. This removes giant swing turns to reduce the chances the robot will hit the wall. This sets our motion profile to move to the frontiers and navigate the map successfully.

Mid-Path Abortion

Since we separated the mapping, cspace calculations, and waypoint navigation into multiple ROS nodes, some asynchronous issues that won't happen in a single-threaded program can arise. For example, the robot may determine that a frontier at the edge of the cspace is valid, set a navigation goal, generate waypoints, and begin driving. A single threaded program would approach the navigation goal, update the frontier, and continue with no issue. However, our nodes take time to publish infrequently to the C-space, frontiers, navigation goal and waypoint topics, so the robot will attempt to drive to a nav goal that is no longer in the c-space. In order to fix this, the large number of waypoints we generate is constantly checked that the target waypoint is still valid, and the waypoint supplier and drive code continually check that both the waypoint is valid, and the waypoint the robot is driving to is still the one being published. If there is a mismatch, the robot will switch to the new waypoint. This allows frontiers to be generated on the fly, and the robot to abort a path if it no longer exists inside the cspace.

AMCL

For phase 3, we used AMCL (Adaptive Monte Carlo Localization) to determine the robot's position when placed in an unknown position. For AMCL to work, the robot must know the “map” of where the robot is placed because AMCL uses a known environment to determine the likelihood that the robot's position is a pose. To use AMCL, we used ROS’ AMCL package to localize the robot’s position by obtaining the robot’s base_scan (lidar) and odometry. By creating randomized poses across the map, AMCL will determine the probability the robot is at each point given the lidar readings. If the readings are not similar, then the pose is unlikely, and a negative bias is created for that pose. Inversely, if the pose readings are identical, then it makes a positive bias for that pose. Given a couple hundred interactions, the robot will localize to a position on the field from these biases.

Our team tuned AMCL by changing the number of particles, the change of rotation and translation, the error from rotation and translation, the initial pose guess, and the particle spread. Increasing the particle count increases the chance that the robot's pose will localize, as it will begin localizing to that pose. However, increasing the points will increase computational time and take longer to localize. Too few particles can lead the model to localize to an incorrect pose. We also increased the spread of the particles from the initial guess and assumed that the robot's yaw was unknown (-pi to pi error). The spread was set to encompass the entire field to localize all positions on the map.

Since AMCL requires the robot to move to update its particles, our team implemented a spinning algorithm that caused the robot to spin in place until the variance of x, y, and heading localized to a specific range. Since we do not have a local cost map planner to navigate through the wall in an unknown position, spinning in place is the safest and easiest way to localize the robot.

In order to detect any disturbances in the robot’s AMCL localization, namely the evil professor moving the robot mid-path, we listen to the internal accelerometer of the robot, and if the robot experiences an acceleration in the vertical direction, it will reset the AMCL algorithm to attempt to re-localize.

Conclusion

By building off what we learned in class, made in previous labs, and using the ROS resources available, we were able to complete the lab very well. Our robot completed the task on the most difficult map in roughly 5 minutes, which is half the allotted time, with minimal collisions. Some improvements we might have made given more time or harder constraints are cleaning up the code and making our algorithms more efficient. One of our biggest bottlenecks is processing map data, which is massively improved by using OpenCV, but we still have some methods in Python from the previous labs that are single threaded and quite slow. With faster mapping, cspace generation, and frontier detection, our robot would be able to move much faster. Another issue we ran into was the robot getting very close to walls, and as a result had to constrain the C-space quite a bit. With sensors or a map able to support higher resolution and a better SLAM library, we would be able to navigate more difficult maps will more room for error. Despite these difficulties, we were able to program our robot to execute the task in a reasonable amount of time in a robust and repeatable manner.