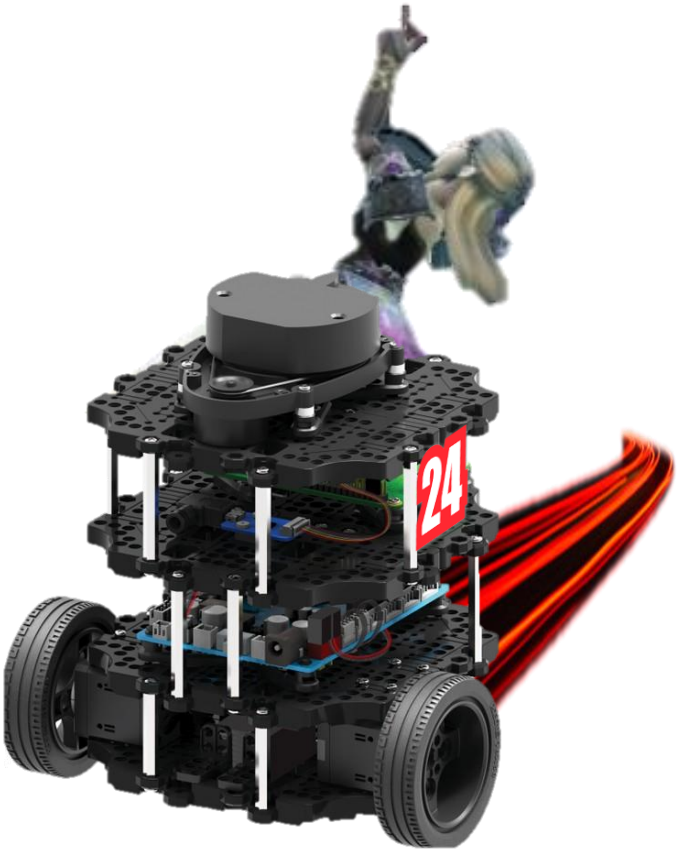
RBE 3002 Final Project



Team 24

Jakub Jandus, Samuel Markwick, and Kevin Siegall

Introduction

Mapping and localization are two very important (and difficult) problems in robotics. We made it look easy because we're just that good, but in general it's a very common problem that any autonomous robot needs to handle, especially if it will be operating in unknown spaces. It's essential that any robot aiming to effectively perform in a new environment be capable of mapping and localizing in real time as it drives.

Methodology

Our approach to the mapping problem was to use the gmapping node built into the turtlebot_slam ros project. Gmapping will automatically process our LIDAR data and give us an occupancy grid which we can use to figure out where to go next. Frontier detection is done through a single scan through our map data. We use OpenCV for padding the walls to mark out a cspace, and also to create a gradient distance from the walls, which we use the square of a difference for the cost function for both Dijkstra and A*. This encourages any generated paths to hug the center of the available space, which aggresively helps our robot avoid collisions with the wall and allows us to safely drive without cspace if our robot starts in the cspace. We also restricted the max map dimensions to [-3.5 m, 3.5 m] in both dimensions, and increased the resolution of the measured map to 0.015 m. This editing of the configuration .yaml files was very important and to our surprise not mentioned anywhere in the lab documentation.

For all walkable nodes in our map (defined as nodes where the value is <50 and != -1), we check if it has an unknown (-1) neighbor, and if it does, we flag it as a frontier. We then turn that frontier list into a bit array, which gets passed to OpenCV for segmentation and centroid location finding. We then run Dijkstra's algorithm to all frontiers to approximate what the shortest path to that node from the robot would be and sort all frontiers using a heuristic that balances the distance from the robot to the frontier and the size of the frontier (which is obtained by OpenCV as well). The robot then grabs the first node in the sorted list of frontiers, calculates A* to get the shortest path there, and then executes the path using Pure Pursuit. Once Pure Pursuit finishes or otherwise exits (during mapping this happens if the robot travels 120 nodes without finishing, or if the robot ever strays too far from the intended path), the robot will stop and recalculate everything given the newly obtained map data, choose a new frontier/goal, and repeat.

To check when the robot is out of explorable frontiers, the robot runs through the regular procedure for locating frontiers. If the number of detected nodes either in the OpenCV step or after running Dijkstra is zero, the robot begins Phase 2 and attempts to return home. When the robot starts, it saves its initial position. Once mapping is complete, it just runs A* to path find back to its initial position and then path follows once again using Pure Pursuit. Upon returning to the home position, the robot saves the newfound map to a file.

To solve the 'kidnapping problem', we utilize ROS's AMCL (Augmented Monte Carlo Localization) node. AMCL gives us a PoseStampedWithCovariance on the /amcl_pose topic, which we listen to instead of the /odom topic as long as a flag is set in the launch file (more on that later).

We start localization by calling the *global_localization* service call, which spreads particles randomly across the entire map. We noticed that the timing for Phase 3 doesn't start until the robot starts moving. As such, instead of spending a lot of time turning to localize, we can call the *request_nomotion_update* service call to AMCL to localize in place. Calling that continuously for 10 seconds localizes the robot decently well, and then we perform one or two last turns to make sure that our covariance is low enough to be confident in our position.

Pure Pursuit is a path following algorithm based on the idea of using a lookahead distance, and preempting changes in the path's trajectory as a result. We had initially started with a distance-based lookahead, but eventually swapped to an index-based lookahead to reduce the amount the robot would cut turns. As such, our methodology is just to find the closest node on the path to the robot's position, grab its index, and then add a static amount to it, in our case 12 cells (0.18 m). If the robot ever deviated from the closest node farther than 0.18 m, pure pursuit would emergency exit and ask the path to recalculate its trajectory given the new current position.

We used a modular launch file for this project. When we were continuously testing and iterating on a single file, we could disable the running of our scripts so that we could run them individually in their own windows. This allows us to see the console output easier, as well as close and restart just those nodes without needing to kill and restart rviz and gazebo. We also had a configurable variable for swapping between phase 1/2 (gmapping) and phase 3 (amcl). Within the gmapping option, we copied and edited the turtlebot3_slam launch file in order to launch rviz with our own config file instead of with theirs, which didn't have any of our visualization listeners.

In Rviz, we had many listeners set up to visually display information that was helpful for debugging, such as A*/Dijkstra wavefronts, the cells and centroids of our frontiers, our cspace with padding, and our path, all in addition to the information already displayed by gmapping/amcl, which is just the robot's position, the laserscan data, and the map. Using this allows us to quickly iterate and run our code without needing to type a ton extra.

Results

OpenCV works amazingly for dilation and our gradient-space. Our A* path planner went through many iterations, as both the heuristic and node weights were extended by penalty for turning and gradually being farther from a wall. This made the calculations heavier, but after tuning generated nice paths in the middle between two walls, which helped avoid accidental scratches by a large margin.

As of the writing of this paper, we are the fastest team this term, by about 10 seconds. We spent 75.74 seconds on phases 1 and 2, and 36.89 seconds on phase 3, totaling 112.64 seconds.

For frontier detection, strictly asking OpenCV for the centroid of the frontier is a little problematic, at least when we have irregularly shaped frontiers (V-shaped frontiers tend to be common when seeing past an obstacle from two different sides). In those cases, the centroid appears in the middle of the unknown region, and we need to run an extra adjustment loop to get it back into walkable territory. We've

additionally reduced the necessity for this by increasing the tolerance for our path following, thus that if a frontier we were driving to turned out to be a wall, we would not wind up stuck in cspace.

While coding, we entered a point where recalculating A* took about 5 seconds per iteration, which was slowing us down significantly. Sitting in rviz, watching the searched-through space grow in pretty blue color looked oddly satisfying, before we realized that we are publishing *every* iteration's occupancy grid message of the A* algorithm to rviz. This produced a ton of data traffic and hogged CPU power. After commenting out the message constructing lines, the runtime increased by 2000% to about 0.25s, which was much more acceptable.

When we started working with Pure Pursuit, we were concerned that it would not follow the path tightly enough. This had prompted us to begin working on a Stanley Controller, which has an extra term for reducing the perpendicular distance to the path. This didn't wind up being necessary, as pp was working well enough once we moved to the index-based lookahead.

Discussion

If the goal of this lab was speed, we definitely achieved that. Under 2 minutes for all three phases! Still, we were not out of space to optimize. Our A* implementation still uses a dictionary to associate nodes with their parents and their costs. It's also poorly tuned and performed similarly to Dijkstra for certain cases, especially when goals are close to walls.

We also still wait a full two seconds to ensure the newest map update before we calculate anything, *every time* we ask it to calculate the path. If there was a need for us to speed our robot up more, we could have it fairly easily. We approximate that we could have cut almost 20 seconds off our time with relative ease.

In the end, we were effectively able to perform mapping and localization. To accomplish all this, we learned how to operate in the ROS environment, write ROS nodes, and compile custom launch files.

Conclusion

In this lab we learned how to implement SLAM and localization on a ROS based robot. For SLAM we implemented the gmapping package to perform LiDAR SLAM mapping, to map out the field using openCV to find frontiers and calculate drivable space and A* and pure pursuit to navigate around it. For Localization we implemented and tuned the amcl package to be able to allow our robot to navigate around the maze after being "kidnapped".

In the realm of RBE, some say it's tough,
But we take those dubs, call their bluff.
Challenges come, we face the grind,
RBE's not hard; it's a path we find.

In lectures and labs, our team is supreme,
We master the puzzles, living the dream.
They call it a myth, but we're here to say,
On the demo field, we'll pave our own way.

References

ROS gmapping package: http://wiki.ros.org/gmapping

ROS AMCL package: http://wiki.ros.org/amcl

Code final release: https://github.com/RBE300X-Lab/RBE3002_B24_Team24/releases/tag/final

Contributions

Realistically, we all touched and tuned around the code so much that it is hard to point out contributions of who did this specific part. On top of that git blame is hard to use since we often would make changes on each other's (specifically Sam's) laptops while working on the robot.