Pathify - Investigation into Real Time Path Planning and Optimization

Jong Seo Yoon, Timi Dayo-Kayode, Michael Edegware

1. Abstract

Robot Operating System(ROS) provides a set of software libraries to build a robot application and simulate it in a virtual space without the need to test on the real world. ROS also provides a tool to start navigating around the map, assuming the the map is already known, using a standard path planning algorithm to get from the current point of the robot to a destination, assuming the path has already been explored. However, without the map being known in advance, the robot runs into difficulty when navigating around the map and must explore the map in advance before it is able to navigate to an object. On the other hand, the camera attached to the robot can detect images regardless of whether the path to the object is known or not to the robot. In this project, we will investigate a way to autonomously explore an unknown space and map different points of the world via image detection algorithm using OpenCV and also provide an interactive session between the robot and the user.

2. Introduction

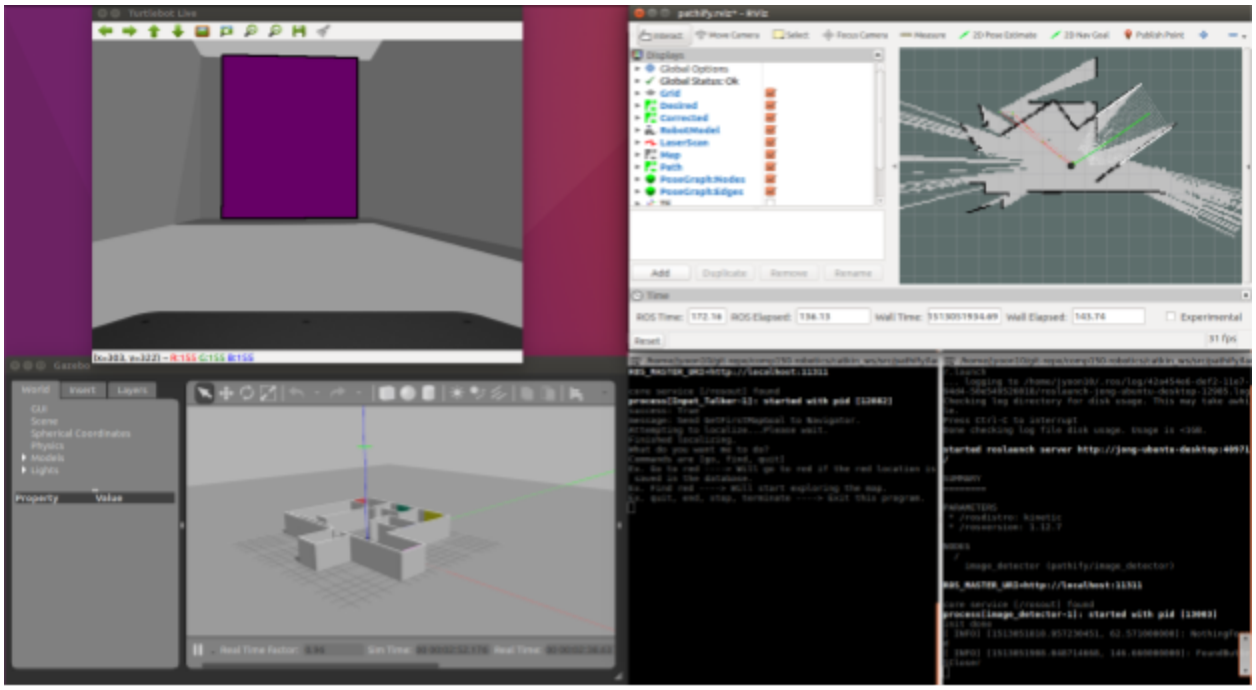

Fig 1: The Path Finding, I/O Interface, Database, and Image Detection modules working hand-in-hand.

This project introduces a way to autonomously navigate around the map and save meaningful locations to the database.

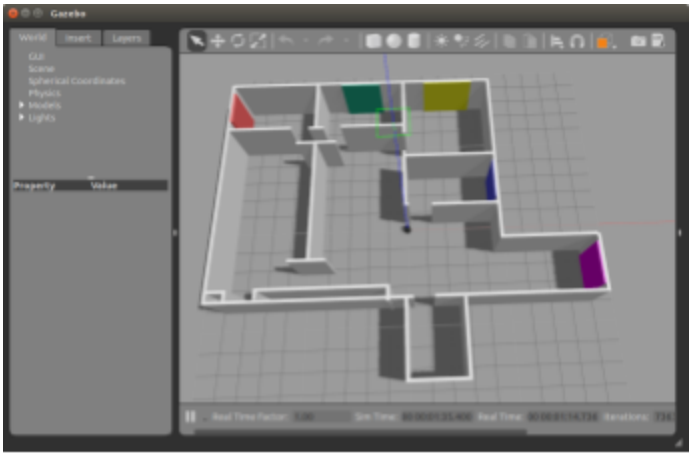

Fig 2: One of the Gazebo maps that we created for our project.

In order to make the problem more simple, we created a 3D map on gazebo which contained either a fixed colored wall, or a colored wall. Gazebo Maps are also designed so that the turtlebot will have a home location, and we set it up such that no colored walls were visible from the home location. To try to ease the autonomous navigation method, we created the map with edges so that all points around the map will be unique no matter where the turtlebot scans, and meaningful locations are detected by the image detection algorithm which takes turtlebot's video stream as an input and detect anything that is not part of the wall.

ROS provides many tools in order to facilitate the development process. ROS enables us to build robot applications by providing the developer with a set of tools and libraries to simplify the task for building complex robot behavior. For 3D maps, we used Gazebo to build three customized maps to test our project on. By simulating a turtlebot on a 3D map on Gazebo, we were able to visualize the turtlebot's actual location as it moved around.

For path planning, we used a tool called RViz which is implemented in ROS. RViz is a visualization tool that generates a veryuseful, live representation of the data for all connected nodes which are running on ROS. In our project, RViz generates a visual representation of the data coming from the sensors of the turtlebot as it navigates across the 3D map in Gazebo. RViz enables the developer to not only visualize the path planning trajectory of the turtlebot, but it also shows different parameters for every node that is connected to the main

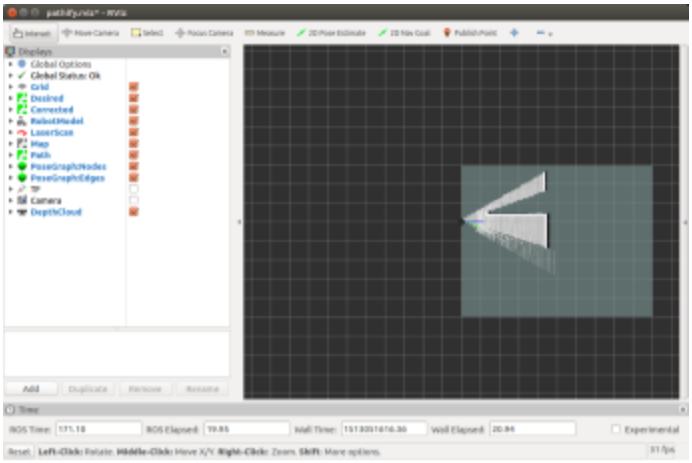

Fig 3: RViz application for visualizing ROS simulation.

ROS program. RViz allows the developer to visualize how most of the nodes, which supports RViz, show up for continuous tweaking as the project matures.

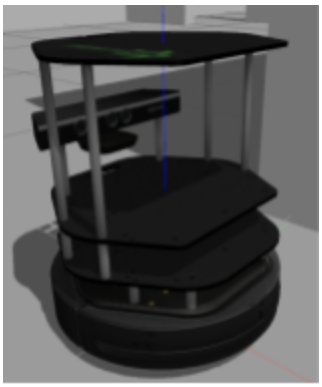

Fig 4: The turtlebot used in our simulations.

The turtlebot we used for simulation is equipped with kinect sensor which allows laser scanning as well as depth of field from the image it outputs. The laser scanner allows the turtlebot to draw a 2D map using nav2d as it explores around the 3D map. We used turtlebot for simulation so that if the simulation worked, we will be able to test it on the real turtlebot as well.

3. Approach

3.1. I/O Interface

The robot offers an interactive input and output interface between it and the user. This interface allows the user to give a **recognizable** input keyboard command, at a time, to the robot and the robot would reacts accordingly. A recognizable input command is any command that the robot has been taught and can interpret. For this project, the robot has been taught how to identify three types of command. They include a single letter command; “y”, which represents an affirmation; “n” which represents a denial; “XXX” which can be a name of a location; a

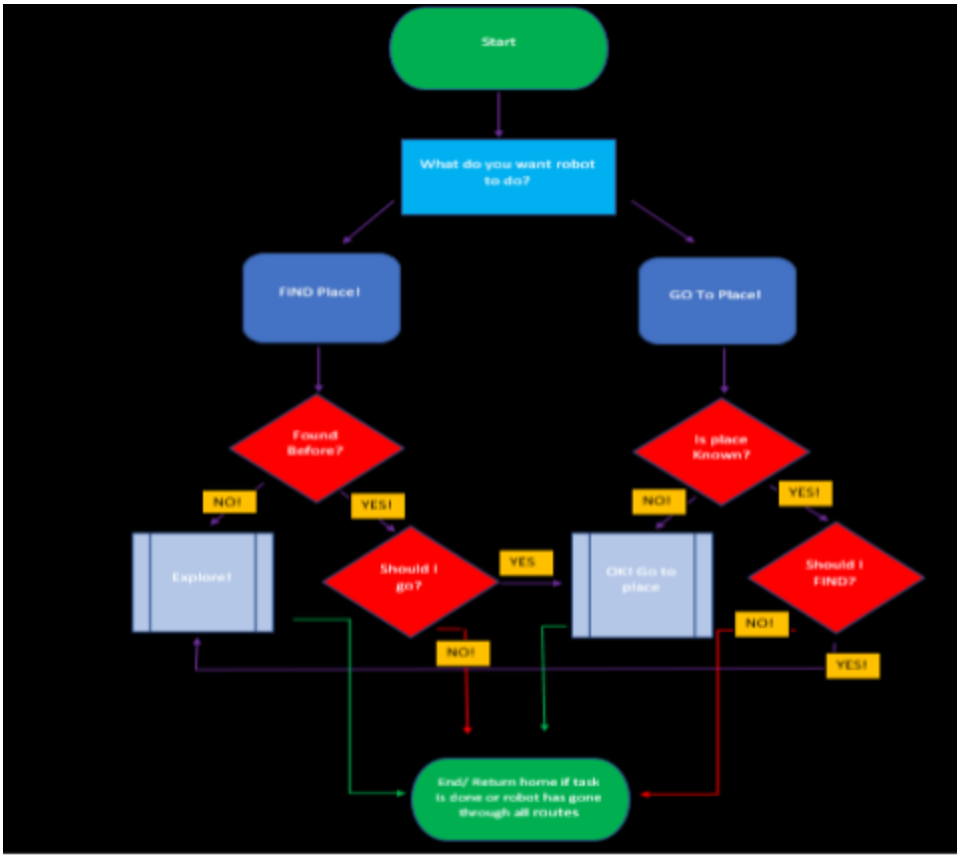

Fig 5: A logic chart of how the database and I/O interface interact.

double word command that must be in the form “find XXX” where XXX is a name of a place in the map; and a three word command which must be in the form “go to XXX” where XXX is same as in above. The letter case of the input is not important. The output is via the monitor screen. The outputs are usually straight-forward questions requiring the user to further clarify what action the robot is required to take. There are two input receiving modes. These include:

1. Rest mode: In this mode the robot is either at a starting point or has just completed a task. There’s no learning taking place in this mode. The robot prompts the user on what action to take(e.g “What do you want me to do?”). After receiving an input, the input is sent to the Arithmetic and Logic Unit(ALU) to execute the command.
2. Operation mode: In this mode, the robot is in the process of completing a task. The robot consistently prompts the user whenever it sees something it cannot recognize. For instance if the robot was asked to “find RED” and it sees an unknown place, it ask the user, “is this RED?” if it gets a denial(“n”), it further prompts the user, “so what is this place?” then send the response to the ALU where memory mapping process will be initiated.

3.2. In-memory Database

The in-memory database is the location where all the things learned by the robot are saved. These things include the robot’s interpretations of the input it gets from the camera, mapped to the user’s understandable language. For instance, while a human knows a place as RED, the robot only knows same place in terms of coordinates values in respect to the whole map it is exploring and rgb values of the camera images of the place RED. The robot’s in-memory database has two parts. They include:

1. The Read Only Memory: This memory is represented by a .txt file and is first loaded to the program when the robot starts. The file composes of a series of lines where each line represents the full details of a particular location parsed into a string value. The details include a 3D coordinate system value, location name, and map name. The ROM gets updated when the robot finds a new location.

2. The Random Access Memory(or cache memory): This memory is represented by a vector/array of structures, where each structure represents the decompressed items in each line of the ROM’s .txt file. It gets updated from the ROM when the robot is powered on. It interacts with other parts of the robot and it is the only

component of the robot that interacts with the ROM. When the robot is powered off, the RAM ensures the ROM is updated and erases itself.

3.3. Arithmetic and Logic Unit

This is the room for decision making. It gets instructions from the I/O module and also listens to the robot's camera input. It reads these instructions and understands the nature of these instructions by looking for flag values. *(" Should I shut down? Am I to GO? Have I found this place before? Will the user want me to FIND place instead?)"* . It determines the flag values by searching for the occurrence of the instruction's representation in the robot's RAM. If the representation is not in the RAM, it updates the RAM from the ROM and searches the RAM again. Then it finally initiates the necessary operation by virtue of the nature of the instruction. It should be noted that if the instruction is logically unnecessary(for instance, asking the robot to find an already found location), the logic unit suggests a better and similar instruction to the output unit (e.g, "should I go instead?") and listens to the Input unit for confirmation.

3.4. Path Finding; NAV 2D

For pathfinding, our main aim was for the robot to autonomously navigate around a 3D environment in gazebo without having any previous knowledge of the map. The robot would then create a 2D representation of the map in the RVIZ program as it navigates around and learns more about the environment.

Our robot of choice; the Turtlebot, would begin at its home, preset with x, y, and z coordinates and would remain stationary while awaiting input through the I/O interface. Once it receives a command to go search for a certain thing -a color in our case- it would begin navigating the map and searching for said color. The turtlebot would be able to detect when it saw a part of the wall that differed in color from the regular gray of the wall and then stop at that location. It would then ask if that color was the color being searched for. In the event that it found the color being searched for, it would then save the color and its x, y, z coordinates of its location as a key-value pair in its database for later usage if necessary. In the event that the color found was not the color being searched for, it would ask what color that was, and save the response and the x,y,z coordinates of that location as key-value pairs in its database for potential retrieval in response to future queries and then continue along its search for the initially requested color search.

Our implementation of the autonomous navigation of the relatively unknown map used ‘Simultaneous Localization And Mapping’; SLAM which allowed the turtlebot to construct and update its 2D representation of the 3D gazebo world in RVIZ while simultaneously keeping track of its location within the environment. By using SLAM, the turtlebot would be able to use the environment to update its position by using landmarks as a way to mark sections of the map and correct for unforeseeable and likely errors in odometry reports.

This method of navigation for this project was designed to mimic a human agents’ interaction with its environment as much as possible. A human agent would navigate its surroundings and make mental notes about significant things it sees; typically things that ‘stand out’ and use these as ‘navigational nodes’ that it could use to find itself and use as a point of inference when trying to find its way home. A typical human agent would recall things that it had found and know not to explore already known places if it had to find something it hadn’t seen before, and know the exact location of something it had to find if it had seen it before.

Our implementation of this environmental interaction, however, wasn’t a complete representation of how a human agent would interact with our generated environment due to obvious limitations of the turtlebot. In our implementation, the turtlebot did not use landmarks or noteworthy objects it had found along its venture out into the maze as points of inference to navigate its way home. Instead we gave our turtlebot a home, and the x, y, z coordinates of this location, and it simply navigated its way back to this location when it was done performing the task for which it ventured out into the environment. Our turtlebot however noticed things in the environment as it navigated regardless of whether they were the things it set out to find, and would be able to save time when asked to find those things by simply navigating to those locations rather than performing a new search just like a human agent walking around in an unknown environment could notice things and recall them when necessary.

3.5. Image Detection

In order for the turtlebot to capture different colors as it moves around the map, we channeled a stream of images from the turtlebot’s kinect and camera sensor to the image detection algorithm we will describe here. The basic idea behind the image detection algorithm is that as the turtlebot moves around the map, the image filter detects any color other than the wall and tells our command prompt I/O interface when it detects a color. When it does detect the color, the image detection algorithm also tells the command prompt I/O interface whether the color is close or far from the turtlebot’s

current location. The distance from the detected color is done by calculating the area of the detected color space. For example, if the detected color is far away, the image detection algorithm will output “DetectedButGetCloser”. When the turtlebot has reached a position where the detected color is very close, the image detection algorithm outputs “Stop”, which will be sent to command prompt I/O interface for further processing.

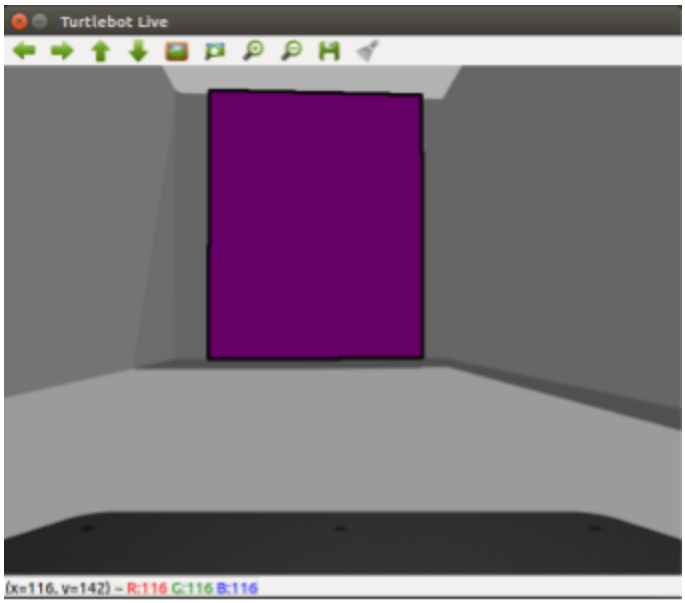

Fig 6: Image Detector program showing image output after filtering colors.

With the turtlebot’s ability to output stream of images from the camera, we used the cv_bridge method to convert ROS’s camera output to OpenCV’s image matrix, and we applied filters for color detection. In order to detect colors, we used “inrange” method which detects all colors in range except for the color of the wall. We intentionally set the wall to be of the same range of color, so that the image detection algorithm would not detect walls. With the color detected, we draw a contour around the detected color and show in real time through a new video output which combines the real world with the filtered image. In order to improve detection, we used the gaussian blur algorithm on the detected color to remove anti-aliasing as much as possible, but the effectiveness of the blurriness seemed to be reliant upon the hardware specification of the project we run on.

4. Experiments / Results

We set all our experiments in the ROS simulation. However, due to the Nav_2D package which handled our autonomous map navigation being faulty, we were unfortunately unable to generate any definitive results from the experiment. For instance, Nav_2D would stop autonomous exploration even though it hasn’t explored all spaces

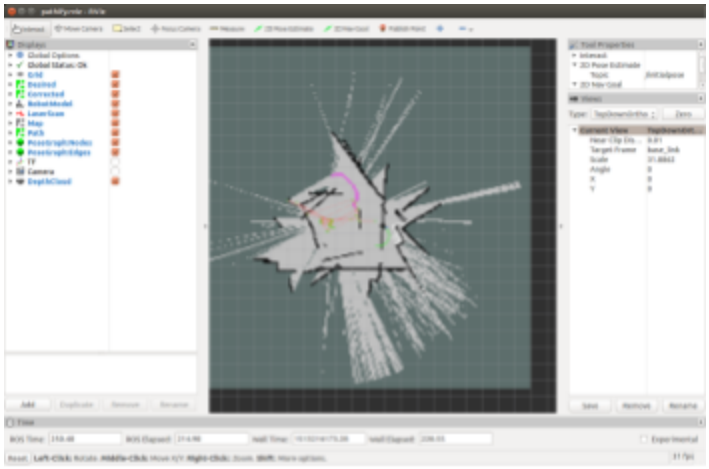
Fig 7. Example of gmapping incorrectly mapping the space.

on the map, as well as due to gmapping leaving a lot of residue from the scan, a space which has been explored before could not be reached the second time around.

We attempted adjusting the velocity of the turtlebot and tried out the Nav_2D package on different maps amongst other things in an attempt to solve the issue we were having with the package through our hypotheses on what the issues affecting the turtlebot's apparent inability to maintain a singular orientation were, all to no avail.

5. Results

While we were successful at implementing command prompt I/O interface and image detection algorithm, we were unable to generate any conclusive results due to the path planning issues we faced while experimenting.

Barring the issues with our path planning module, our plan to show results was to show a graph of the time the turtlebot took to find a color **for the first time** vs **after previously finding it** to show how it had mimicked human memory and remembrance by recollecting the location of an object in its exploratory environment and being able to navigate to it a lot of faster as it knew where the object was.

6. Conclusion / Future Works

In conclusion, we were able to create a package that threaded together all the separate modules of our project; the I/O interface, Image Detection, and Path Planning. These modules all interact with each other for the purpose of our project. We were able to successfully implement all of the modules except the path planning module which we couldn't quite get to work perfectly because we had issues with the turtlebot reorienting itself and tried methods such as reducing adjusting its speed, and having an asymmetrical map so that it could use numerous points as 'landmarks of reference' to maintain its orientation all to no avail.

To further develop on our package, we would make the autonomous maze navigation smarter such that the turtlebot would be able to navigate the unknown maze in the most optimal way possible and reduce the time spent navigating unknown parts of the maze. We would also give the package the functionality to enable it to accept known items as parameters for a search using keywords such as 'near' for example. A user performing a search could specify that the object is near some known object, and so the turtlebot would only search a given radius around the known object.

7. References

1. Hwang, Kao-Shing, Chih-Wen Li, and Wei-Cheng Jiang. "Adaptive exploration strategies for reinforcement learning." In *System Science and Engineering (ICSSE), 2017 International Conference on*, pp. 16-19. IEEE, 2017.
2. Mirowski, Piotr, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andy Ballard, Andrea Banino, Misha Denil et al. "Learning to navigate in complex environments." *arXiv preprint arXiv:1611.03673* (2016).
3. Smart, Paul R., and Katia Sycara. "Place Recognition and Topological Map Learning in a Virtual Cognitive Robot." In *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, p. 3. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.
4. Meikle, Stuart, and Rob Yates. "Computer vision algorithms for autonomous mobile robot map building and path planning." In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, vol. 3, pp. 292-301. IEEE, 1998.
5. Zhu, Weiyu, and Stephen Levinson. "Vision-based reinforcement learning for robot navigation." In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, vol. 2, pp. 1025-1030. IEEE, 2001.
6. Se, Stephen, David Lowe, and Jim Little. "Vision-based mobile robot localization and mapping using scale-invariant features." In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 2, pp. 2051-2058. IEEE, 2001.
7. Jan, Gene Eu, Ki Yin Chang, and Ian Parberry. "Optimal path planning for mobile robot navigation." *IEEE/ASME transactions on mechatronics* 13, no. 4 (2008): 451-460.
8. Center, Artificial Intelligence. "SHAKEY THE ROBOT." (1984).
9. Quigley, Morgan, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. "ROS: an open-source Robot Operating System." In *ICRA workshop on open source software*, vol. 3, no. 3.2, p. 5. 2009.