Find Object in an Unknown Environment for Robot using ROS

Tushar Anil Mittal

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

September 16, 2022

Abstract

Autonomous robots are becoming commonplace in daily life and in several modern industries to improve personal safety, quality of life, and business efficiency. In this project, we work towards developing a fully autonomous robot with the capabilities of finding objects in unknown new environments using ROS (Robot Operating System) and the Navigation Stack it offers. The robot is able to explore, map, avoid obstacles and plan a path to reach the object without the help of human input. The findings from this paper can be used in developing search and rescue robots, warehouse robots and others.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Tushar Anil Mittal Signature: Tushar Anil Mittal

Acknowledgements

I would like to thank Dr. Nicolas Pugeault for all his support and guidance throughout the dissertation. I would also like to thank my friends and family for their encouragement during this time.

Contents

Chapter 1: Introduction

As technology improves and the cost of robots decreases, indoor mobile robots have become commonplace in everyday life, such as robot vacuum cleaners, restaurant service robots, and exploration robots. Robots are also being used by the military and search and rescue teams to find people or objects to reduce personnel casualties and reach areas that humans cannot reach. Most of these robots are teleoperated and require a person to control them, reducing the number of robots that can be deployed, as teleoperating robots require trained personnel. In this project, we look at developing a completely autonomous robot that can find objects and requires no human control.

Developing an autonomous navigation system requires solving problems in robot localization, mapping, collision avoidance and path planning, which requires a long time and research to develop from scratch. Therefore, we use the open-source toolkit called ROS (Robot Operating System) [18] for this project. There are currently two versions of ROS; ROS2 is a successor of ROS1 but is still in the early stages and has a smaller community; therefore, we will work with ROS1, referred to as ROS, in this paper.

ROS helps develop robotic projects by allowing the developer with functionalities such as low-level device control, message-passing between processes, package management and more. It has an extensive collection of open-source packages implementing state-of-the-art algorithms related to autonomous navigation. ROS provides a publish-subscribe messaging infrastructure designed to support the quick and easy construction of distributed computing systems. It is designed to be a combination of nodes; each is compiled individually to complete one task. The topic is used as a tunnel to transport messages between nodes. It supports various 2D and 3D simulators for robot simulation. For visualizing sensor data, robot models and the environment from robots' perception of the world, it uses RViz. ROS supports C++ and Python, and communication between a Python node and a C++ node can be easily implemented, which means we can develop with different programming languages, integrate many reusable parts into our system, and improve co-working quality.

1.1 Aims and Objectives

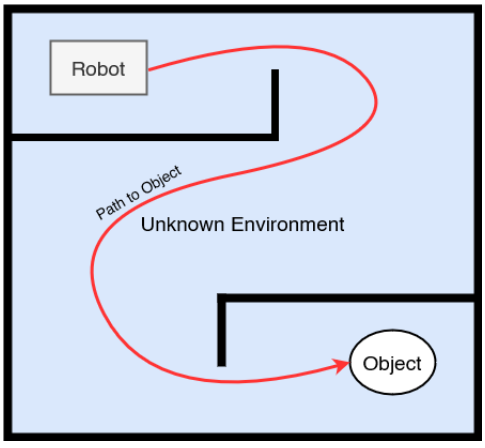


Figure 1.1: An illustration of the Aim of the Project.

This project aims to develop an autonomous robot that can find a given object in an unknown environment while avoiding obstacles along the way, example shows in Fig. 1.1. This is a challenging task as the robot is unknown of the environment therefore path planning while avoiding obstacles is complicated, also change in lighting conditions can reduce the robots capability to recognize an object. The robot in this project is equipped with low-cost sensors and therefore, the information gathered can be useful in developing low cost robots for everyday use and security scenarios such as military, search and rescue teams or finding lost items in warehouses.

The goal is achieved by combining common research topics such as SLAM(Simultaneous Localization and Mapping), object avoidance, path planning, autonomous exploration and object detection. We also look at and compare some of the ROS packages developed to solve these problems. The project is developed, and experiments are done in a 3D simulator.

The project objectives are broken down into steps:

- Configure a Robot with a depth sensor and distance sensors.
- Mapping and Localization using SLAM.
- Configure path planning and obstacle avoidance.
- Autonomous exploration to learn and map the environment.
- Find an object using object recognition and navigate to it.

1.2 Report Structure

The report contains a total of 5 chapters. Chapter 1 introduces the project and discusses the aims and objectives. Chapter 2 discusses the background survey, Chapter 3 explains the implementation of the project, Chapter 4 discusses the tests conducted to evaluate the solution and the results. Chapter 5 is the conclusion, limitations, and suggestions for future work.

Chapter 2: Background

Following the context introduced in the first chapter, this chapter discusses the systems developed by the industry that already exist and various ROS packages, algorithms involved and published works related to robot navigation.

2.1 Related Works

2.1.1 DARPA (Defense Advanced Research Projects Agency) Challenges

DARPA is the most prominent research organization of the United States Department of Defense. It organizes the **DARPA Grand Challenge** [22] for creating autonomous vehicles in different scenarios. Over the years, DARPA has organized challenges related to diverse environments and goals, such as the **Urban challenge** took place in 2007, where the autonomous cars had to follow a pre-defined path while following the traffic rules and avoiding traffic and obstacles. In 2017 DARPA started a 5-year Subterranean Challenge [5], where the task was to build robots that can autonomously map, navigate and search environments which lack lighting, low GPS signals and other challenging scenarios. The challenge was conducted both virtually and in the physical environment.

The team CERBERUS won the Subterranean challenge [20], the autonomous exploration and searching worked by collaboratively exploring and path finding using aerial and ground robots. The robots had multimodal perception capabilities and mapping autonomy, and communicated over the network with each other.

All robots were equipped with LiDARs, as well as visual or thermal cameras, the information gathered from all the robots helped collectively map and localize in environments which were constructed with visual limiting scenarios. The collective system implemented a unified exploration and path planning strategy. All robots had different mission roles, like the aerial robots were used for quickly mapping the environment allowing the ground robots to plan safe path.

The DARPA grand challenge is an excellent source for new emerging technologies related to autonomous mobile robots.

2.2 ROS Packages

2.2.1 ROS Navigation Stack

ROS Navigation Stack [15] is a collection of open-source packages that are used to navigate a robot from a starting location to a goal location while avoiding obstacles. It is Navigation starting place for students, robotics enthusiasts, and sometimes companies. The Navigation Stack is made of modules containing SLAM implementations, localization, and global and local planning algorithms. These algorithms or technologies can be swapped based on the sensors on the robot, the environment of the agent and its desired behaviour. All components offer configuration parameters that must be adjusted depending on the desired behaviour and the type of robot.

Limitations

The Navigation Stack has been designed as a general-purpose tool but has some hardware limitations:

- It was developed for only differential drive and holonomic wheeled robots. It assumes that the robots are controlled by transmitting velocity commands in the form of x velocity, y velocity, and theta velocity.
- It was designed for square-shaped robots but also works with circular robots. Although, making it work for other arbitrary shapes is complicated and requires complex algorithm changes.
- It only supports a static global map, so any changes made to the environment will not reflect on the map and therefore need to create a new map if the global environment changes.
- Works best in planar environments; environments with high elevation changes can interrupt the path planning process.

2.2.2 SLAM

One of the problems in navigation is creating a map of an unknown environment that the robot can use for path planning. This problem is solved using SLAM (Simultaneous Localization and Mapping) algorithms, which create a map while keeping track of the robot's localization within the map. With the increased use of ROS in the robotics industry has originated several SLAM algorithms which can be deployed on robots compatible with ROS. The choice of the algorithm depends on the sensor type used for mapping and the agent's environment. Gmapping [7] and Hector SLAM [10] are laser-based Occupancy Grid Mapping algorithms. In contrast, ORB SLAM [16] is for monocular RGB cameras, whereas RTAB-Map [13] is meant for RGB-D cameras but can also work with other sensors.

Due to the low cost of RGB-D sensors such as Microsoft Kinect, we use a simulated Kinect for mapping in this project so that this project can easily be deployed on a Robot. Gmapping and Hector SLAM, although made for laser scanners such as Lidar, can be used for Kinect by converting the depth information to simulate laser, but due to the short range of the depth sensor and also slow acquisition range, the quality of the map generated using the two SLAM methods can vary compared to one made using Lidar. Therefore, we must choose a SLAM algorithm that works best with the RGB-D sensors.

The research [4] evaluates ROS compatible SLAM algorithms for RGB-D sensors. Their research compared Gmapping, Hector SLAM, ORB SLAM, ORB SLAM 2 and RTAB-Map on a varying-sized environment 3D dataset, trajectories and other parameters. The authors found that Gmapping worked accurately and is good for 2D mapping on computationally limited robots, whereas RTAB-Map is state of the art for 3D SLAM but requires more processing power. This research is good but limited because of the fact that it was done on pre-recorded processed datasets and not in a live environment. Also, the RTAB-Map ROS package can be used to generate 2D occupancy grid maps similar to Gmapping, resulting in use of low processing power and there is no metric to compare the map to the ground truth. Although the research has its limitations, we learn that Gmapping and RTAB-Map are the best methods; therefore, in this project, we will work with the two, evaluate and find the one that works best for us.

2.2.3 Move_Base

move_base [14] is at the heart of the Navigation Stack; the task of this module is to move a robot from its current position to a goal with the help of other navigation nodes. It uses *costmap_2d* package [6] to create a global costmap related to the global environment and a local costmap related to the local environment of the robot. The size of the local and global costmap can be configured. The Map data from the SLAM algorithm is used to populate the global costmap. In contrast, the local map is populated using various sensors such as range detectors and lasers and is dynamic. It then uses the selected global and local path planning algorithms to navigate the environment.

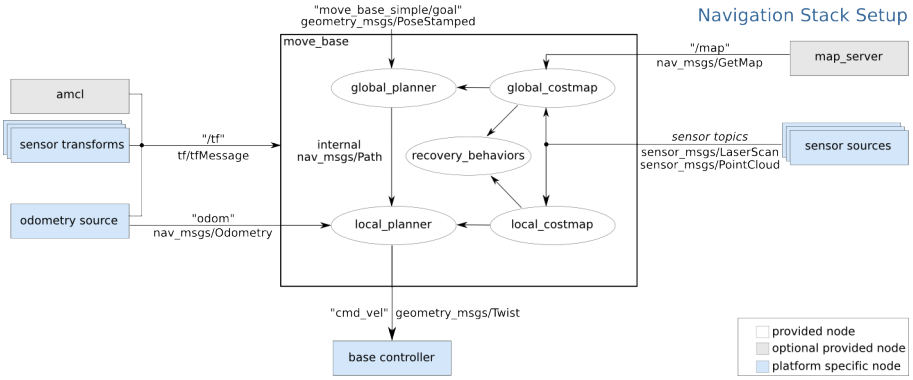


Figure 2.1: A *high-level* view of the *move_base* node and it's interaction with other components of navigation stack, from ROS Wiki [14].

Both path planning algorithms work in different ways; the global planner takes the robot's current position and maps the shortest path to the goal, whereas the local planner works over the local costmap, and, since the local costmap is smaller, it usually has more definition. It, therefore, can detect more obstacles than the global costmap. Thus, the local planner is responsible for creating a trajectory rollout over the global trajectory that can return to the original global trajectory with less cost.

move_base also supports various recovery behaviours Fig 2.2 for scenarios when the robot thinks it is stuck. These recovery behaviours help make the path planning more robust, although they are not perfect, as in specific scenarios robot would brute force its way and crash into an obstacle.

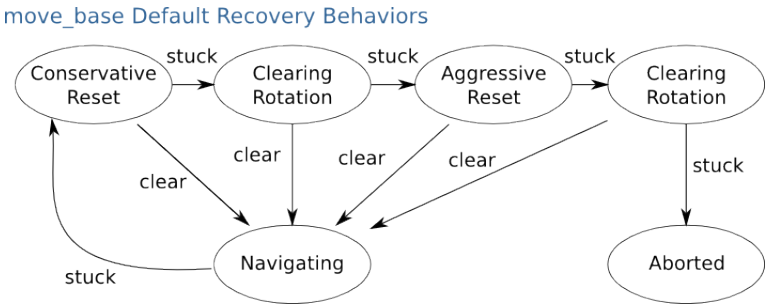


Figure 2.2: Flowchart of the default recovery behaviours in *move_base* and their flow of execution, from ROS Wiki [14].

2.2.4 Path Planning

Over the years, several path planning algorithms have been proposed to navigate and explore a map. ROS supports the implementation of various algorithms such as Dijkstra's algorithm

and A* algorithm for global path planning and works using the map generated by SLAM. Dijkstra algorithm begins traversing from an initial point and explores all possible ways until the destination is reached. In contrast, the A* algorithm tries to look for a better path using a heuristic function and calculating and prioritizing better nodes, reducing this computational cost.

Algorithms such as Dynamic Window Approach(DWA) and Trajectory Rollout are used for local planning and work in robots defined velocity control space and give trajectory commands to reach a goal. The local planning algorithms use sensory information to plan a path. In this project, the optimization of the path planning algorithms is not highly prioritized but in research performed in paper [17], it is evident that both A* and Dijkstra have similar performance in small-scale maps, and the performance of Dijkstra reduces as the map size increases because it searches for all possible ways to reach the goal. Since we will work with small-scale maps, the algorithm choice will not affect the outcome.

2.2.5 Exploration

Autonomous exploration and mapping of an unknown environment has been research interest in the robotics industry for a long time. ROS offers several packages for single or multi-robot autonomous exploration. Some of these packages are *rrt_exploration* [21] which implements a multi-robot RRT-based map exploration algorithm, *frontier_exploration* [1] and *explore_lite* [8] which are frontier based and made for single robot exploration.

Both *frontier_exploration* and *explore_lite* work by searching for frontiers. Frontiers are transitions between explored and unknown space and are defined as navigation goals for the robot. *frontier_exploration* requires the user to select an area to explore. The exploration goal contains an initial point to start exploration, and a polygonal boundary to limit the exploration scope, which the robot explores until all the frontiers are discovered. Whereas, *explore_lite* provides greedy frontier-based exploration and requires no user input. Once the node runs, the environment is explored until no frontiers can be found. Since, we want a completely autonomous robot, we will be working with *explore_lite* package.

2.2.6 Object Recognition

Due to the scale of the project and time limitations, we use feature matching for object recognition in place of machine learning algorithms. Feature detection and matching techniques are used to detect features in target images and match them if presumed to be identical according to a given threshold. Features are specific structures in images such as corners, edges, points and objects. As computer vision techniques improve, various methods for detecting and matching features have been developed over the years, such as SIFT, SURF, and BRISK are scale-invariant feature detectors. ORB and BRISK were invariant to affine change, and SIFT and SURF have the highest overall accuracy. SURF is approximately three times faster than SIFT, with a slight trade-off in accuracy.

Feature detection and matching techniques consist of a detector and a descriptor. The detector is used to locate and find areas of interest in a provided image, such as edges and corners. By contrast, the descriptor provides a robust description of the detected features. It provides increased matching performance through high invariance, even for changes in scale, rotation, and partial affine image transformation of each feature in the image pairs.

ROS has *find_object_2d* [11] package that provides a ROS integration for **Find-Object** application [12]. *find_object2d* includes a simple Qt interface and implements state of the art

feature detectors and descriptors. The package allows easy experimentation and configuration of a number of algorithms. If using a depth camera the package also publishes the center of the detected object and rotation over TF. We will use this package to detect object and get the coordinates.

Chapter 3: Implementation

3.1 Simulator

This project was performed in a 3D simulated environment. Therefore it was essential to choose a simulator that was well integrated with ROS and has a robust physics engine. For this, we looked at two simulators, Gazebo and Webots.

Gazebo is an open-source 3D dynamic simulator that can simulate robots in indoor and outdoor environments efficiently and correctly. It has a rich collection of sensors that can be simulated to see the environment.

Webots is also an open-source 3D simulator maintained by Cyberbotics Ltd. It supports a multitude of programming languages to develop robot controllers, and they can be written in C++, Java, C, Python and ROS.

Eventually, Gazebo was chosen as the simulator of choice because of its more significant community and better integration with ROS; the advantages and disadvantages of each are mentioned in the table 3.1.

Simulator	Advantages	Disadvantages
Webots	• Robust GUI-based model editor allows for ease of creating worlds and new robots. • Supports many programming languages. • A wide range of robots are supported and work with almost all versions of webots.	• ROS integration is supported but requires many adjustments by the developer. • The Webots models needed to be converted to URDF. • It does not publish all pieces of information by default that ROS requires to utilize a sensor.
Gazebo	• Supports a wide range of sensors and Robots. • Large community support and documentation. • Robust ROS integration and publishes all the topics required by ROS by default.	• The GUI model editor is good only for basic tasks. • Modeling a robot requires scripting in SDF or URDF format. • Models usually have to be imported from other 3D modelling applications.

Table 3.1: Difference between Gazebo and Webots.

3.2 Robot Configuration

The robot used is Rosbot 2.0 from husarian [9]; it is a 4-wheeled differential drive robot equipped with Lidar, RGB-D camera, IMU, encoders, and ultrasound distance sensors. The Lidar has been removed for this project, and the depth camera has been configured to simulate Microsoft Kinect.

The Microsoft Kinect's depth information has been configured for a range of *0.5 metres* to *3 metres*; any depth information beyond the limits is considered unreliable and ignored. Since the Kinect cannot detect information below 0.5 metres, the robot is equipped with four ultrasound distance sensors which have a range of *0.01 metres* to *0.9 metres*. Also, to increase the robot's viewing angle, a *Continuous Effort Joint* has been added that rotates the depth camera by 30 degrees left and right at a rate of *0.2 Hz*. The robot can be seen in Fig. 3.1, Robots tf frames were also configured, tf keeps track of the robot's coordinate frames. The tf frames are used by the ROS Navigation stack; therefore, the configuration of the robot and it's frames has to be correct for it to work.

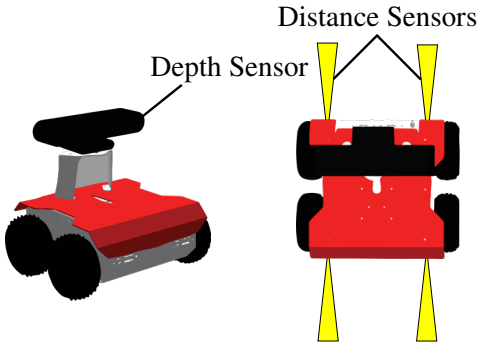


Figure 3.1: Rosbot and it's sensor config.

3.3 Mapping and Localization

3.3.1 Depth information to Laser scan

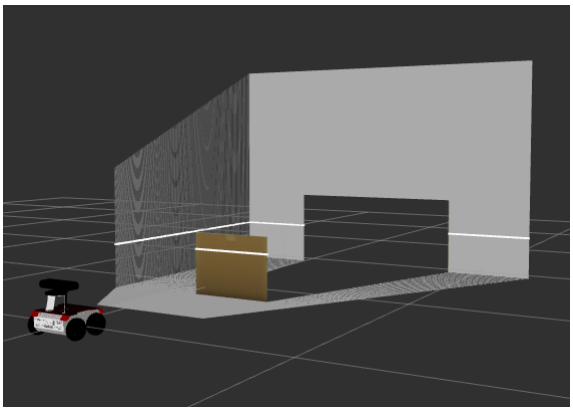


Figure 3.2: Depth Image to Laserscan

The mapping algorithms require laser scan data such as Lidar. Since we are working with just a depth camera, the depth information must be converted to laser data. There are two ways to convert the depth information received from Kinect to laser information, one is converting the point cloud information to laser data using *pointcloud_to_laserscan* package [2], and the other is converting the depth image using *depthimage_to_laserscan* [3]. During the

project, we tried both, but there was an apparent difference between the two. It was noticed that the point cloud conversion is more challenging to configure correctly and is sometimes unreliable, resulting in inconsistent maps. In contrast, the depth image information was first filtered to remove the noise and then converted to laser data, resulting in better and more precise information and better map generation. The laser data from the depth image is taken at the height of 10 units; this can be seen in Fig. 3.2, where the white line represents the fake laser and the depth information from Kinect.

3.3.2 Gmapping and RTAB-Map evaluation

We evaluated both Gmapping and RTAB-Map algorithms by robot travelling the same path for both algorithms and calculated the difference from its actual coordinates in Gazebo to coordinates published by both algorithms at the goal. The SLAM algorithm publishes the localization information in the /map coordinate frame, whereas Gazebo uses its own global coordinate frame. Therefore, the robot is spawned at x = 0 and y = 0 in Gazebo to make sure both coordinate frames coincide for more straightforward calculations. Therefore, it is computed using:

$$d_{distance\_to\_goal} = \sqrt{(x_{map} - x_{gazebo})^2 + (y_{map} - y_{gazebo})^2} \tag{3.1}$$

The test was run 20 times for each algorithm. With various configuration parameters, the average error for Gmapping was 1.232, with 0.783 being the best performance achieved. In contrast, for RTAB-Map, the average error was only 0.280, with 0.043 being the best result. The results are shown in Table 3.2.

Due to the performance results in our evaluation and the quality of the map generated Fig. 3.4b, the SLAM method of choice was RTAB-Map. It also allows configuration through its immense number of parameters, the setup for RTAB-Map used in this project is shown in Fig. 3.3, we use the 2D Occupancy Grid map generated by the algorithm to reduce the computational cost required.

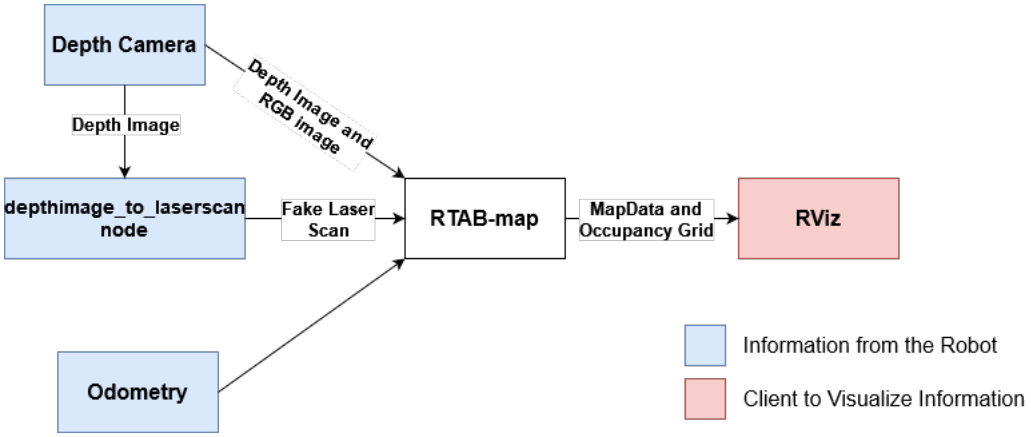
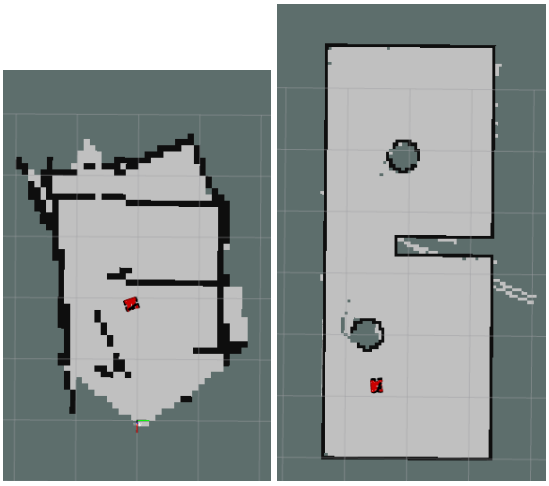


Figure 3.3: RTAB-Map sensor configuration used.

Metric	Gmapping	Rtab Map
Average Error	1.232	0.132
Minimum Error	0.280	0.043

Table 3.2: Testing results of Gmapping and RTAB-Map.



(a) Gmapping (b) RtabMap

Figure 3.4: Maps

3.4 Tuning Navigation Parameters

move_base requires fine tuning of configuration parameters for maximizing the performance of global and local path planners. The default parameters of the *move_base* configuration files do not work for this projects setup, therefore the parameters are tuned using trial and error technique and visual inspection to achieve the best balance in navigation performance and computational cost. The parameters are broken down into 4 files and some of these are shown with their values and description in Tables 3.3, 3.4, 3.6, 3.7, and 3.5.

Parameter	**Value**	**Description**
shutdown_costmaps	false	Do not shut down the costmaps of the node when move_base is in an inactive state.
controller_frequency	5.0	The frequency to run the control loop and send velocity commands to the base.
controller_patience	3.0	Wait in seconds before fallback to recovery behaviors if cannot find valid control command.
recovery_behaviors	[{name: rotate_recovery, type: rotate_recovery/RotateRecovery}]	Only use Rotate Recovery behavior.
planner_frequency	1.0	Global planning loop frequency.
planner_patience	5.0	Wait in seconds before fallback to recovery behaviors if cannot find path.
base_global_planner	”navfn/NavfnROS”	Global planner to use for path planning

Table 3.3: move_base parameters stored in move_base_params.yaml

Parameter	Value	Description
footprint	[[0.12, 0.14],[0.12,-0.14], [-0.12,-0.14],[-0.12,0.14]]	The footprint of the robot.
obstacle_layer		Layer tracks the information from laser scanners.
track_unknown_space	true	Important for path planning in unknown area.
obstacle_range	2.5	Max range in metres of the obstacle to track.
raytrace_range	3.0	Maximum range in meters at which to raytrace out obstacles from the map.
observation_sources	laser_scan_sensor	Sensor information to use for obstacle layer.
range_sensor_layer		Layer configuration for distance sensors.
clear_on_max_reading	true	Clear the costmap if max value.
topics	[/range/fl, /range/fr]	Only use the front two ultrasound sensors.
inflation_layer		Inflates the obstacles in order to make the costmap represent the configuration space of the robot [19].
cost_scaling_factor	5.0	A scaling factor to apply to cost values during inflation.
inflation_radius	0.55	The radius in meters to which the map inflates obstacle cost values.

Table 3.4: Common costmap parameters for global and local costmaps, stored in costmap_common_params.yaml

Parameter	Value	Description
max_vel_x	0.2	The maximum velocity allowed.
min_vel_x	0.05	The minimum velocity allowed.
max_vel_theta	1.5	The maximum rotational velocity allowed.
min_vel_theta	-1.5	The minimum rotational velocity allowed
holonomic_robot	false	
meter_scoring	true	Use 'meters' as units.
xy_goal_tolerance	0.15	Distance to goal tolerance to consider successful.
yaw_goal_tolerance	0.25	Tolerance in radians for the controller in yaw/rotation when achieving its goal.

Table 3.5: Parameters used by TrajectoryPlannerROS for controlling the robot, stored in trajectory_planner.yaml

Parameter	Value	Description
update_frequency	5	frequency at which to update map.
publish_frequency	2.0	Frequency at which to publish map display information.
transform_tolerance	0.5	Delay in transform (tf) data that is tolerable in seconds.
static_map	false	
rolling_window	true	Use a rolling window local map, i.e follow the robot.
width	4.0	width of the local map.
height	4.0	Height of the local map.
resolution	0.1	Resolution of the map.
plugins		Layers to use for building the local costmap.
name: range_sensor_layer	type: range_sensor_layer:: RangeSensorLayer	
name: obstacle_layer	type: costmap_2d::VoxelLayer	
name: inflation_layer	type: costmap_2d::InflationLayer	

Table 3.6: Local costmap parameters, stored in local_costmap_params.yaml

Parameter	Value	Description
update_frequency	1.0	Frequency at which to update map.
publish_frequency	0.5	Frequency at which to publish map display information.
transform_tolerance	0.5	Delay in transform (tf) data that is tolerable in seconds.
static_map	true	
plugins		Layers to use for building the global costmap.
name: static_layer	type: costmap_2d::StaticLayer	
name: obstacle_layer	type: costmap_2d::VoxelLayer	
name: inflation_layer	type: costmap_2d::InflationLayer	

Table 3.7: Global costmap parameters, stored in global_costmap_params.yaml

3.5 Object Recognition

Feature extraction algorithms require an image with many features to work and do not work on images with only a few features, the more features, the better the algorithm matching performance. We have chosen the object in Fig 3.5 as the object of choice to find in the scene. Since the robot can approach the object from any direction, we need images from all possible directions to match the features. Since feature matching algorithms are not illumination invariant, we also need pictures in various lighting scenarios for the matching to be successful in all lighting conditions. The number of pictures has a direct impact on the real-time performance of the system. The feature extraction and description are highly computative, and the processing power required scales with the number of pictures to match against. The configuration used for *find_object_2d* package that has been found to have best performance is shown in Table. 3.8.

Figure 3.5: Coke Can object for feature matching.

Parameter	**Value**
Detector	SURF
Descriptor	ORB
Homography method	RANSAC
Nearest Neighbout	Brute Force

Table 3.8: Find-Object Cofiguration.

3.6 Project Setup and Information Flow

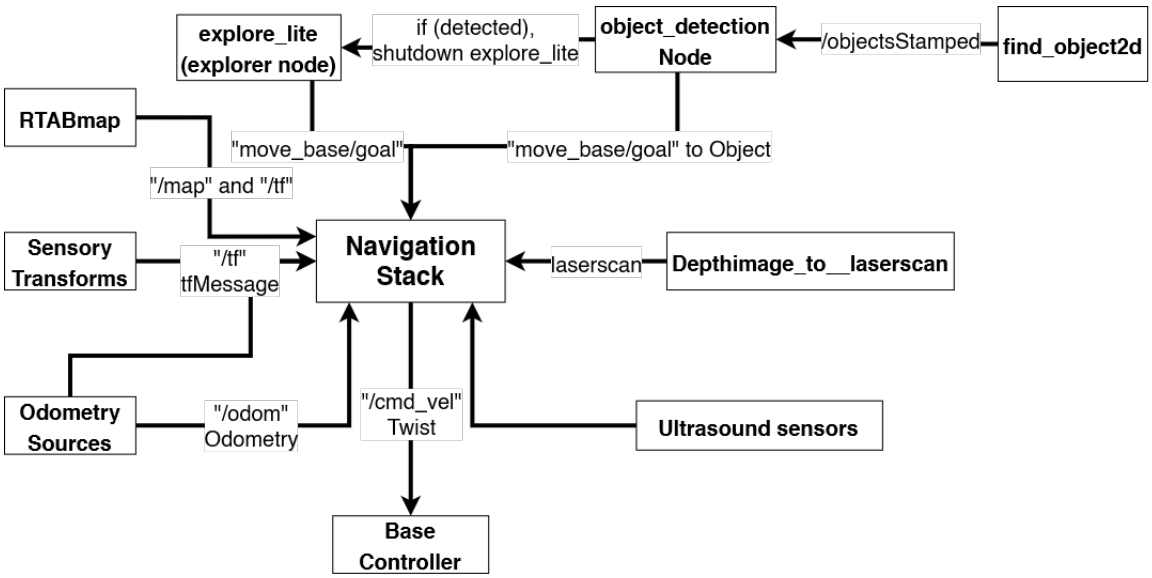


Figure 3.6: Project setup and information flow.

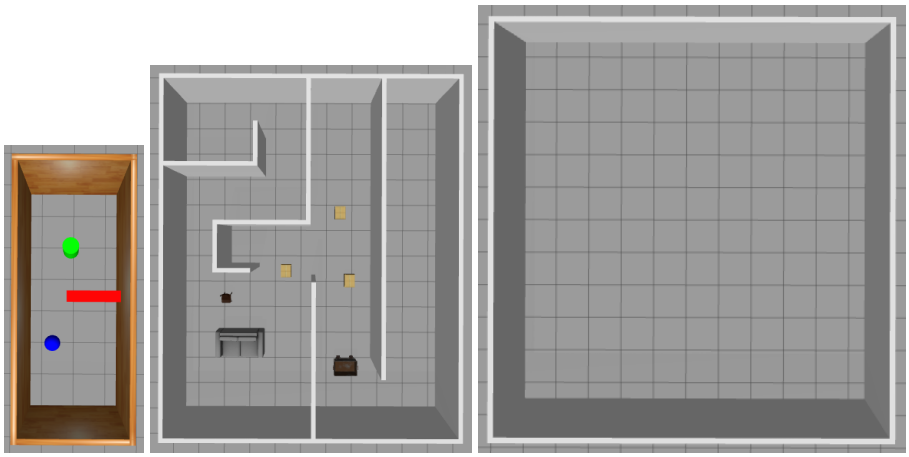
Fig. 3.6 shows the project's final setup and the information flow. Once the project starts running, the *explore_lite* node runs until the object to be found is detected. Once the object is detected *find_object_2d* publishes a topic named "objectsStamped", this topic contains information about the 'id' of the object detected and it's 'frame'. A node named 'object_detection' has been implemented to process the information in 'objectsStamped'. If the object is detected, 'object_detection' takes the object's coordinates in the object's frame and converts it into the *map* frame. This is required because move_base uses the *map* coordinate frame to send goal commands. The exploration node is shut down through system commands, and the converted coordinates are sent to the Navigation Stack as the next goal for the robot. Once the goal is completed, a message 'Successfully found object' is printed in the console.

Chapter 4: Results

In this section we will talk about the results obtained on the setup proposed in this project.

4.1 Worlds

Three worlds have been created to navigate the robot and test the solution presented in this project. The world in Fig. 4.1a is small, with little room for movement for the robot. The world in Fig. 4.1b is built a bit more complex, and random obstacles have been added to make it difficult for the robot to navigate a path. The world in Fig. 4.1c is built to test if the solution works in large empty areas.



(a) world1 (b) world2 (c) world3

Figure 4.1: Worlds

4.2 Tests

There are a total of 3 worlds designed for tests. Each world is tested in low, standard, and bright lighting conditions to check if the solution works in each scenario. Each scenario is tested ten times, resulting in a total of 30 tests for each world. The object to find is kept randomly anywhere in the world. The frequency of success and failures of the solution for each world, and in-depth information on failures are reported in the following sections.

4.2.1 World 1

The graph in Fig. 4.2 shows the results of the tests run for World 1. We can see that the solution proposed does not work well for the given world as the success rate across all three scenarios is approximately 40%.

World 1 is small, and the area of movement is relatively less, due to which the robot is close to obstacles in all instances. Since Kinect cannot detect obstacles closer than its minimum range, the global map does not know the location of the obstacle. The distance sensors were implemented to counter this problem, but they only detect obstacles in front. There is no information about obstacles on the sides of the robot if they are not detected by Kinect, resulting in collisions when turning.

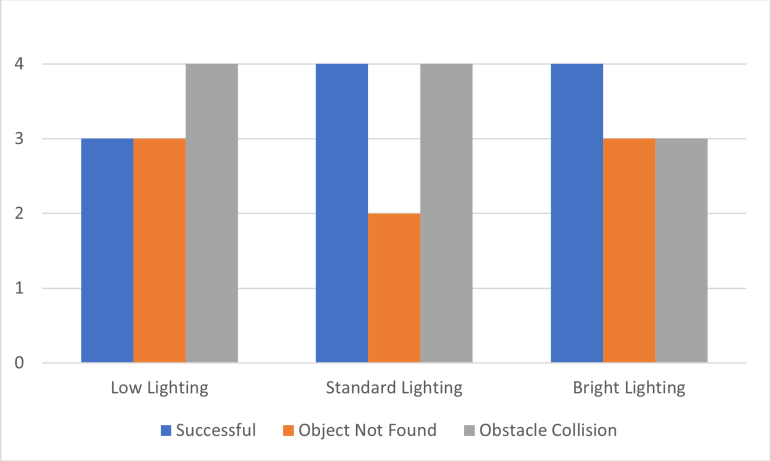


Figure 4.2: World 1 results.

The reason for the object detection failure is the same in this scenario. If the robot is closer than 0.5 meters when it sees the object, the depth information received is ignored because of the Kinects configuration. And since the *find_object_2d* uses the depth information to calculate and broadcast the detected object's location, the result is a failure due to no location information.

4.2.2 World 2

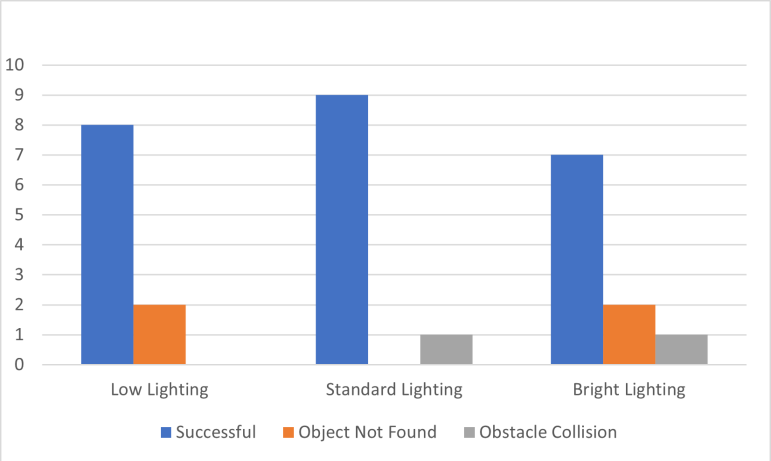


Figure 4.3: World 2 results.

From the results of the tests for World 2, shown in Fig. 4.3, we can say that the world is ideal for our robot and the solution presented. The success rate is almost 80%. This result is because the world is larger, and the greater distance between obstacles allows the robot to detect and update the map with obstacle information before coming near the obstacle. Also, because of the more space between obstacles, the path planning algorithms work better in keeping the robot outside the inflation radius and, therefore, is able to avoid them.

Also, the failure in object detection is expected due to the reduction in features with low and bright lighting conditions. But, since the sample images are taken in both conditions the failure rate is low.

4.2.3 World 3

In the world 3, the robot failed to explore the environment therefore could not find the object, Map shown in Fig. 4.4. This was the result of the short range of the Kinect sensor, the map contains information which is either unknown, free or obstacle. The spaces are marked free according to the distance to the obstacle. Since, the maximum range of kinect is only 3.0 metres in our setup, in the areas with no obstacles the map was not updated. This failed the *frontier_lite* package from finding a path to the next possible frontier.

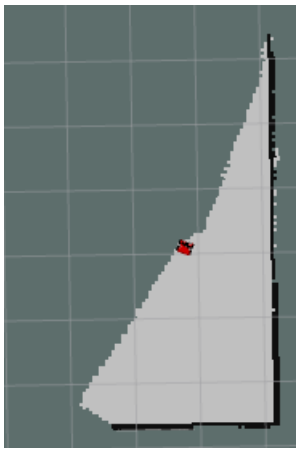


Figure 4.4: World 3 results

Chapter 5: Conclusion

5.1 Achievements

We started the project by defining the objectives and tools and techniques required to fulfill as part of this project. We were able to achieve the goal of finding an object in an unknown environment using the various available ROS packages and compare the various options available. The braking down of the goal into smaller objectives helped us to evaluate and find the best working configurations of the ROS packages at each stage of the development. The proposed setup helps to achieve the goal at a high efficiency and least number of failures in some world scenarios whereas it fails in the others.

It was found that the Kinect sensor is suited only for certain scenarios and that to increase accuracy of feature matching in changing lighting conditions more pictures are required.

The knowledge gained about the various configuration parameters and packages can serve as a guideline and base for development of autonomous robots by others.

5.2 Limitation and Future Work

The project presented in this paper has its limitations. The project was conducted in a 3D simulator; therefore, the results may not transfer entirely and deliver similar results in a real-world application. In the tests, it is seen that the Kinect does not perform well in scenarios with large empty spaces and when obstacles are close to the sensor. Research can be done on other low-cost sensors that can improve the robot's navigation performance in such scenarios.

The tests were conducted in scenarios where there is only one object with a high number of features, so the chances of detecting the wrong object because of feature mismatch was not present. Machine learning algorithms can be used in scenarios with more objects, and the robot can also be taught to find multiple objects.

During the project, it was also assumed that the same *move_base* parameters would work in all indoor scenarios, but this failed in our tests. This can be remedied by creating configuration files that can be run depending on the expected environment and would result in better performance.

The project can also be further developed to allow the robot to do tasks such as following a given object, picking and delivering and various others. Also, due to my limited knowledge of the ROS framework and the scale of the project, someone with more intensive knowledge will be able to optimize the parameters for higher accuracy.

5.3 Challenges Faced

Since, this project has been developed in a 3D simulator running on Ubuntu over a VM, most of the challenges faced during this project were technical and related to the utilization of GPU in order to run tests at real time performance. Also, over the years ROS has released many versions and some packages only work with certain versions of ROS. Therefore, configuring the packages to work with the version in use was a time consuming task.

Appendix A: Appendix

A.1 Github Repository

The codebase for this project is hosted at the following Github repository:

`https://github.com/RandomGuy-coder/autonomous-mobile-robot`

Bibliography

[1] Paul Bovbel. frontier_exploration. http://wiki.ros.org/frontier_exploration, 2017. [Online; accessed 24-July-2022].

[2] Paul Bovel and Tully Foote. Wiki pointcloud_to_laserscan. http://wiki.ros.org/pointcloud_to_laserscan, 2015. [Online; accessed 12-July-2022].

[3] Rockey Chad. Wiki depthimage_to_laserscan. http://wiki.ros.org/depthimage_to_laserscan, 2020. [Online; accessed 12-July-2022].

[4] Bruno M. F. da Silva, Rodrigo S. Xavier, Tiago P. do Nascimento, and Luiz M.G. Goncalves. Experimental evaluation of ros compatible slam algorithms for rgb-d sensors. In *2017 Latin American Robotics Symposium (LARS) and 2017 Brazilian Symposium on Robotics (SBR)*, pages 1–6, 2017.

[5] DARPA. Darpa subterranean challenge. https://www.subtchallenge.com/, 2015. [Online; accessed 16-Sep-2022].

[6] Dave Hershberger Eitan Marder-Eppstein, David V. Lu. costmap_2d. http://wiki.ros.org/costmap_2d, 2018. [Online; accessed 07-July-2022].

[7] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1):34–46, 2007.

[8] Jiri Horner. explore_lite. http://wiki.ros.org/explore_lite, 2017. [Online; accessed 24-July-2022].

[9] Husarion. Rosbot 2. https://husarion.com/manuals/rosbot/, 2022. [Online; accessed 15-July-2022].

[10] Stefan Kohlbrecher, Oskar von Stryk, Johannes Meyer, and Uwe Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, pages 155–160, 2011.

[11] Mathieu Labbe. find_object_2d. http://wiki.ros.org/find_object_2d, 2016. [Online; accessed 24-June-2022].

[12] Labbé, M. Find-Object. http://introlab.github.io/find-object, 2011. [Online; accessed 24-June-2022].

[13] Mathieu Labbé and François Michaud. Online global loop closure detection for large-scale multi-session graph-based slam. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2661–2666, 2014.

[14] Eitan Marder-Eppstein. move_base. http://wiki.ros.org/move_base, 2020. [Online; accessed 07-July-2022].

[15] Eitan Marder-Eppstein. Navigation. http://wiki.ros.org/navigation, 2020. [Online; accessed 06-July-2022].

[16] Raul Mur-Artal, J. Montiel, and Juan Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31:1147 – 1163, 10 2015.

[17] Dian Rachmawati and Lysander Gustin. Analysis of dijkstra's algorithm and a* algorithm in shortest path problem. *Journal of Physics: Conference Series*, 1566:012061, 06 2020.

[18] Open Robotics. ROS-robot operating system. `https://www.ros.org/`, 2021. [Online; accessed 24-June-2022].

[19] ROS. Inflation costmap plugin. `http://wiki.ros.org/costmap_2d/hydro/inflation`. [Online; accessed 30-July-2022].

[20] Marco Tranzatto, Takahiro Miki, Mihir Dharmadhikari, Lukas Bernreiter, Mihir Kulkarni, Frank Mascarich, Olov Andersson, Shehryar Khattak, Marco Hutter, Roland Siegwart, and Kostas Alexis. Cerberus in the darpa subterranean challenge. *Science Robotics*, 7, 05 2022.

[21] Hassan Umari. rrt_exploration. `http://wiki.ros.org/rrt_exploration`, 2018. [Online; accessed 24-July-2022].

[22] Wikipedia. Darpa grand challenge. `https://en.wikipedia.org/wiki/DARPA_Grand_Challenge`, 2022. [Online; accessed 16-Sep-2022].