Motion Planner to Explore Unknown Rough Terrain

R.M.K.V. Ratnayake (150533H)

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

December 2019

Abstract

Autonomous navigation is one of the main research areas related to modern robotics. Unstructured environment navigation can be considered as one of the main branches of Autonomous navigation. The subsystem of the robot that is responsible for taking decisions related to navigation is called a motion planner. These systems analyze the environment, recognizes obstacles and calculate a path, which the robot can follow to achieve its goals. Recent research has been focusing on building motion planners that are capable of navigating in unstructured environments. To address the issue of navigating in an unstructured unknown terrain without prebuilt maps, I present a motion planner that is capable of planning a path for the robot to follow, using point clouds of the environment. This research focuses on building a motion planner consisting of a point cloud analyzer, a path planner and a velocity controller that is capable of finding a path in an unstructured area. This research also describes how the proposed motion planner can be used to explore and map an unstructured environment.

Keywords: Unstructured terrain navigation, Path planning, Point Cloud Based Navigation, Unknown Area Mapping, Octomap based Navigation

Acknowledgment

First and foremost, I would like to express my gratitude to my supervisors, Dr. Chandana Gamage and Dr. Sulochana Sooriyarachchi for giving me this research opportunity. Their guidance, motivation, support and consistent supervision throughout the research duration helped me to complete this research successfully. In addition to that, my sincere thanks also go to Dr. Kutila Gunesekera who evaluated the research project thoroughly, pointed out various issues, possibilities and helped me understand them. As well, thanks go to Ms. Tamasha Malepathirana for evaluating my research project and providing valuable feedback.

I would like to extend my gratitude to Ms. Tharushi de Silva and all the other postgraduate students at the Department of Computer Science and Engineering who helped me during the research by making valuable comments and suggestions.

Finally, I would like to extend my gratitude to my friends who helped me with the development of various components of the motion planner by lending an ear to the various ideas and concepts I had.

Contents

Contents iii
List of Figures iv
List of Tables v
List of Abbreviations vi
1. Introduction 1
  1.1. Background 1
  1.2. Problem Statement 2
  1.3. Motivation 3
  1.4. Research Objectives 3
2. Literature Review 4
  2.1. Path Planning 4
  2.2. Outdoor Rough Terrain Exploration 5
  2.3. Octomap based path planning 6
  2.4. Unstructured area path planning 8
3. Methodology 10
  3.1. Proposed Solution 10
  3.2. Implementation 10
    3.2.1. Goal Identifier 12
    3.2.2. Path Planner 17
    3.2.3. Ground Evaluator 28
    3.2.4. Velocity Controller 30
    3.2.5. Custom messages used in Topics and Services 35
4. Experimental Evaluation and Results 41
  4.1. Goal Identifier Testing 42
  4.2. Velocity Controller Testing 44
  4.3. Ground Evaluator Testing 45
  4.4. Path Planner Testing 45
  4.5. Whole system Testing 46
  4.6. Real-World Testing 50
5. Conclusion 53
References 54

List of Figures

Figure 3.1 Connectivity between subsystem 11
Figure 3.2 Technology stack of the motion planner 11
Figure 3.3 Voxel State 12
Figure 3.4 Structure of goal_identifier_node interface 15
Figure 3.5 IdentifierObject structure 16
Figure 3.6 Complete structure of Goal Identifier Sub system 17
Figure 3.7 Inflation (green) of obstacles (blue) 21
Figure 3.8 Structure of path_planner_node interface 24
Figure 3.9 Decision sequence inside systemCallback 25
Figure 3.10 Building the 2D grid for navigation 27
Figure 3.11 Combined path_planner_node and plannerObject 27
Figure 3.12 Structure of ground evaluator 29
Figure 3.13 Structure of velocity_control_node 35
Figure 4.1 Back of Kobuki (virtual) 41
Figure 4.2 Front of Kobuki (virtual) 41
Figure 4.3 Gazebo Virtual testing environment 41
Figure 4.4 Combined system of VisualizerObject and goal_visualizer_node 43
Figure 4.5 Undiscovered clusters' center points (red) and goal (green) 43
Figure 4.6 Structure of test_velocityControl_node 44
Figure 4.7 Combined test_velocityControl_node and velocity_control_node 44
Figure 4.8 Before(left) and after(right) analyzing for water (Image 2) 45
Figure 4.9 Before(left) and after(right) analyzing for water (Image 1) 45
Figure 4.10 Grid map with obstacles (blue), padding (green) and path (red) 48
Figure 4.11 Gazebo Testing Environment 48
Figure 4.12 Several Stages of Mapping and navigation process 49
Figure 4.13 Stages of first environment mapping 50
Figure 4.14 First Testing environment 50
Figure 4.15 Navigation grid of the real environment testing 51
Figure 4.16 Second Testing Environment 51
Figure 4.17 Stages of second environment mapping 51
Figure 4.18 Navigation grid of the second environment 52

List of Tables

Table 3.1 goal_identifier_node connections 14
Table 3.2 Functions available in the IdentifierObject 16
Table 3.3 global_path_planner_node connections 22
Table 3.4 PlannerObject functions and functionality 26
Table 3.5 Velocity_control_node connections 33
Table 3.6 pointData message structure 36
Table 3.7 pointDataArray message structure 36
Table 3.8 gridPoint message structure 37
Table 3.9 gridRow message structure 37
Table 3.10 gridMap message structure 38
Table 3.11 goalControl service message structure 38
Table 3.12 goalRemove service message structure 39
Table 3.13 baseDrive service message structure 39
Table 3.14 baseRotate service message structure 40
Table 3.15 systemControl service message structure 40
Table 4.1 test_goalIdentifier_node connections 42
Table 4.2 goal_visualizer_node connections 42
Table 4.3 functions and functionality of VisualizerObject 43

List of Abbreviations

Abbreviation	**Meaning**
FPGA	Field Programmable Gate Array
GPS	Geo Positioning System
HSV	Hue Saturation Value (Image format)
IMU	Inertial Measurement Unit
RGBD	Red Green Blue and Depth
ROS	Robot Operating System
RRT	Rapidly-exploring Random Tree
SLAM	Simultaneous Localization and Mapping

1. Introduction

1.1. Background

The project “Motion planner to explore unknown rough terrain” is focused on building a motion planner for “Search and Rescue robot” which is currently under development at the Intellisense Lab of Computer Science and Engineering Department of University of Moratuwa. Motion planners are systems that perceive the environment and calculate paths in the environment that the robot can take to reach a goal without colliding with the obstacles in the environment, without falling to holes in the ground or without tripping over mounds on the ground.

All the moving robots require a motion planner system and these systems tend to be unique for different kinds of robots depending on the size of the robot, shape of the robot, purpose of the robot and the environment it is supposed to work.

Motion planners usually contain two main components, a global planner and a local planner. The definition and structure of these two components can vary according to the developer and some motion planners might have only one component.

As a general explanation, global planners can be said to focus on perceiving the environment and on calculating the path to take without colliding with obstacles. Global planners are designed to calculate a path in the environment in a way that the robot can achieve its navigation goals. They calculate the direction the robot should head as well as the speed, tilt, pitch and pan of the robot body.

Local planners on legged robots can be said to focus on calculating footfall locations for the robot to move forward while maintaining balance. And local planners on non-legged (wheeled or tracked) robots can be said to focus on avoiding small obstacles that do not appear on the global map but can obstruct the movement. Dynamic obstacle avoidance is a common focus of local planners on both legged and non-legged robots.

Exploration is one of the main uses of robots and can be divided into two parts, known area exploration, and unknown area exploration. Known areas are areas with a complete map that indicates all the objects in that area. Unknown areas may or may not have maps like in known areas and even if such maps exist, they would only be partially complete, which means the locations of objects are not defined in those maps and needs to be identified by the robot. Throughout this report, unknown areas with maps are identified as “structured unknown areas” and unknown areas without maps are identified as “unstructured unknown areas”. Motion planners depend on these maps when they calculate paths from start to goal positions.

During exploration, robots must be able to traverse various kinds of even and uneven terrain according to their purpose. Flat terrains have an even surface without pits or mounds and rough terrains have uneven terrain that with mounds of pits. Flat terrain traversal is the basic traversal method and rough terrain traversing is an advance ability that some robots have. The motion planner needs to be designed according to the traversal method the robot is intended to use.

1.2. Problem Statement

When a robot traverses the terrain, indoor or outdoor, it must perceive the environment to calculate a path to keep itself from colliding with the obstacles. Additionally, it must calculate whether to go over or to go around the pits or mounds that are in the calculated path.

The purpose of the “Search and Rescue” robot is to explore an unstructured unknown area. The motion planner has to calculate a path in an environment where a goal position or a map does not exist. It must calculate a path so that it can travel around without colliding with any obstacles, map the environment and return to the starting position. It also has to consider pits and mounds on the calculated path.

According to the above explanation, the problem statement for the project will be,

“Building a motion planner that can explore an unstructured unknown environment”

1.3. Motivation

The main motive of this project is to support the ongoing "Search and Rescue robot" development by developing a motion planner that can be used as its navigation subsystem. Another motive is to learn about and contribute to the development of motion planners which are used in unstructured unknown environment navigation worldwide.

1.4. Research Objectives

- Design and development of a global planner that can calculate a path in an unknown unstructured environment to map the whole area.
- Design and development of a local planner that can calculate speed and direction according to the state of the path and distance to the next point.
- Development and Testing of a motion planning system that can navigate in an unstructured unknown environment.

2. Literature Review

2.1. Path Planning

Path planning and obstacle avoidance in robotics both came to attention once the robots started to move. Ever since then, a lot of research has been conducted to find ways to perceive the environment from the point of view of the robot and to make it decide on which path to take and how to move. Eldershaw and Yim came up with a motion planner [1] which had two components, a center of gravity planner and a foot location planner. The center of gravity planner calculated the path of the center of gravity of the robot and the foot planner decided where to place the leg. They defined the requirements a motion planner should fulfill and how the developers could design a motion planner. The center of gravity planner functioned as the global planner and foot planner functioned as the local planner.

Methods such as point clouds built using stereo cameras came forward as a method to perceive the environment but lacked a mathematical approach for calculations. As a solution, occupancy grid maps [2] were introduced by A. Elfes. Occupancy grid maps discretized the environment into a cubic matrix and gave each cell a probability of being occupied. Initially, data from proximity sensors and sonar sensors were used to build the occupancy grid map. But now point clouds generated from stereo cameras are used.

Occupancy grid maps were created to understand the environment and obstacles while algorithms such as Simultaneous Localization and Mapping were introduced for localization of robots with respect to the environment. Mapping happened with respect to the robot and since the robot was mobile, the need for ground relation came up. Bourgault et al. suggested to use both occupancy grid maps and SLAM to achieve localized occupancy grids. They proposed an Information space [3] which was constructed using occupancy grid map data and SLAM algorithm data. With it, it was possible to calculate a path considering the position and orientation of the robot.

Mapping was initially developed for static environments. Challenge was to handle dynamic environments where the environment constantly changed. Biswas et al. came up with the map differencing technique [4] that used several occupancy grid maps built at several points in time.

he combined them and compared them with each other using several techniques to learn about different objects in the map as well as to derive their paths. He had few issues with individually identifying objects that traveled closely.

Han et al. also proposed a method to navigate using layered cost maps. They proposed to predict an object's position using its own kinematic information as well as the other objects' information [5]. Then they created a new layer in the cost map and set different cost values for around objects predicted path. A combination of these gave the occupancy grid map that contained information of dynamic objects and it could be used for navigation in dynamic environments.

After mapping the environment, researchers had to come up with a way to make the robot understand where it was and where the obstacles were. Oussama Khatib came up with the artificial potential field concept [6] where the robot had a repellent force reaction between the robot and the obstacles or the occupancy grid map clusters and an attraction force reaction to the robot body and the goal. This made the robot to select the lowest potential route between obstacles to get to the goal. Initially, this was proposed for robot arm manipulation but was later got adopted in robotic navigation.

2.2. Outdoor Rough Terrain Exploration

One of the main challenges in outdoor exploration is the requirement to identify the holes and mounds in the ground which can become an obstacle to the robot. Sabe et al. proposed a method based on occupancy grid maps to sense the floor. The method used stereo cameras to build a point cloud of the ground and built an occupancy grid map. Then it used Hough transformation to extract the floor. With this, a single layer occupancy grid map [7] was built where it could be used to identify holes and mounds on the ground. It was unable to give the depth or the height of those holes or mounds. Even if this method was mainly developed for legged robots it can be used on non-legged robots as well.

Konolige et al. proposed a method of using stereo vision instead of laser rangefinders to map an unstructured outdoor area. They selected stereo vision because it had a higher range than laser range finders. They used a depth map to identify obstacles and floor and then sightlines extend it to distance [8]. They also used color images to identify roads using color models. They combined these on a 2D occupancy grid map using GPS data and visual odometry data for localization purposes. They then used this map for the path planning of an outdoor robot.

The ability to detect water puddles is another main requirement for outdoor navigation. If not identified correctly, this can be hazardous for most of the ground-based robots. Rankin et al. proposed a method to detect water during daytime [9] using RGBD cameras. They proposed to use multi cues of an image such as color, texture, detection of reflections and depth to detect water. They then proposed a rule base to combine the results from separate cues to a single one and they showed it works by implementing it into a run-time passive perception subsystem.

Different uneven terrains have different physical properties and the ability to control the robot's speed and other parameters according to the environment can lead to efficient and faster navigation. Siva et al. proposed a joint terrain representation and apprentice learning-based approach in [10] to change the robots' adaptability to the environment dynamically. Their approach focuses on treating representation of the environment and apprenticeship-based learning under a unified optimization framework. Their approach is also capable of identifying discriminative features of the terrain and adapt the robot accordingly.

2.3. Octomap based path planning

Robots need maps to traverse an area. But when robots improve from 2D movement to 3D movement, maps also needed to become 3D. Many layers for occupancy grid maps could solve this issue but working with all at the same time became computationally intensive for hardware mountable on robots. As a solution, Hornung et al. came up with a framework to generate volumetric 3D environmental models [11]. Their approach was based on octrees and used probabilistic occupancy estimation just like in occupancy grid maps. In addition to the occupancy state of a voxel, it also another two states that represent free and unknown voxels. Additionally, they proposed compression techniques for the map.

When navigation planning is done using Octomap, they use the occupancy data in the map to build a 2D Occupancy grid map. Maier et al. demonstrated this by using a consumer-level depth camera to build the Octomap of a room in real-time. They overlaid this newly acquired occupancy data on a previously built Octomap of the room and showed that data is correct [12]. Then they used the Octomap built using depth camera data to build an occupancy grid map which in turn was used to navigate a bipedal robot without colliding with obstacles in the room.

Using the Octomap to acquire the map and then converting it into a 2D map for navigation made sure that it was computationally feasible but also meant that losing data along a dimension. Since the targeted robot only had 2D movements it did not pose any issues. But when used in mobile manipulation systems, this loss of data became an issue because the robot had movements in all 3 dimensions. As a solution, Hornung et al. proposed a new planning approach that used a combination of multi-layered 2D and 3D representations. They used the Octomap generated from the depth camera to build several 2D occupancy grid maps on different height ranges [13]. This height was decided according to the shape of the robot. This was faster than directly building several maps from scratch but needed more computational power than before. They demonstrated the system using a mobile manipulation robot named PR2.

Octomap framework can be used to analyze the environment as it segments the space into cubes and presents it in a mathematical manner. Ramanagopal et al. proposed a method to use the Octomap framework to model structures as monuments and buildings [14] using a ground robot. They suggested using an approach similar to the wall following to map the perimeter of the structure. After detecting unmapped cavity areas, they proceed to explore them individually. For the localization purposes, they use a Visual Simultaneous Localizing and Mapping method which benefits from their choice of movement. Their approach does not assume bounds on the structure and thus initial dead reckoning localization can affect the visualization process. They have shown that their system is capable to create the model of the structure even under those conditions. This carries the concept of calculating the goal of path planning according to the requirement of the robot than setting it externally by an operator.

While Ramanagopal et al. proposed ground robot-based 3D modeling, Schmid et al. proposed the use of flying robots to map indoor and outdoor buildings [15]. In this approach, they use the ‘waypoint following’ approach while using the Octomap framework to identify 3D obstacles in the environment and to calculate paths around them. They use stereo cameras to capture the environment and have developed an FPGA based approach to do the required processing on the robot itself including Octomap generation.

Since Octomaps are able to create 3D maps of the environment, it becomes quite useful in relief operations performed in disaster zones. They can be used to map a damaged building or a disaster site which can be unreachable for humans. Dube et al. proposed a robotic navigation system that can be used by firefighters to map a disaster zone [16]. Their proposed system uses navigation points selected by a firefighter operator as goals to calculate paths. They use previously built Octomaps to calculate the initial paths and are capable of continuing the mapping from a previously stopped point rather than starting all over again every time they stop.

2.4. Unstructured area path planning

Robot path planning systems depend on maps on varying degrees to calculate paths. Some systems use completely defined maps and some use semi-defined maps that contain floor layout but not the obstacle positions. In semi-defined maps, they use depth cameras or laser rangefinders to detect the distance to the obstacles in the vicinity and use localization methods to correctly identify the position of the robot with respect to obstacles.

Navigation without maps forces the system to recognize floor and traversable areas in addition to mapping obstacles in the vicinity. Yang et al. presented a method that used stereo images to build a point cloud of the surrounding and then extract the floor from the data [17]. The method used edge detection to extract the free road from obstacles. They used RRTs to store these data and later on used for path planning. Their system was able to solve the navigation without building a global map.

Klaser et al. also proposed a stereo vision-based approach to navigate in an unstructured area in [18]. They propose to use a stereo camera to build a point cloud of the area and then use a probabilistic evaluation on the point cloud to reduce the noise. During the implementations, they used the Octomap framework for point cloud probabilistic evaluation. Since it is an unstructured environment, they have opted out to use Extended Kalman Filter for localization by integrating IMU data, GPS data and wheel encoder data. Ultimately, they prepare a navigability map by down projecting the Octomap and then uses motion primitives of their test vehicle for path planning. They have simulated the system and shown that their proposed system is capable of successfully navigating in an unstructured area.

Autonomous navigation in an environment with slopes and staircases can be mentioned as one of the main challenges in uneven terrain navigation. Wang et al. proposed a navigation system in [19] that uses the Octomap framework to identify and overcome slopes and staircases. Their proposed method was to create an Octomap of the environment using a 3D SLAM approach and then slice it at different heights to create several occupancy grids. They after evaluating them, they create a 'Traversability map' that considers slopes as traversable areas. In order to support this navigation, they use 2D SLAM for initial localization and camera relocalization method based on the regression forest for main localization. After creating the 'Traversability map', they use variable step size RRTs for path planning purposes.

Navigation systems prepare various kinds of maps to support their path planning processes. These maps have their own advantages and disadvantages unique to them. Occupancy grid maps, feature maps, topological maps, navigability maps, Traversability maps, and hybrid maps can be shown as examples. Each one except for the last hybrid maps captures one aspect of the environment and use it to navigate where hybrid maps are created using both feature maps and topological maps. Guivant et al. proposed a new hybrid map named Hybrid Metric Map [20] that bases itself on feature maps and incorporate other metrics representations. This enables users to use and extract various kinds of data from the same map and helps to increase the relativity of data. They also present a SLAM algorithm for path planning using all the maps at once. Since these can capture various aspects at once, it becomes useful when navigating in unstructured areas without early knowledge.

3. Methodology

3.1. Proposed Solution

The purpose of the “Search and Rescue” robot is to map an unstructured unknown area. The scope of this project is to build a motion planner so that the robot can build a map of the area effectively. The robot uses the Octomap framework to build the map using the point cloud generated by the depth camera of the robot. The proposed method to build the motion planner is to use the map developed using the Octomap framework for path planning. Following steps are proposed for navigation,

- Recognizing unexplored areas.
- Selecting the nearest point just beyond the explored region.
- Calculating an occupancy grid considering discovered obstacles
- Calculating the path according to the occupancy grid.
- Reaching the point
- Repeating the process again

3.2. Implementation

For the ease of development and testing, the motion planner has been divided into 4 separate subsystems as,

- Goal Identifier
- Path Planner
- Ground Evaluator
- Velocity Controller

Goal Identifier analyses the Octomap generated by the mapping system and recognizes the unexplored areas. It also calculates the nearest unexplored point just beyond the explored region.

The second subsystem, Path Planner also analyses the Octomap generated by the mapping system to recognize explored areas as well as the obstacles in it. It calculates a 2D occupancy grid considering discovered obstacles and then uses that grid to calculate a path using the A-star algorithm.

Ground Evaluator supports the Path Planner with recognizing water puddles that cannot be detected from the map built using Octomap. Goal Identifier, Path Planner and Ground Evaluator combined system acts as the global planner of the motion planner.

Velocity Controller connects the motion planner with the robot by converting Path Planner movement commands into velocity commands which can be recognized by the robot. This acts as the local planner of the motion planner.

Figure 3.1 shows the connectivity between subsystems. Arrows represent the direction of communication.

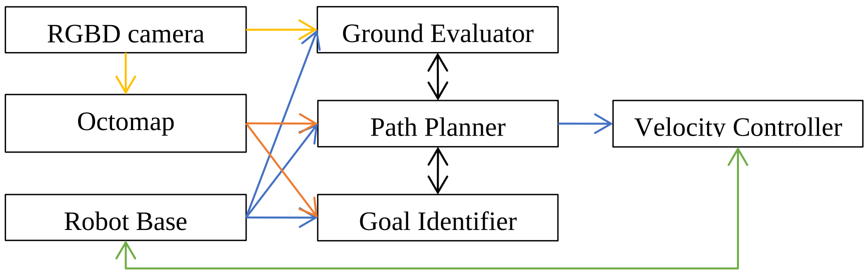


Figure 3.1 Connectivity between subsystem

The system was implemented on the Robot Operating System [21] along with the Octomap framework. Octomap Framework can be invoked as a ROS node named "Octomap Server" and publishes Octomap as a ROS message. Figure 3.2 explains the technology stack for the proposed motion planner.

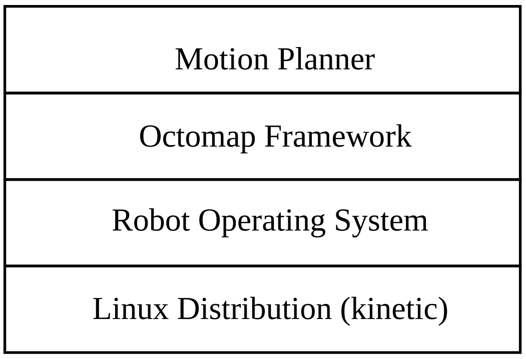


Figure 3.2 Technology stack of the motion planner

3.2.1. Goal Identifier

When a motion planner calculates a path, it takes a location (current location of the robot is used in general) as the start of the path, the location it intends to reach (henceforth “goal”) as the end of the path and attempts to calculate a path in between. So, the goal becomes one of the main requirements for calculating a path. But when unstructured unknown areas are concerned where a map or a goal has not been clearly defined, planning becomes impossible. Thus, it is important to clearly identify a goal such that a path can be calculated to explore the area efficiently.

3.2.1.1. Design

The Goal Identifier subsystem analyses the Octomap generated by the Octomap server ROS node and selects the nearest unexplored location. To achieve this, the subsystem contains two components,

- Discover-Clusters
- Find-Nearest-Cluster

Discover-Clusters takes the Octomap generated by the Octomap server and the dimensions of the area we are interested in exploring as input and produces a list of undiscovered clusters as an output. This component iterates through the voxels in the Octomap generated by the Octomap server. It divides the total area of the Octomap into clusters with a predefined size and iterates in them while inspecting the state of the voxels. Voxels have 3 states named occupied, unoccupied (free) and unknown [11] which is explained in figure 3.3,

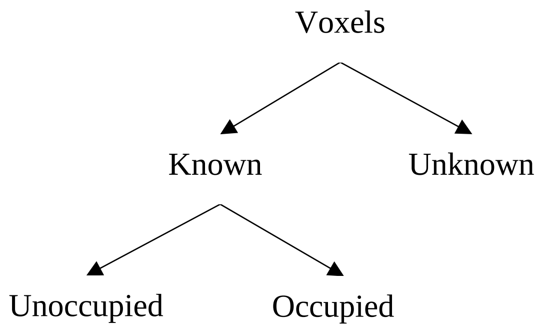


Figure 3.3 Voxel State

Discover-Clusters component then produces a list of center points of clusters that are undiscovered. Clusters are classified as “undiscovered” if the percentage of unknown pixels is higher than a predefined threshold. Following pseudocode describes the procedure,

Discover-Clusters (Octomap, area dimensions)

- Divide Octomap into clusters of a pre-defined size
- Iterate through all the clusters evaluating the state of voxels (free, occupied, undefined)
- If the percentage of undefined voxels in a cluster is over the threshold, flag it as an undiscovered cluster.
- Return a list of the center points of unknown clusters

Find-Nearest-Clusters is the other component of the Goal Identifier subsystem. It takes the list of cluster centers produced in the Discover-Clusters component as the input and calculates the nearest cluster to the current position. It iterates over the list of cluster centers while calculating the Euclidean distance to the current position from each cluster center. It returns the closest cluster in the undiscovered region (cluster with the least Euclidean distance) as the output. Following simple algorithm describes the procedure,

Find-Nearest-Cluster(UnknownCluster list, CurrentPosition)

- Iterate through the clusters while calculating the distance to the current position from the center of the cluster
- Select the closest cluster as the goal.

3.2.1.2. Implementation

The above-explained Goal Identifier subsystem was implemented as a C++ ROS node and a C++ object combination. The ROS node (named “Goal_identifier_node”) acts as an interface between the C++ object (named “IdentifierObject”) and the rest of the motion planner’s subsystems while C++ object do the required calculations to calculate the goal. During the initialization of the ROS node, the object is created with the specified parameters. C++ object contains all the functions required to calculate the goal and those functions are invoked through the services offered by the “Goal_identifier_node”.

3.2.1.2.1. Goal_identifier_node

The ROS node functions as the interface between the IdentifierObject and the rest of the system. Table 3.1 contains details about its subscriptions of topics, publications of topics and advertisements of services.

Table 3.1 goal_identifier_node connections

Type	Name	Callback	Task
Subscribed Topics	/octomap_full	mapCallback	update the Octomap used for calculations
	/odom	currentPosition Callback	update the current position used for calculations
Published Topics	/centerArray	-	Publish center points of undiscovered clusters
	/goalPoint	-	Publish the selected goal
Advertised Services	/goalPosition	executionCallback	find the nearest undiscovered cluster’s center
	/goalRemove	removeCallback	Remove a cluster center

The ROS node uses the general spinner functionality to grab and evaluate the callback queues at a rate of 100Hz. The respective callbacks update the Octomap and position variables of the C++ object at the same rate the callbacks are called.

The values published through “centerArray” and “goalPoint” can be used to visualize the result. They act as a means to visualize the process happening inside the Goal Identifier subsystem and does not contribute to any processes related to goal calculation. centerArray topic publishes ROS messages of the type “pointDataArray” and goalPoint topic publishes messages of the type “pointData”. These custom messages are explained further in section, “Custom messages of Topics and Services”.

The service "goalPosition" offers the functionality to analyze the Octomap and to calculate the goal. The results of the goal calculation, whether it found a goal or not and the coordinates of the goal, are returned as the response of the service call. This service gets requested by the Path Planner subsystem.

The service "goalRemove" offers the functionality to remove a point from the set of cluster centers which acts as candidates for the goal. This service is called when a goal gets flagged as unreachable and want to be removed from future calculations. This service gets requested by the Path Planner subsystem.

Figure 3.4 explains the connections the goal_identifier_node has with the IdentifierObject as well as the external nodes such as Octomap_server node and kobuki node and global_path_planner_node.

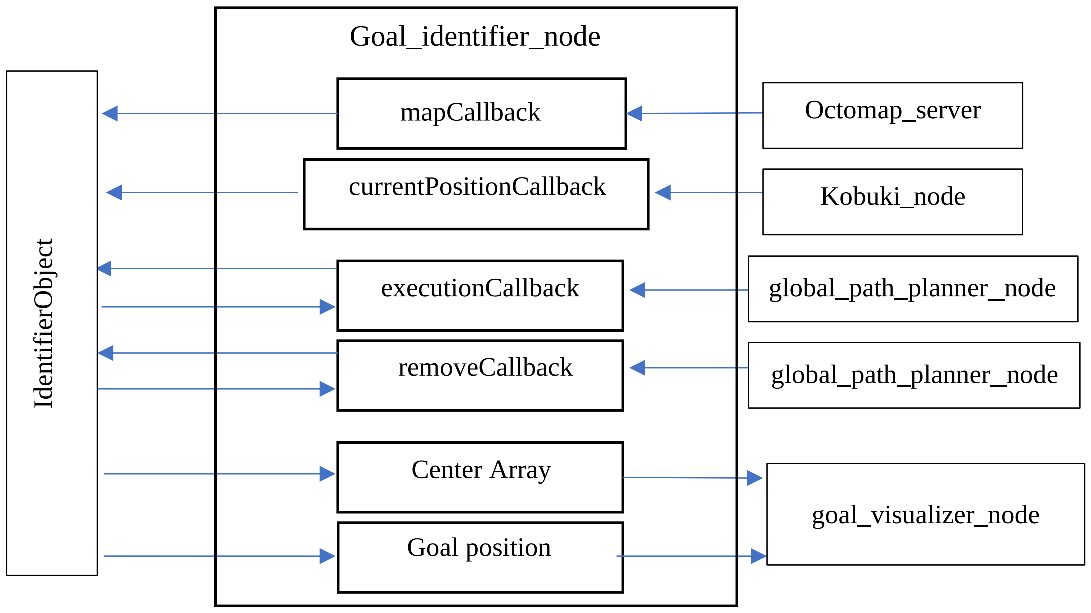


Figure 3.4 Structure of goal_identifier_node interface

3.2.1.2.2. IdentifierObject

The IdentifierObject contains the functions required to calculate the goal. This object gets created at the initialization of the goal_identifier_node and the methods of this object get invoked through the callbacks and service calls available in the goal_identifier_node. Table 3.2 contains the names and the basic functionalities of the functions of this object.

Table 3.2 Functions available in the IdentifierObject

Function	Functionality	Input/output
update_position	Updates the Octomap used for map calculation.	3D point (input)
update_tree	Updates the current position of the robot	Octomap (input)
calculate	Calculates the goal position. Implements Discover-Clusters and Find-Nearest-Cluster functionality described in the design section	3D point (output)
remove	Remove a point from goal candidates	3D point (input)

Figure 3.5 shows the structure of the IdentifierObject,

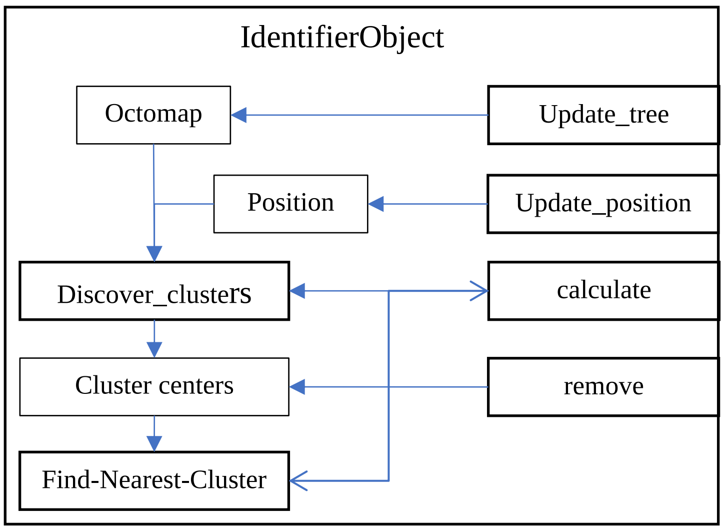


Figure 3.5 IdentifierObject structure

As mentioned earlier, this Goal_identifier_node and IdentifierObject make up the Goal Identifier subsystem. Figure 3.6 explains the final structure of the subsystem,

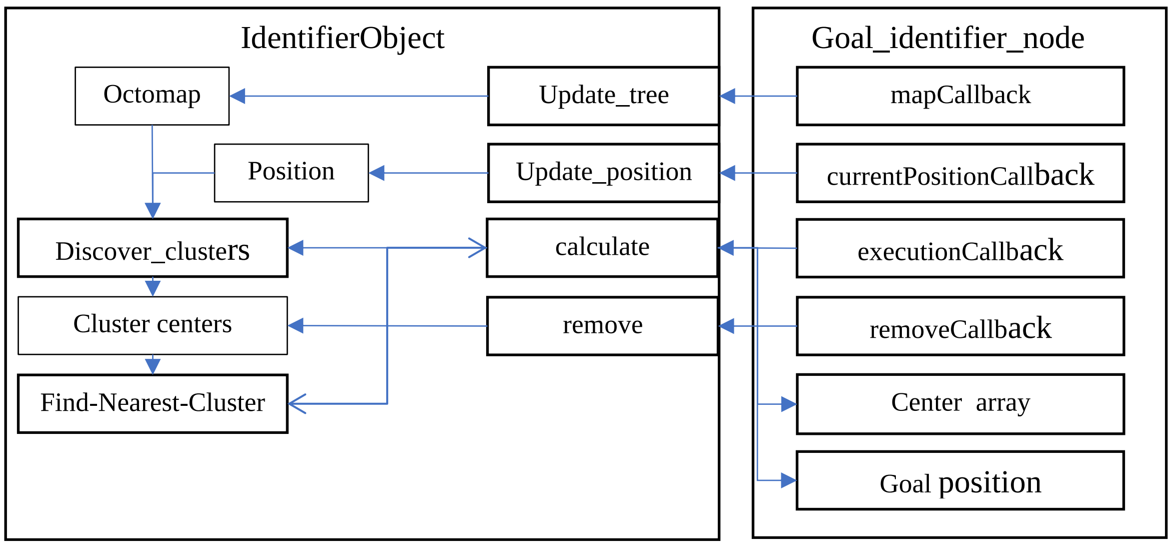


Figure 3.6 Complete structure of Goal Identifier Sub system

3.2.2. Path Planner

Path Planner is the subsystem that calculates an obstacle-free path. It takes a start position, a goal position, and a map that contains obstacles as inputs and calculates a path which is obstacle-free, as output. Generally, motion planners use localization methods to identify the correct position of the robot with respect to the map and then use algorithms such as artificial potential field method or the cell decomposition method to identify available paths. This process converts the map into a graph where paths are represented by edges and intersections are represented by vertices. If there is more than one path, a graph traversal algorithm can be used to select a path.

The Path Planner subsystem proposed in this section uses the current position of the robot as the start point, goal position calculated by the Goal Identifier subsystem as the goal and a map derived from the Octomap generated using the Octomap_server to calculate a traversable path. Localization uses the robot's odometry as well as a map of the ground to correctly calculate the current position of the robot. Odometry data alone is generally considered not reliable enough due to wheel slips and collisions.

The Path Planner has to operate in an unmapped environment and because there are no existing maps of the environment, the Path Planner cannot use localization processes and thus has to rely on odometry data alone. So, reducing the chance of wheel slips as well as collisions that can interfere with odometry readings was one of the main focuses during the designing of this system.

Another main issue that this motion planner faced was the inability to use artificial potential field method or any other algorithm to directly identify a path. Not having a map of the environment was the reason. To solve this, a simple occupancy grid was created after analyzing the Octomap generated by the Octomap_server. This occupancy grid needs to be recalculated every time, before a path calculation, to capture the updated areas in the Octomap. The A-star algorithm was used to calculate a path according to the created occupancy grid.

3.2.2.1. Design

The Path Planner subsystem was designed to move a robot in an unknown environment. To move the robot from one position to another, it needs to calculate an obstacle-free path. To calculate an obstacle-free path, it needs a map with known obstacles marked in it. To build a map with obstacles, the robot needs to analyze the environment around it.

Octomap, generated by the Octomap server provides a probabilistic representation of the surrounding environment and the robot can use it to analyze the surrounding. Then the planner can create a 2D or 3D map with obstacle positions marked in it. The map then can be used to calculate an obstacle-free path for the robot to move.

The above first analysis breaks the problem into smaller parts that depend on each other and highlights the approach one needs to take to solve it. The second analysis proposes an approach that can be followed to solve the problem as explained in the first analysis. This was the basic breakdown of the problem and the bottom-up design of the solution that happened during the path planner design. According to that design, the following steps were recognized as necessary to create the Path Planner.

1. Analyzing the Octomap
2. Build a 2D map with obstacles.
3. Inflate the obstacles
4. Calculate a path

3.2.2.1.1. Analyzing the Octomap

The Octomap can be analyzed using two approaches,

- Using the node iterator provided by the Octomap framework to iterate along the Octomap and build a 2D grid with occupancy details.
- Assuming a 3D grid over the area in the Octomap where we need to analyze and iterating through it and extracting information to build the 2D grid with occupancy details.

Out of the two approaches, the first approach works on the octree structure on which the Octomap is based on. Due to this reason, it returns nodes according to the octree structure and the system has to filter out the points which are outside the area subjected to the analysis. The second approach directly checks whether the points inside the specified area are discovered and occupied. The Path Planner uses the second approach. Octomap analyzing can be done in two stages,

- Surrounding analysis
- Ground analysis

Surrounding analysis evaluates the Octomap nodes from the ground level up to the height of the robot. This makes sure that the robot can move through the underpasses that are higher than the robot height. The planner looks for Octomap nodes that are occupied and then marks them on the 2D grid as obstacles. In this analysis, occupied nodes in the Octomap represents obstacles in the real world.

The ground analysis evaluates the Octomap nodes which are at the ground level. These nodes represent the actual ground and the planner looks for nodes that are unoccupied. Occupied nodes during ground analysis mean that the floor exists, and unoccupied nodes mean floor does not exist (pits and falls). So, the planner looks for unoccupied nodes during ground analysis and marks them on the 2D grid as obstacles.

3.2.2.1.2. Building the 2D grid

As mentioned in the above section, a 2D grid is used to store data about the obstacles in the real world. Since the obstacles found in the Octomap are marked on the 2D grid, the grid was designed with twice the resolution of the Octomap (i.e. if the Octomap's resolution was 400 cubic nodes per side each with 0.05m side, the grid has a resolution of 800 square nodes per side each with 0.025m side). The grid values can have 2 states,

- Occupied
- Not occupied

Obstacles are represented by the occupied state and free space is represented by the unoccupied state. One of the issues that the planner had was that the goal was almost always got selected from outside of the discovered region. Due to that, the path calculation faced the issue of having to measure the distance to nonexistent points. The solution was to assume that the whole grid was traversable with no obstacles and then add the obstacles gradually rather than adding occupied and unoccupied points at once. During the Octomap analysis, the grid gets updated as explained in the above section.

3.2.2.1.3. Inflating the obstacles

After building the 2D grid as explained in the above sections, before a path can be calculated using that 2D grid, it needs to be modified so that the robot can navigate without colliding with obstacles. Path calculation considers the robot as a point object. Due to this, if the 2D grid created by analyzing Octomap was directly used for path calculation, the robot becomes unable to go near any obstacles or go around any obstacles because the width of the robot has not been factored into the calculation.

To factor-in the robot width as well as to keep the calculation of the path as it is, the obstacles in the 2D grid need to be inflated by at least half the width of the robot. The blue areas in figure 3.7 represent the obstacles and the green areas in figure 3.7 represent the inflation done in order to account for robot width.

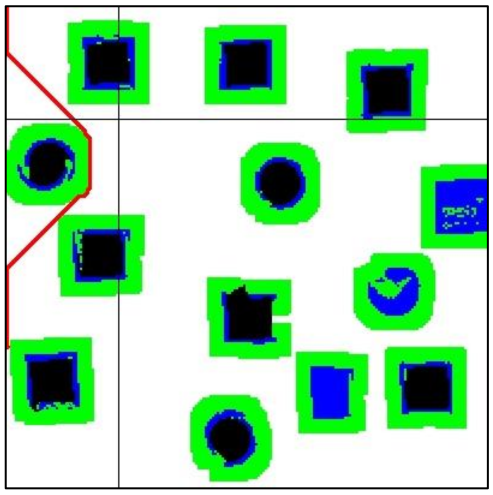


Figure 3.7 Inflation (green) of obstacles (blue)

3.2.2.1.4. Calculate a path

After inflating the obstacles, the path planner can proceed to calculate a path. The grid becomes a graph where each value represents a vertex. All the vertices are connected with each other and each edge has a cost of ‘1’. Values with the occupied state become invalid vertices and are considered unreachable while the values with unoccupied state become vertices in the graph that can be reached.

A graph traversal algorithm can be used to calculate a path. The motion planner uses the A-star algorithm because it is capable of finding a path if one exists. If the path does not exist, the algorithm returns a Null value signifying an issue with the 2D grid. If the issue is with the source then the robot needs to be moved to an adjacent free location, if the goal was not valid then it can be removed as an inaccessible goal. The source can be affected due to map updates that cause the inflation area to encompass the robot’s current position.

After successfully calculating the path, it needs to be converted into real-life position values from the respective grid values. After that, it can be fed to the velocity controller subsystem to move the robot along the path.

3.2.2.2. Implementation

This subsystem is implemented as a ROS node (named “global_path_planner_node”) and a C++ object (named “plannerObject”) combination. global_path_planner_node acts as the interface between the plannerObject and the rest of the motion planner. In addition to an interface, the global_path_planner_node also acts as the control unit of the whole motion planner. During the initialization of the global_path_planner_node, the plannerObject is created with the specified parameters. It contains all the functions required for the path calculation and those functions are invoked through the services offered by the “global_path_planner_node”. The control unit of the motion planner is also implemented inside the global_path_planner_node as a service and an external program can start the process through a service call.

3.2.2.2.1. Global_path_planner_node

The ROS node functions as the interface between the plannerObject and the rest of the system as well as the control unit of the whole system. Table 3.3 contains the details about its subscriptions of topics, publications of topics and advertisements of services as well as usage of services.

Table 3.3 global_path_planner_node connections

Type	Name	Callback	Task
Subscribed Topics	/octomap_full	mapCallback	update the Octomap used for calculations
	/odom	currentPositionCallback	update the current position used for calculations
Published Topics	/gridMap	-	Publish 2D grid used to calculate the path
Advertised Services	/explore	systemCallback	Path calculation and exploration
Requested Services	/baseRotate	rotateClient	Rotate robot around z axis
	/baseReverse	reverseClient	Move robot backward
	/baseForward	forwardClient	Move robot forward

	/goalPosition	clientGoalPosition	find the nearest undiscovered cluster’s center
	/goalRemove	clientGoalRemove	Remove a specific cluster center

The ROS node uses the asynchronous spinner functionality to grab and evaluate the callback queues at a rate of 100Hz. An asynchronous spinner can be used to assign each callback queue a single processing thread, allowing parallel processing of callback messages unlike in general spinner functionality. A general spinner processes all the callback queues sequentially under a prespecified frequency. This can cause a loss of messages which can be harmful in cases where velocity control and position tracking is involved. The asynchronous spinner allows the node to process messages as they arrive. Parallel processing of callback queue messages can cause a “Race Condition” when updating the variables of plannerObject. It can be prevented by using a mutex inside callbacks where the plannerObject variables are updated.

The mapCallback updates the map of the plannerObject while the currentPositionCallback updates the position of the plannerObject. These two callbacks function as the first part of the interface of the ROS node. Other parts of the interface consist of the clients for the services offered by Goal_identifier_node and the velocity_control_node. Clients for the services goalPosition and goalRemove control the calculation and removal of goal while the clients for the services baseForward, baseReverse, baseRotate controls the robot movement for the plannerObject.

Figure 3.8 explains the connections of the global_path_planner_node with other nodes of the motion planner as well as other nodes as kobuki node and Octomap server node. “External control” block represents the external program used to start the mapping process as explained below.

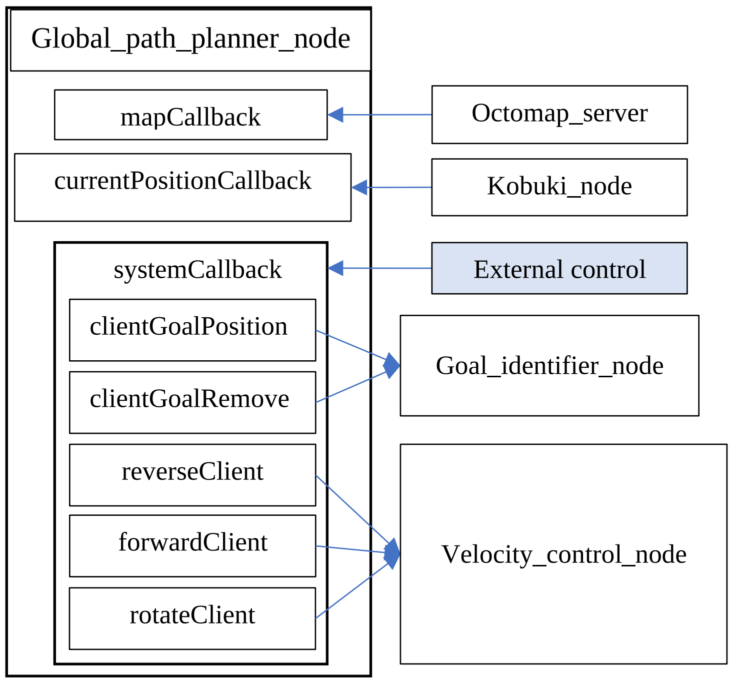


Figure 3.8 Structure of path_planner_node interface

The path planning and navigation process are triggered by the service “explore”. Rather than starting this as the main process at the beginning of the system load, it is implemented as a ROS service so that the user has more control over the system before starting the navigation and planning process. Once the service gets a request from the controller (represented in figure 3.8 by the “External control” block), it starts the process and continues till the whole map is explored and the robot has returned to the starting point. Then it responds with a “success” notification to the controller. The callback of the “explore” service, “systemCallback” acts as the main control unit of the motion planner by connecting with other subsystems of the motion planner through the global_path_planner_node’s interface as explained in figure 3.8.

Publishing the developed 2D grid using the “gridMap” topic lets the user an idea about the progress of the mapping progress. It uses custom ROS messages “gridMap”, “gridRow”, and “gridPoint” to publish the objects, inflation area, discovered area and path separately. The usage has been explained in detail in section “Custom messages of Topics and Services”.

Figure 3.9 illustrates the decision sequence used inside the systemCallback which acts as the control unit of the motion planner.

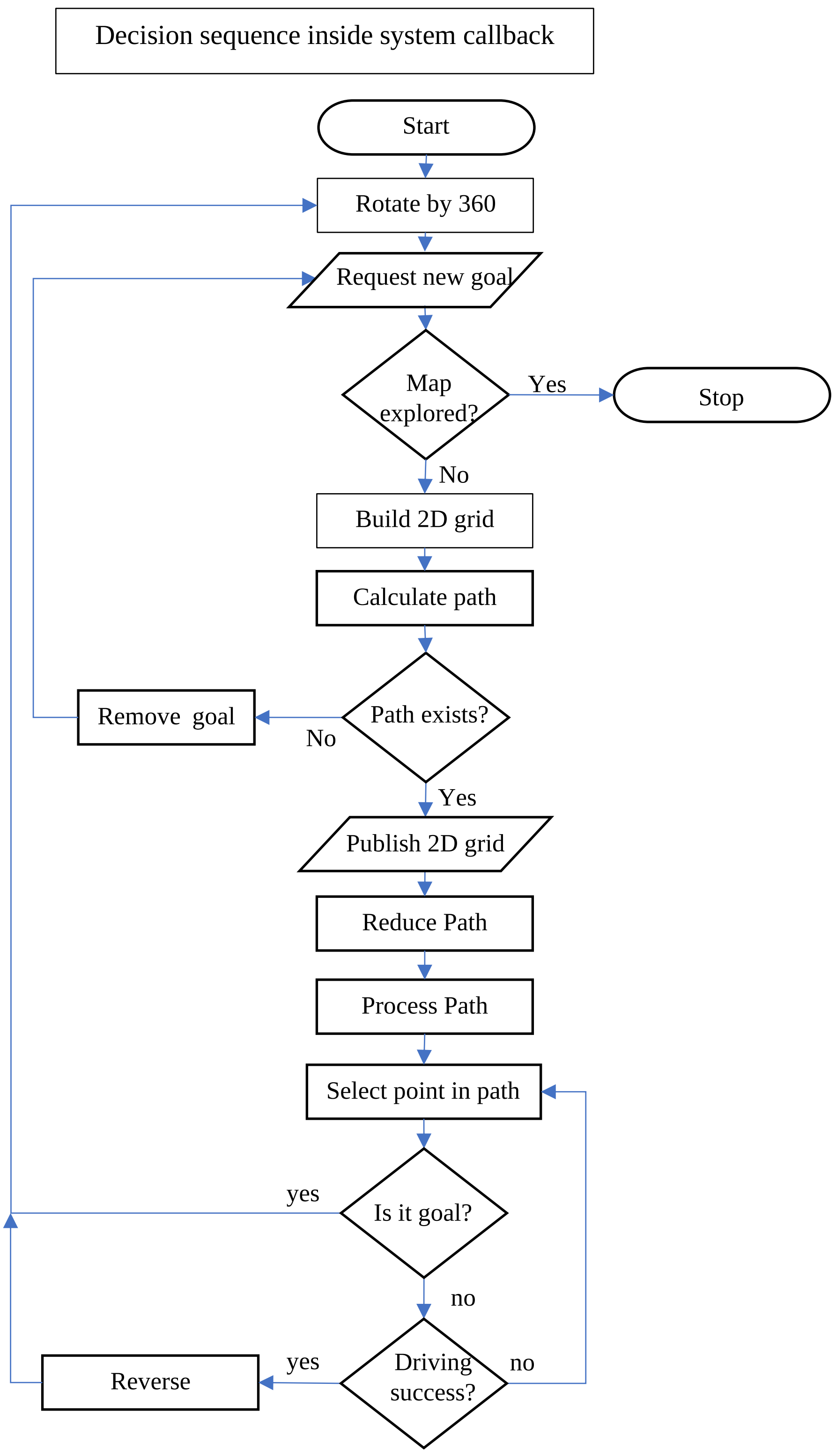


Figure 3.9 Decision sequence inside systemCallback

3.2.2.2.2. plannerObject

The plannerObject contains the functions required to calculate the path as explained in the design section. This object gets created at the initialization of the global_path_planner_node and the methods of this object get invoked by the ROS node during callbacks and service calls. Table 3.4 contains the names and the basic functionalities of the functions of this object.

Table 3.4 PlannerObject functions and functionality

Function	Functionality	Input/output
update_position	Updates the Octomap used for map calculation.	3D point (input)
update_goal	Updates the goal used for path calculation	3D point (input)
update_tree	Updates the current position of the robot	Octomap (input)
search	Calculates the path using the 2D grid	2D Grid (input) 2D points (output)
buildMap	Builds the 2D map used for path calculation according to the procedure explained in the section Design	Octomap (input) 2D grid (output)
processPath	Converts 2D grid point into real-world 3D Points for robot's navigation	2D array (input) 3D array (output)
isBlocked	Checks whether a point is inside an occupied are. Used to check whether source in the inflated area	3D point (input) Boolean (output)
reducePath	Reduces points in the path	3D array (input) 3D array (output)
nearestUnblocked	Remove a point from goal candidates	3D point (input)

Figure 3.10 shows the implementation of the "buildMap" function which is responsible for creating the 2D Grid used for path calculation. From the Octomap it extracts obstacles by iterating through the Octomap nodes. It also records the area that has been explored. After extracting the obstacles, it inflates them as explained in the previous Section 3.2.2.1.3. Then the area gets filtered with the discovered area. Only the obstacles in the discovered area are used to calculate the path.

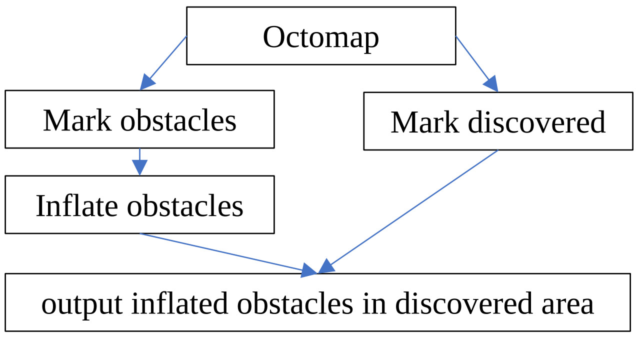


Figure 3.10 Building the 2D grid for navigation

Figure 3.11 shows the combined system of global_path_planner_node and plannerObject.

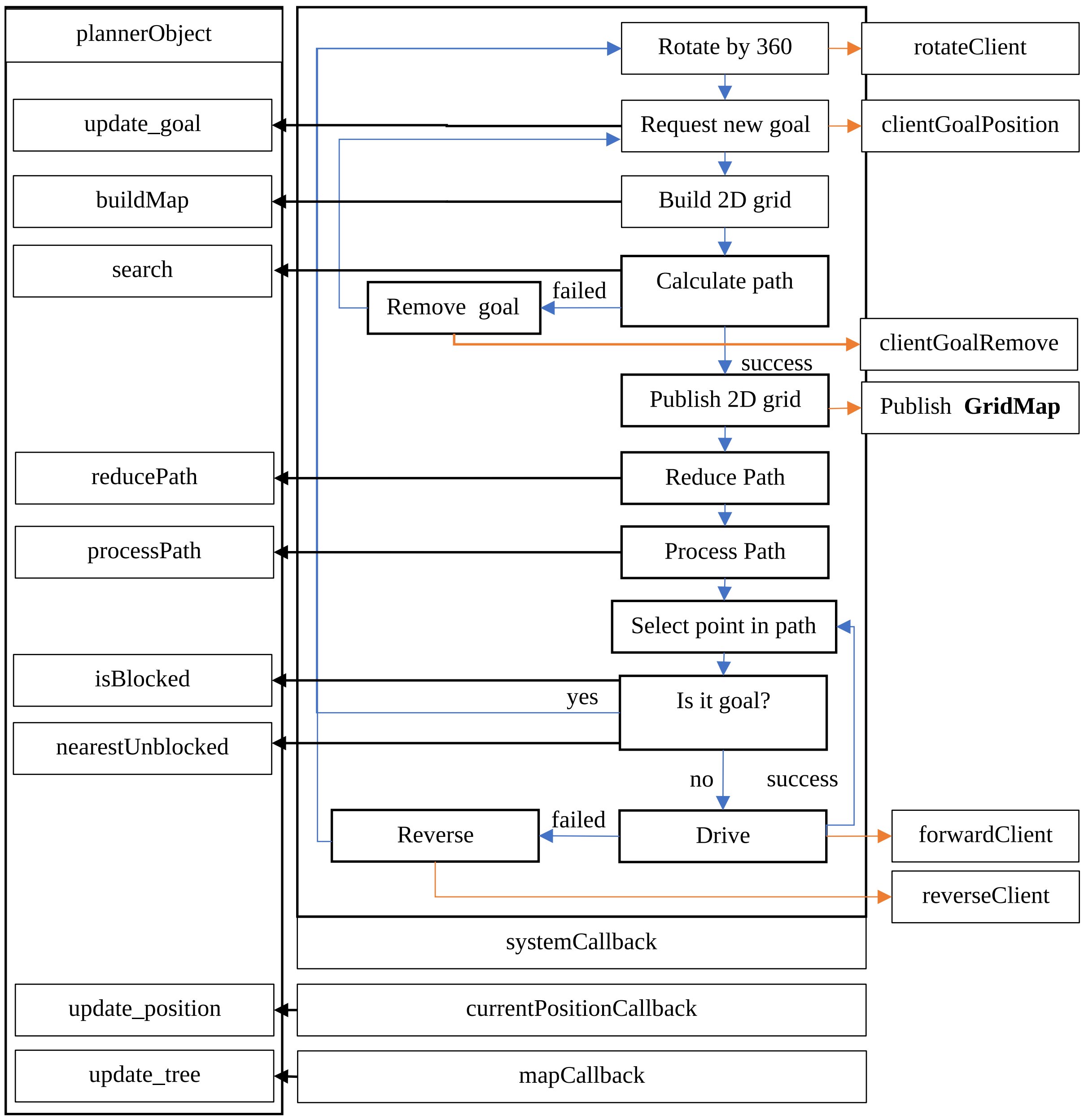


Figure 3.11 Combined path_planner_node and plannerObject

3.2.3. Ground Evaluator

This subsystem operates on images capture through the main camera and the depth camera of the robot and acts as a supporting system to the Path Planner subsystem. The current system is supposed to use images from the Kinect mounted on the Kobuki robot.

Due to the increased complexity of the Path Planner subsystem and the lack of time, the integration of the Ground Evaluator subsystem to the motion planner system did not occur. The development and testing of the basic system to detect mud puddles completed. The development of the ROS node which acts as the interface between the motion planner and basic system did not occur.

3.2.3.1. Design

The main requirement of this subsystem was the ability to work in environments that it has never been to. To achieve this, it was decided not to use neural networks or other machine learning-based approaches. The solution decided was to use the image cues [10] as proposed by Rankin et al. After analyzing their results, it was decided that the main image cues,

- texture
- color
- range reflection

HSV image data and the greyscale image can be used to analyze the texture of the image. After running a variance filter on them, pixels with saturation values and greyscale values larger than 25 can be used for further processing as described by Rankin et al. A boolean mask containing recognized points with water needs to be generated for texture cue named “textureMask”.

HSV image data can be used to analyze the color of the image. The sky visibility in the image has the chance to interfere with the analysis and Rankin et al. have proposed an extended evaluation of HSV data depending on the availability of the sky. Another boolean mask containing recognized points with water needs to be generated for color cue named “colorMask”.

The last cue is range reflections. According to Rankin et al. this is a phenomenon that occurs in the depth image when the reflections of the far objects are visible on the reflective surfaces such as mud puddles. Inflection points in the depth image columns can be used to recognize areas with range reflections. A third boolean mask needs to be generated for range reflections cue named “rangeMask”.

A final Boolean mask can be calculated by using the above 3 masks. It can be generated according to the final results Rankin et al. has presented. Equation 3.1 shows the relationship between the 3 masks and the final mask.

Equation 3.1 Calculation of water detection mask

$$finalMask = (rangeMask) \mid (\overline{rangeMask} \,\&\, colorMask \,\&\, textureMask)$$

3.2.3.2. Implementation

This system has been developed as a python script. Each of the above-mentioned image analysis happens inside user-defined functions and all those are called upon by a single function. Figure 3.12 explains the structure of the system.

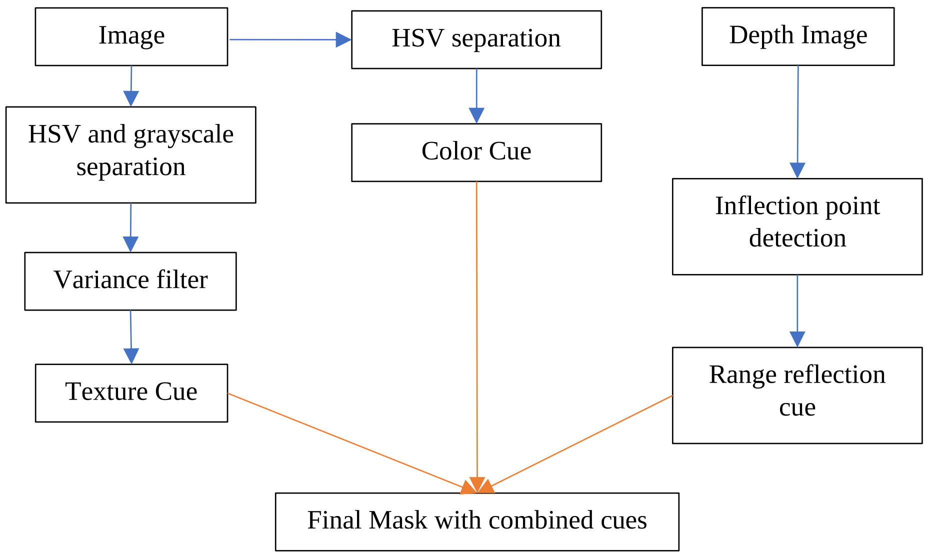


Figure 3.12 Structure of ground evaluator

3.2.4. Velocity Controller

This subsystem acts as the interface between the motion planner and robot motion control system. It converts the position data calculated by the Path Planner into velocity commands compatible with the robot base. This subsystem acts as the local planner of the motion planner system.

3.2.4.1. Design

To fulfill the requirements of the motion planner, the robot needs to be able to rotate around its z-axis, move forward and move backward. Following pseudocodes describes the steps taken to achieve each motion.

Moving forward to a given point requires the robot to first rotate around its z-axis to face the correct direction and then to move forward to reach the point. Both of these actions need to happen in small increments so that the robot can move without errors.

Forward (position, currentPosition, currentYaw)

1. Calculate the angle that needs to be achieved to face the correct direction as shown in equation 3.2.

Equation 3.2 Calculation of Yaw

$$yaw = tan^{-1}\left(\frac{position.y\ -\ currenPosition.y}{position.x\ -\ currentPosition.x}\right)$$

2. Rotate around the z-axis to achieve the desired yaw. Angular velocity is calculated proportionally to the remaining angle to be rotated as shown in equation 3.3. The maximum rotational velocity ω_{max} has been capped to prevent odometry errors that can occur due to high rotational speeds. k_{ω} is a propositional constant.

Equation 3.3 Calculation of Angular Velocity

$$\omega_z = \min\{\ \omega_{max},\ k_{\omega}(yaw - currentYaw)\}$$

3. Repeat steps 1 and 2 till the $(yaw - currentYaw)$ the value becomes neglectable.

4. Calculate distance to the position from the current position as shown in equation 3.4.

Equation 3.4 Calculation of Distance

$$distance = \sqrt{\begin{array}{l}(position.x - currentPosition.x)^2 + \\ (position.y - currentPosition.y)^2 + \\ (position.z - currentPosition.z)^2\end{array}}$$

5. Move forward to travel the calculated distance. Forward velocity is calculated proportionally to the remaining distance to be traveled as shown in equation 3.5. The maximum velocity V_{max} has been capped to prevent odometry errors due to high rotational speeds. k_x is a propositional constant.

Equation 3.5 Calculation of Velocity

$$V_x = \min\{V_{max}, k_x * distance\}$$

6. Repeat steps 4, 5 until $(distance)$ becomes neglectable.

Unlike moving forward, moving backward only requires the robot to travel backward a specified distance. This distance is calculated relative to the resolution of the Octomap since it is only needed when the robot runs into an obstacle not marked or falsely marked on the Octomap. Here also the movement is achieved through small increments rather than moving at once to reduce odometry issues.

<u>Backward (currentPosition)</u>

1. Mark the current position as startPosition.
2. Calculate distance to the startPosition from the current position as shown in equation 3.6.

Equation 3.6 Calculation of Distance

$$distance = \sqrt{\begin{array}{l}(startPosition.x - currentPosition.x)^2 + \\ (startPosition.y - currentPosition.y)^2 + \\ (startPosition.z - currentPosition.z)^2\end{array}}$$

3. Move backward to travel the specifiedDistance distance. Backward velocity is calculated proportionally to the remaining distance to be traveled as shown in equation 3.7. The maximum velocity V_{max} was capped to prevent odometry reading errors due to high rotational speeds. k_x is a propositional constant.

Equation 3.7 Calculation of Backward Velocity

$$V_x = \max\{-1 * V_{max}, -1 * k_x * (specifiedDistance - distance)\}$$

4. Repeat steps 2, 3 until $(specifiedDistance - distance)$ becomes neglectable.

The last action is the rotation around the z-axis, and it is performed in order to update the map. To update the map evenly, the robot base rotates in a constant rotational velocity, rather than calculating the rotational velocity proportional to the angle to be rotated. To prevent the robot from rotating more than 360 degrees, robot rotation happens in small increments rather than rotating at once.

Rotate (desired angle)

1. Calculate the new yaw value considering the current yaw and the angle to be rotated as shown in equation 3.8. If this value becomes more than 2π radians, it needs to be converted into a value between 0 and 2π.

Equation 3.8 Calculation of Yaw

$$yaw = currentYaw + desiredAngle$$

2. Rotate around z-axis to achieve yaw. Since the Angular velocity is constant as in equation 3.9, the only requirement is to make sure that the robot does not exceed the specified yaw.

Equation 3.9 Calculation of Angular Velocity

$$\omega_z = \omega_{rotate}$$

3. Repeat steps 1, 2 until $(yaw - currentYaw)$ value neglectable.

3.2.4.2. Implementation

The velocity controller subsystem consists of a single ROS node called "velocity_control_node". It converts position commands calculated by the Path Planner subsystem into velocity commands that can be recognized by the robot motion controller. It also listens to bumper events published by the robot in order to monitor collisions that can occur. It controls the base of the robot through ROS topics and offers movement functionality to path planner as ROS services. Table 3.5 contains the details about the services the velocity_control_node offers as well as the topics it subscribes to and topics it publishes,

Table 3.5 Velocity_control_node connections

Type	Name	Callback	Task
Subscribed Topics	/mobile_base/events/bumper	bumperCallback	Checks whether the robot collides with obstacles
	/odom	currentPosition Callback	update the current position used for calculations
Published Topics	/mobile_base/commands /velocity	-	Velocity commands to the robot base
	/mobile_base/commands /motor_power	-	Power commands to the robot base
Advertised Services	/baseRotate	rotateCallback	Rotate robot around z axis
	/baseReverse	reverseCallback	Move robot backward
	/baseForward	driveCallback	Move robot forward

The ROS node uses the asynchronous spinner functionality to grab and evaluate the callback queues at a rate of 100Hz. Asynchronous spinner can be used to assign each callback queue a separate processing thread allowing parallel processing of callback messages unlike in general functionality. This becomes important when keeping track of robot position and velocity. General spinners process all the callback queues at once, sequentially with a fixed frequency that can be harmful to velocity control and position tracking. But the asynchronous spinner allows the node to process callback queue messages as they arrive.

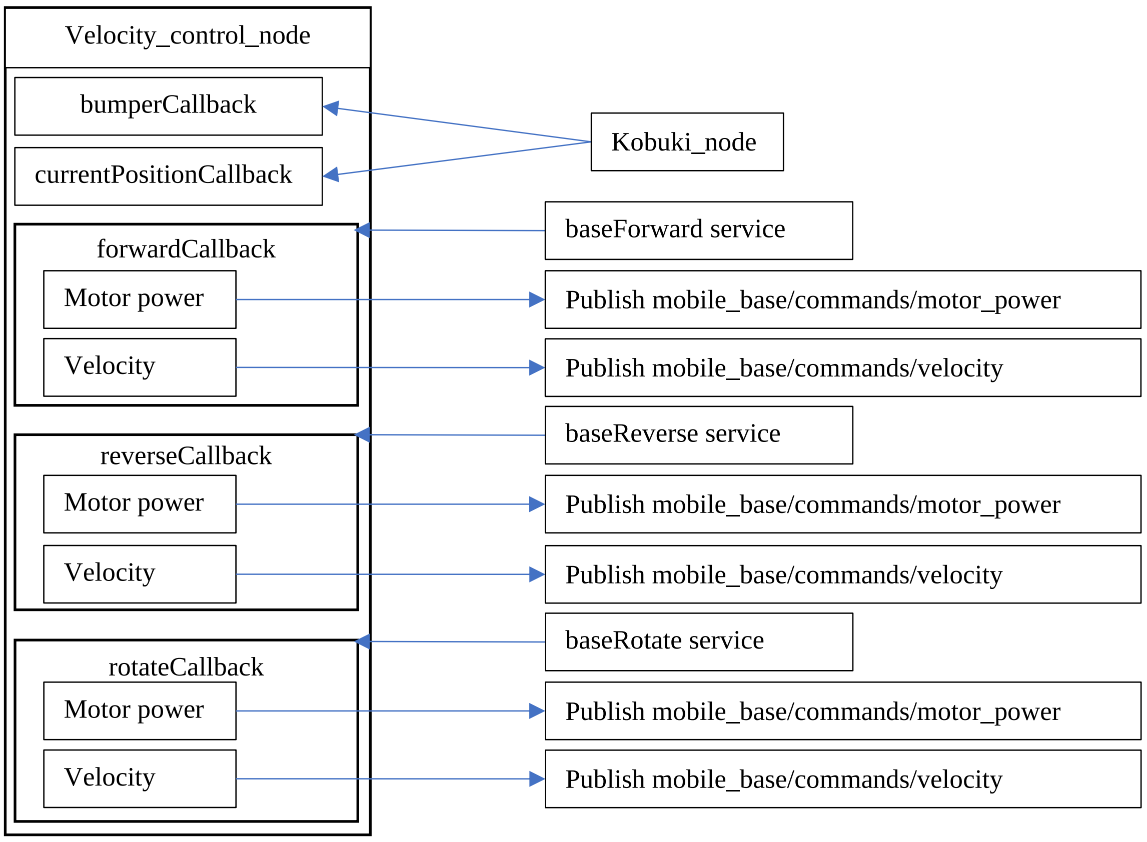
The driveCallback implements the above procedure named “forward” and responds to the service “baseForward” which offers to move the robot forward to a given point. The point to be moved is included in the request of the service “baseDriveRequest” while the completion of the operation is included in the response of the service “baseDriveResponse”.

The reverseCallback implements the “reverse” procedure explained above and responds to the “baseReverse” service which offers to move the robot backward a fixed distance. The current position is included in the request of the service “baseDriveRequest” while the completion of the operation is included in the response of the service “baseDriveResponse”.

The rotateCallback implements the “rotate” procedure explained above and responds to the “baseRotate” service which offers to rotate the robot around its z-axis. Angle to be rotated is included in the request of the service “baseRotateRequest” while the completion of the operation is included in the response of the service “baseRotateResponse”.

The velocity_control_node subscribes to the odometry topic and bumper event topic published by the robot base. From the odometry readings, it derives the current position of the robot as well as the current yaw of the robot for calculations related to the base movement. Bumper events indicate whether the base has collided with an obstacle or whether it is free to move. If the bumper events get triggered while the robot is on the move, the velocity_control_node stops all movements and waits for the Path Planner subsystems response. Current Path Planner implementation responds with a move back command.

The robot calculates the velocity according to the position and yaw of the robot and then publishes the respective velocity commands as well as the power on/ off commands to the robot base through the “mobile_base/cmd_vel” topic and the “mobile_base/motor_power” topics. Figure 3.12 illustrates the structure of the velocity control node as well as its connections with external subsystems and nodes.



Velocity_control_node
bumperCallback
currentPositionCallback
Kobuki_node
forwardCallback
Motor power
Velocity
baseForward service
Publish mobile_base/commands/motor_power
Publish mobile_base/commands/velocity
reverseCallback
Motor power
Velocity
baseReverse service
Publish mobile_base/commands/motor_power
Publish mobile_base/commands/velocity
rotateCallback
Motor power
Velocity
baseRotate service
Publish mobile_base/commands/motor_power
Publish mobile_base/commands/velocity

Figure 3.13 Structure of velocity_control_node

3.2.5. Custom messages used in Topics and Services

ROS uses topics, services and actions as a communication medium to communicate between ROS nodes. ROS topics utilize basic ROS messages which can only be sent in one direction, from the publisher to the subscriber. These messages can be empty or can carry data. ROS services use more complex message pairs which are called “Requests” and “Responses”. Request messages are sent from the client node to the server node and contain data required for the server-side process. After completing the process, the server replies with a Response message that contains the results of the server-side process. Both Request and Response can be empty or can carry data as mentioned above.

During the implementation of the Motion Planner, custom messages, service requests, and service responses had to be created in order to fulfill the needs of the subsystems of the motion planner.

3.2.5.1. Messages

The current motion planner system contains 5 types of messages to fulfill 2 requirements. pointData and pointDataArray messages focus on transferring a 3D point array while the gridPoint, gridRow, and gridMap messages focus on transferring a 3D matrix.

3.2.5.1.1. pointData

This message contains 3 data fields of basic ROS data type “float32”. The data fields as shown in table 3.6 can be used to store x, y, z coordinates of any 3D point.

Table 3.6 pointData message structure

Data field type	Data field name	Purpose
Float32	x	Hold X coordinate
Float32	y	Hold Y coordinate
Float32	z	Hold Z coordinate

3.2.5.1.2. pointDataArray

This message contains 1 data field of custom ROS message type “pointData”. It has been implemented as an array and can be used to store messages of type pointData. This creates a message with a 3D point data array that fulfills the first requirement. The goal_identifier_node uses pointData and pointDataArray messages to send the goal and center points of the unmapped clusters to the goal_visualizer_node for visualizing. Table 3.7 explains the fields of the pointDataArray message.

Table 3.7 pointDataArray message structure

Data field type	Data field name	Purpose
pointData[]	centerPointsArray	Contain point data array

3.2.5.1.3. gridPoint

This message contains 3 data fields of basic ROS data type int8. These data fields can be used to store data related to a single point in the 2D grid used for path calculation. Table 3.8 explains the fields of the gridPoint message.

Table 3.8 gridPoint message structure

Data field type	Data field name	Purpose
Int8	init	Details about obstacles
Int8	proc	Details about inflation
Int8	disc	Details about map state

3.2.5.1.4. gridRow

This message contains 1 data field of custom ROS message type gridPoint. It has been implemented as an array and can be used to store gridPoint messages. This message represents a row in the 2D grid map used for path calculation. The gridPoint messages inserted into this array message should contain data from the points from the same row in the 2D grid. Table 3.9 explains the fields of the pointDataArray message.

Table 3.9 gridRow message structure

Data field type	Data field name	Purpose
gridPoint[]	row	Contain points in a row

3.2.5.1.5. gridMap

This message contains 3 data fields. They are the custom ROS message type "gridRow", custom ROS message type "pointData" and basic ROS data type "int16". The first field has been implemented as an array and can be used to store gridRow messages. This data field represents the 2D grid map used for path calculation and the gridRow messages inserted into this array message should contain data from the rows of the 2D grid. This creates a data field capable of capturing a whole matrix at once fulfilling the second requirement.

The second field implements an array of pointData messages which can be used to insert the path calculated by the Path Planner into the message. The implementation is similar to the pointDataArray implementation explained in the above section. The last field contains a single variable of type int16 that carries the length of the array in the second field. These fields are explained in table 3.10.

Table 3.10 gridMap message structure

Data field type	Data field name	Purpose
gridRow[]	grid	Contain 3D matrix
pointData[]	path	Contain calculated path
Int16	pathLength2	The length of the path

3.2.5.2. Service requests and responses

The motion planner system contains 5 types of request and response message pairs which are used in 6 services. goalControl and goalRemove are related to the Goal Identifier subsystem and baseForward, baseReverse, and baseRotate are related to the Velocity Controller subsystem.

3.2.5.2.1. goalControl

The goalPosition service offered by the goal_identifier_node uses this message pair to communicate with its client in the global_path_planner_node. goalControl request carries the command to start the goal calculation and the goalControl response carries the results of the server-side process, whether the goal calculation was successful at finding a goal and if so its coordinates. Table 3.11 contains the fields related to each component response and request.

Table 3.11 goalControl service message structure

	Data field type	Data field name	Description
goalControl request	bool	execute	Command to start server-side process
goalControl response	bool	isNull	Found the goal or not
	Float32	x	X coordinate
	Float32	y	Y coordinate
	Float32	z	Z coordinate

3.2.5.2.2. goalRemove

The goalRemove service offered by the goal_identifier_node uses this message pair to communicate with its client in the global_path_planner_node. goalRemove request carries the coordinates of the center point to be removed. goalControl response carries the results of the server-side process, whether the goal remove was successful or not. Table 3.12 explains the fields of the request and response related to goalRemove.

Table 3.12 goalRemove service message structure

	Data field type	Data field name	Description
goalControl request	Float32	x	X coordinate
	Float32	y	Y coordinate
	Float32	z	Z coordinate
goalControl response	bool	success	Success of removal

3.2.5.2.3. baseDrive

The baseForward and baseReverse services offered by the velocity_control_node uses this message pair to communicate with its clients in the global_path_planner_node. baseDrive request carries the coordinates of the position the robot has to reach and the baseDrive response carries the results of the movement, whether the robot was able to reach the point or not. Table 3.13 explains the fields of the response and request related to baseDrive.

Table 3.13 baseDrive service message structure

	Data field type	Data field name	Description
baseDrive request	Float32	x	X coordinate
	Float32	y	Y coordinate
	Float32	z	Z coordinate
baseDrive response	bool	success	Success of reaching

3.2.5.2.4. baseRotate

The baseRotate service offered by the velocity_control_node uses this message pair to communicate with its client in the global_path_planner_node. baseRotate request carries the angle the robot base has to rotate and the baseDrive response carries the results of the server-side process, whether the robot was able to rotate or not. Table 3.14 explains the fields of the response and request related to baseRotate.

Table 3.14 baseRotate service message structure

	Data field type	Data field name	Description
baseRotate request	Flaot64	angle	Angle to rotate
baseRotate response	bool	success	rotated or not

3.2.5.2.5. systemControl

The explore service offered by the global_path_planner_node uses this message pair to communicate with its client in the test_system_node. systemControl request carries the command to start the navigation and path planning process in the global_path_planner_node and the systemControl response carries the results of the server-side process, whether the whole area was successfully explored or not. Table 3.15 explains the fields of the response and request related to systemControl.

Table 3.15 systemControl service message structure

	Data field type	Data field name	Description
systemControl request	bool	activate	Command to start exploring
systemControl response	bool	success	rotated or not

4. Experimental Evaluation and Results

The system testing was completed using the Gazebo physics simulator [22] along with the ROS system. Kobuki Turtlebot robot model was used because its real-world robot and the simulation robot model both had a similar control interface. Figure 4.1 and figure 4.2 contains the Kobuki Turtlebot and figure 4.3 contains the simulated testing environment.

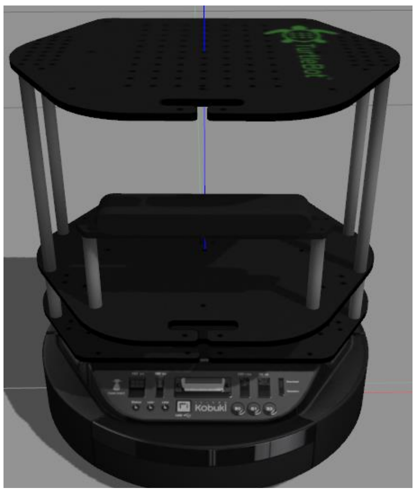


Figure 4.1 Back of Kobuki (virtual)

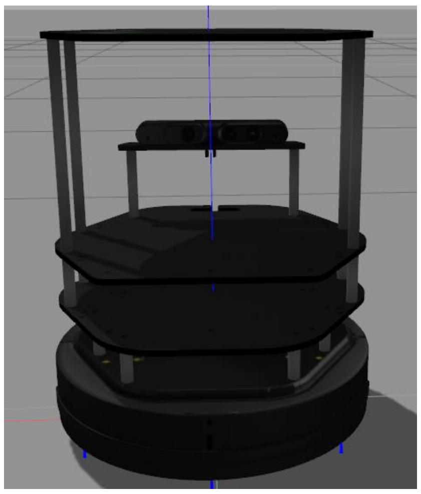


Figure 4.2 Front of Kobuki (virtual)

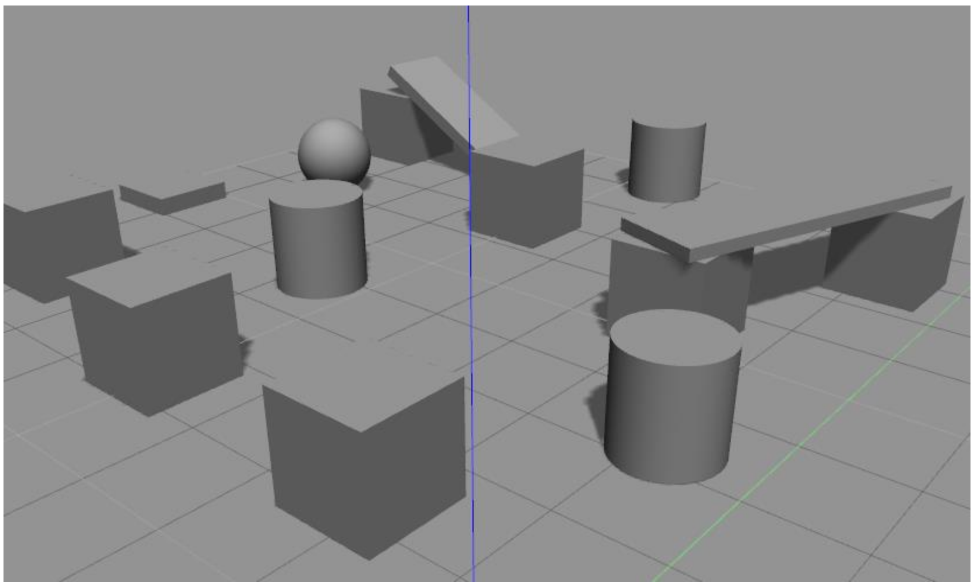


Figure 4.3 Gazebo Virtual testing environment

4.1. Goal Identifier Testing

A support system named “Goal Visualizer” was developed for the purpose of testing the Goal Identifier subsystem and visualizing the results. It contained an individual ROS node “test_goalIdentifier_node” as well as a ROS node & C++ object combination named “Goal_visualizer_node” and “VisualizerObject”.

test_goalIdentifier_node is responsible for calling the “goalPosition” service and starting the Octomap analyzing the process. Goal_visualizer_node subscribes to the “centerArray” and “goalPoint” topics and then transfers those data to the VisualizerObject C++ object. The VisualizerObject converts those data into an Octomap and returns it back to the Goal_visualizer_node which publishes the Octomap under the topic “octomap_centers”.

Table 4.1 contains the connections of the ROS node test_goalIdentifier_node with other nodes of the system,

Table 4.1 test_goalIdentifier_node connections

Type	Name	Callback	Task
Requested Services	/goalPosition	-	Request goalPosition service

Table 4.2 contains the connections of the ROS node goal_visualizer_node with other nodes of the system,

Table 4.2 goal_visualizer_node connections

Type	Name	Callback	Task
Subscribed Topics	/centerArray	arrayCallback	update the Octomap with centerArray
	/goalPoint	goalCallback	update the Octomap with goal
Published Topics	/octomap_centers	-	Publish the created Octomap

Table 4.3 contains the functions of the C++ object VisualizerObject,

Table 4.3 functions and functionality of VisualizerObject

Function	Functionality	Input/output
update_cluster_centers	Updates the Octomap with new cluster centers	3D point array (input)
update_nearest_cluster	Updates the Octomap with goal	3D point (input)
get_tree	Creates the Octomap and returns it	Octomap (output)

Figure 4.4 visualizes the combined structure of the VisualizerObject & Goal_visualizer_node testing support system,

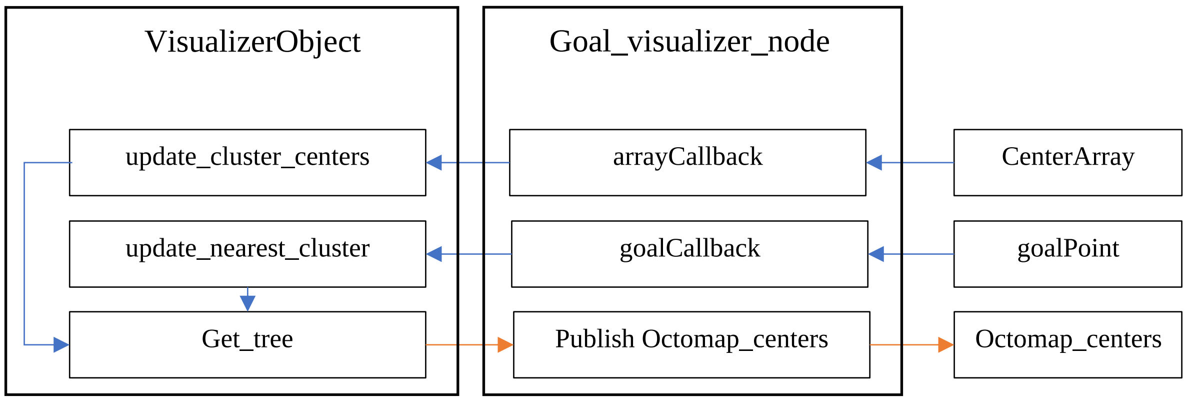


Figure 4.4 Combined system of VisualizerObject and goal_visualizer_node

Figure 4.2 shows the resulting Octomap after processing the center point array and goal position inside the VisualizerObject.

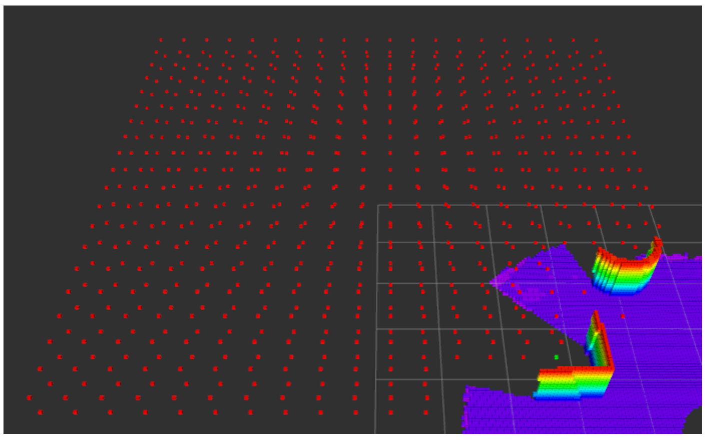


Figure 4.5 Undiscovered clusters' center points (red) and goal (green)

4.2. Velocity Controller Testing

A separate system was developed for the purpose of testing the velocity controller system. It includes an individual ROS node named “test_velocityControl_node” which was created in order to call each of the services offered by the velocity_control_node and to request the motion planner to move the robot to specified places in the gazebo simulation environment.

The velocity_control_node connects with the simulated Kobuki robot via kobuki_node wrapper which is available in ROS as a package. Figure 4.6 shows the structure of the test_velocityControl_node,

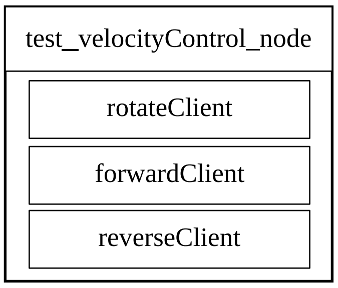


Figure 4.6 Structure of test_velocityControl_node

Figure 4.7 shows the combined systems of velocity_control_node and the test_velocityControl_node which was used to debug the velocity_control_node.

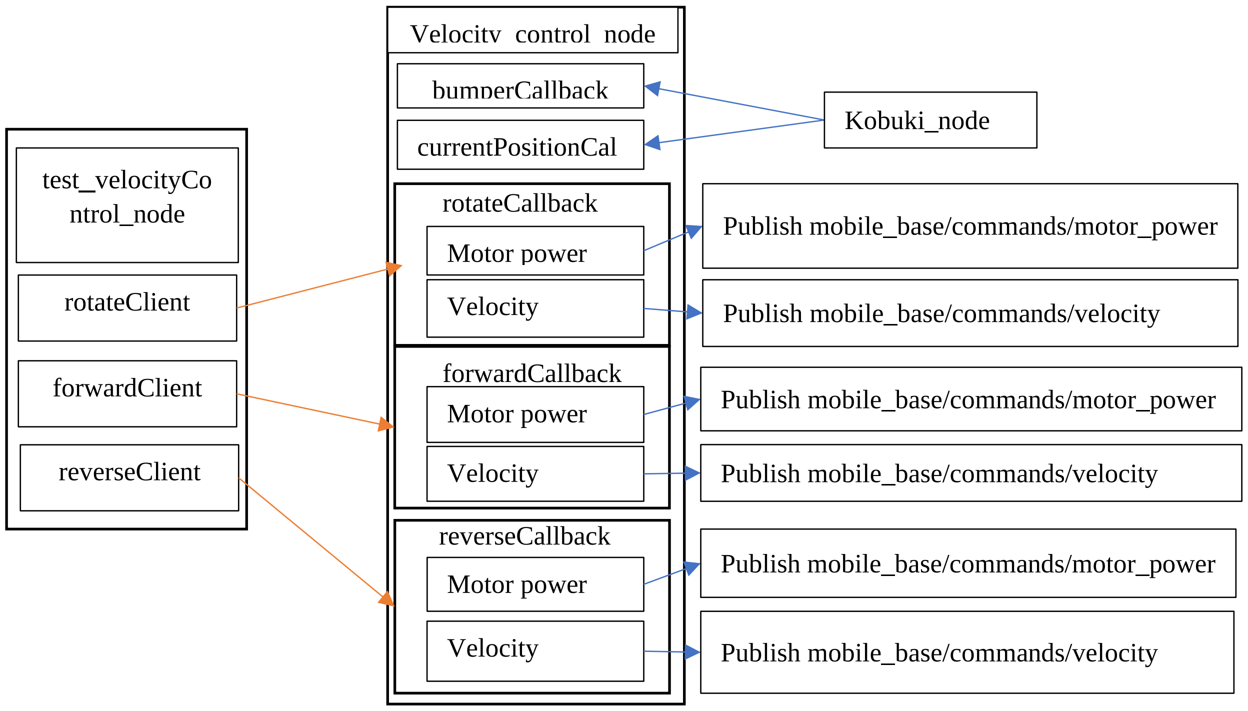

Velocity_control_node
bumperCallback
currentPositionCal
Kobuki_node
test_velocityControl_node
rotateClient
forwardClient
reverseClient
rotateCallback
Motor power
Velocity
forwardCallback
Motor power
Velocity
reverseCallback
Motor power
Velocity
Publish mobile_base/commands/motor_power
Publish mobile_base/commands/velocity
Publish mobile_base/commands/motor_power
Publish mobile_base/commands/velocity
Publish mobile_base/commands/motor_power
Publish mobile_base/commands/velocity

Figure 4.7 Combined test_velocityControl_node and velocity_control_node

4.3. Ground Evaluator Testing

Since the ROS node was not implemented, the depth image from the camera could not be captured. The only testing that happened was focused on testing the ability to detect water using the color cue and texture cue. Figure 4.8 and figure 4.9 show the results of color and texture analysis. Blue color indicates where the mask values become positive (water detected).

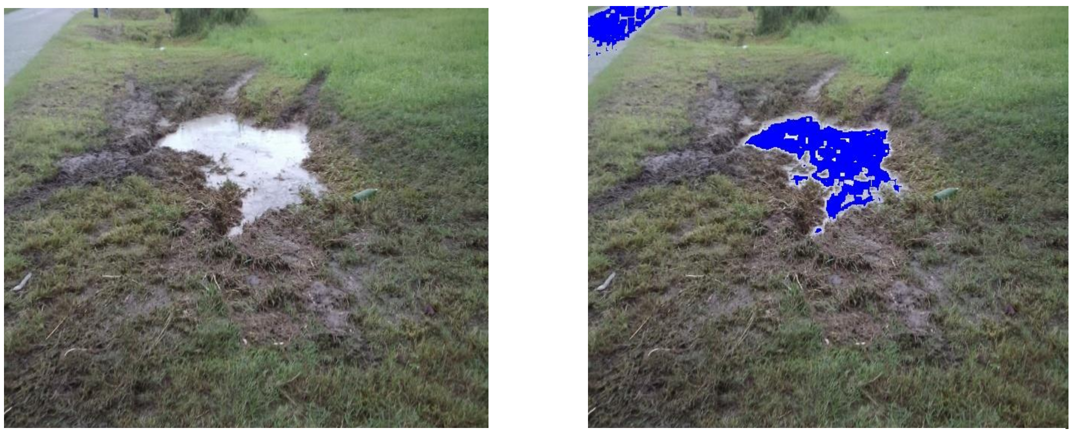


Figure 4.9 Before(left) and after(right) analyzing for water (Image 1)

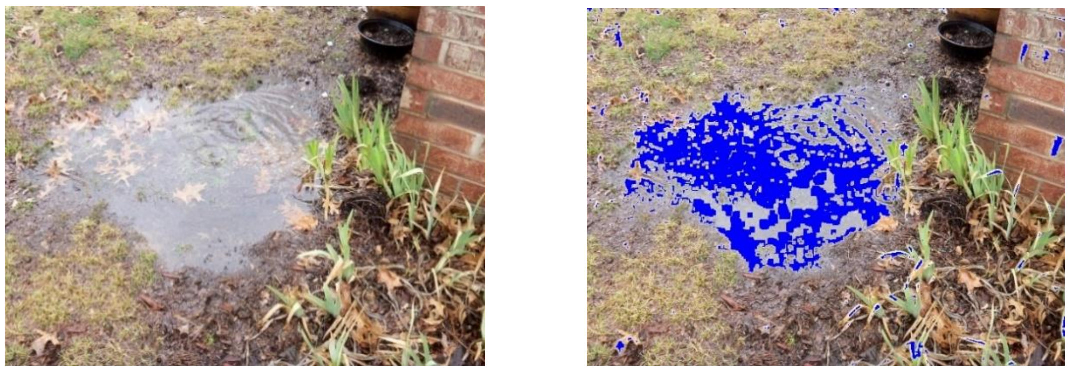


Figure 4.8 Before(left) and after(right) analyzing for water (Image 2)

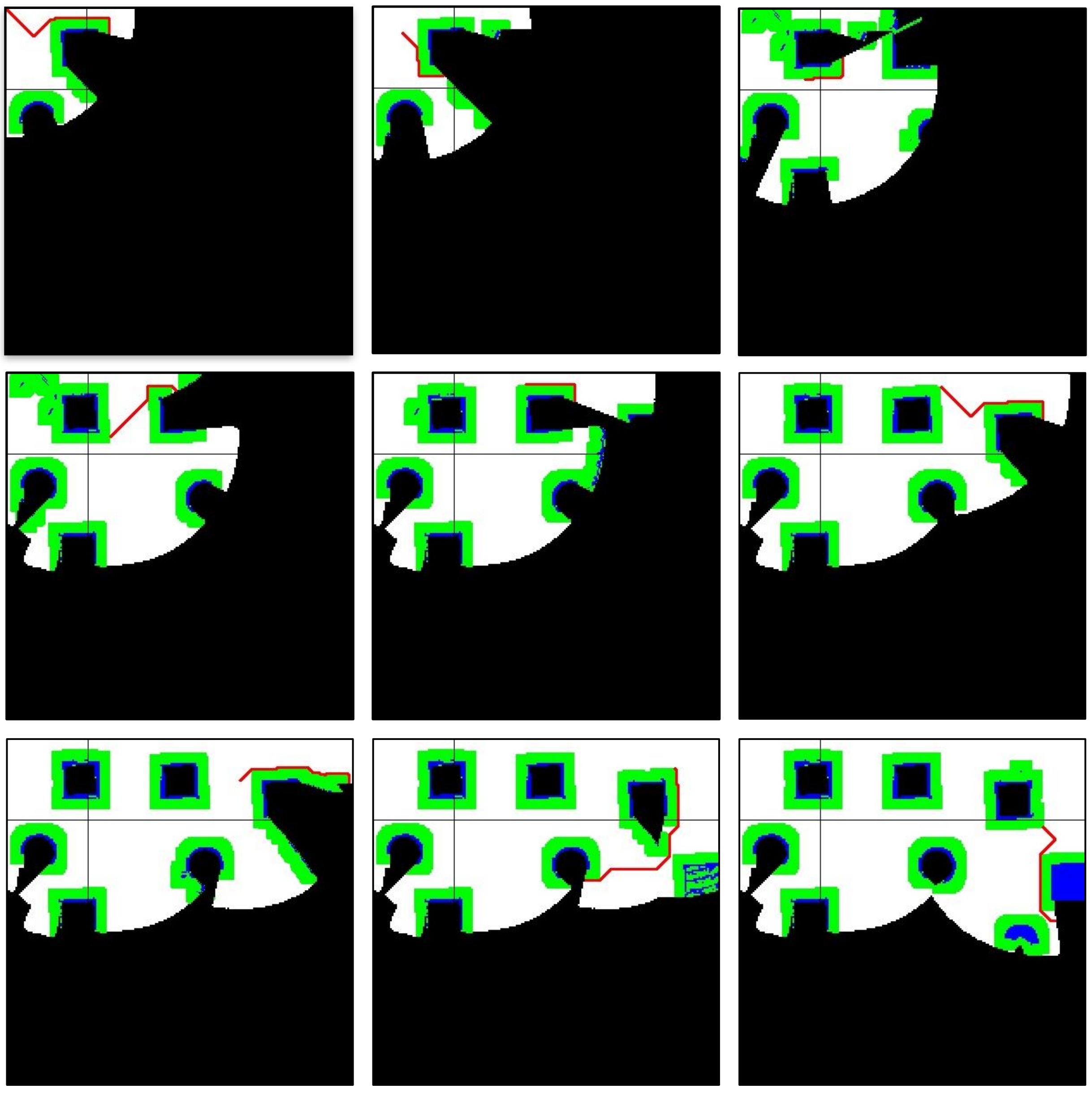
4.4. Path Planner Testing

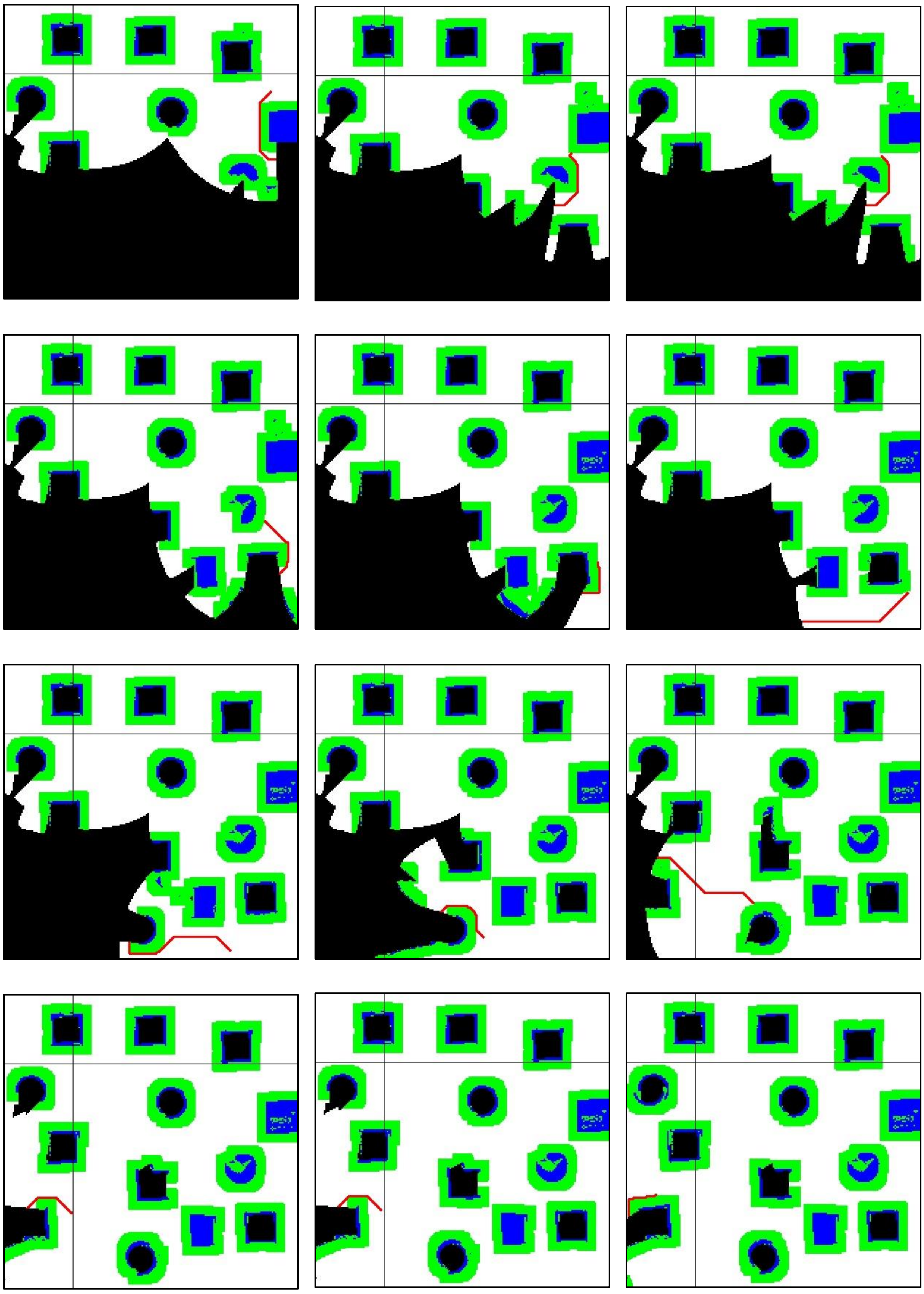
A separate ROS node named test_system_node was created as the client for the service "explore" advertised by the path_planner_node. In order to test the Path Planner subsystem, Velocity Controller and Goal Identifier subsystems had to be connected to it because the Path Planner is dependent on them. Due to this reason this subsystem test is the same as the whole system test. The details and results of the test are the same as in the next section.

4.5. Whole system Testing

The final simulation was done on the ROS gazebo with a 10m * 10m simulation environment. The robot model used was a Kobuki Turtlebot and the Microsoft Kinect was mounted on it as the RGBD camera.

The motion planner consisted of the Goal Identifier, Path Planner, Velocity Controller subsystems. The Ground evaluator subsystem was left out of the motion planner due to incompletion and less compatibility. Figure 4.10 contains images of the Grid map and the path calculation related to several stages of the navigation and mapping process that happened during the simulation.





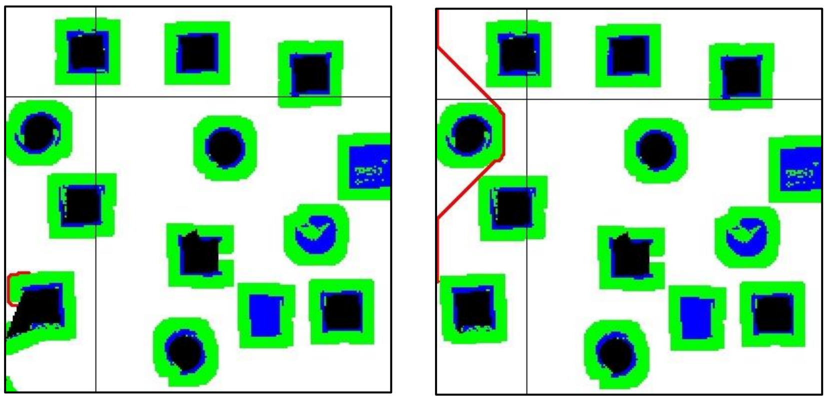


Figure 4.10 Grid map with obstacles (blue), padding (green) and path (red)

Figure 4.11 shows images of several instances of the navigation and mapping that happened inside the ROS gazebo simulation environment. Figure 4.5 and 4.12 shows the Gazebo simulation environment.

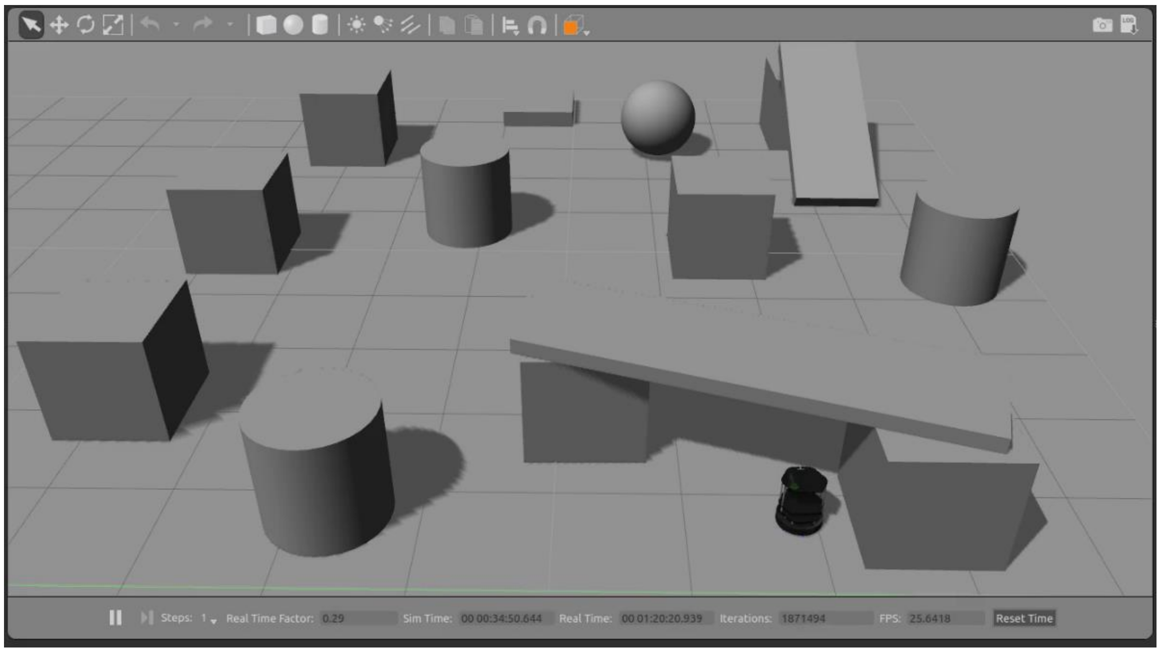


Figure 4.11 Gazebo Testing Environment

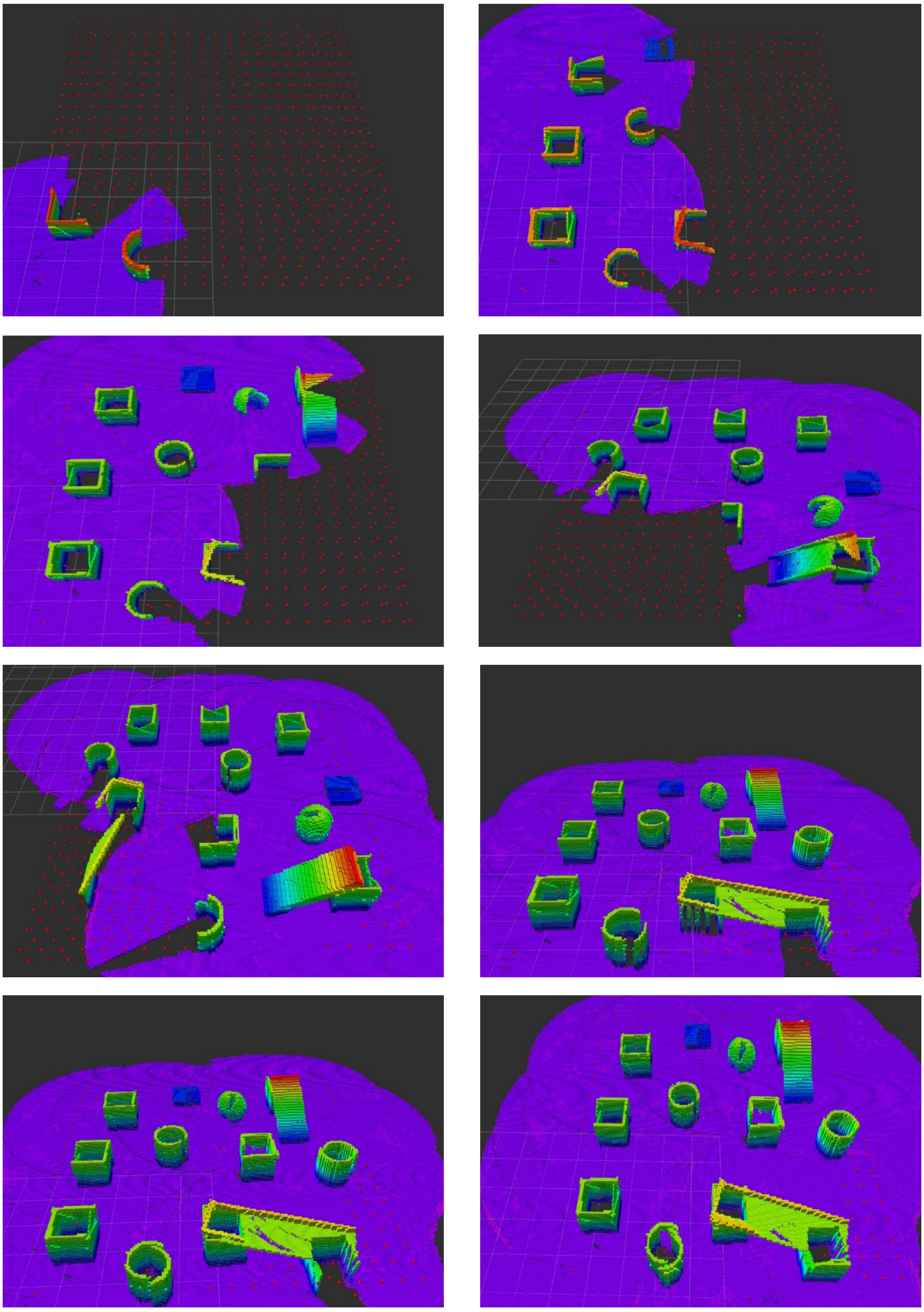


Figure 4.12 Several Stages of Mapping and navigation process

4.6. Real-World Testing

The motion planner system was directly implemented on the kobuki robot since it had the same control interface as the Kobuki robot model available in ROS Gazebo. The Freenecet package had to be used to connect with the Microsoft Kinect which was mounted on the kobuki robot. The control parameters of the real robot had to be reduced vastly from the values which were used on the simulation robot. These parameters included velocity, angular velocity, and scan radius. Higher velocity and angular velocity values used in the simulation robot tended to make the real robot unstable and caused odometry errors. The distance the real Kinect could see was also less than the distance the simulated Kinect could see. So, the scan radius also had to be reduced.

The accumulation of odometry issues was faster and more prominent in the Real-world robot compared to the simulated robot which caused the real robot to fail after some time. Figure 4.13 shows the first testing environment, figure 4.14 shows several stages of the map building and figure 4.15 shows several stages of path planning until the robot failed.

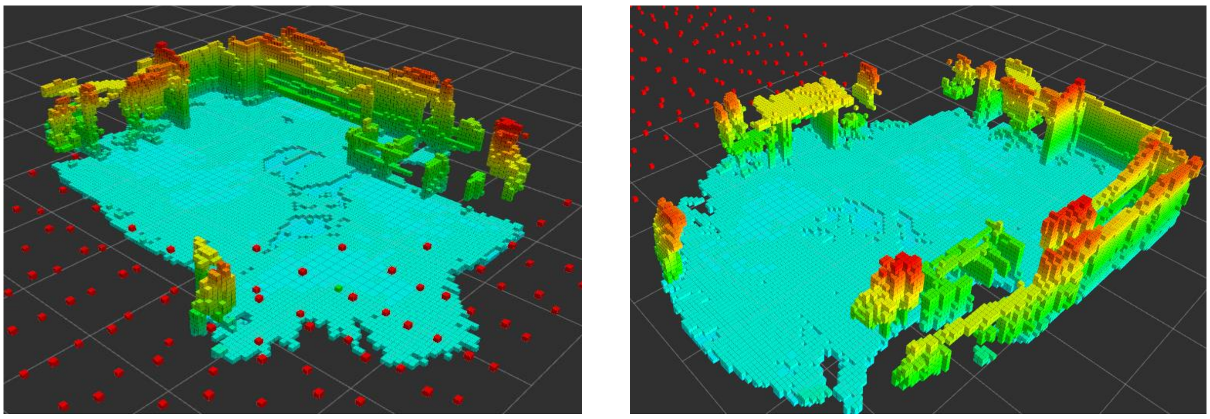


Figure 4.13 Stages of first environment mapping

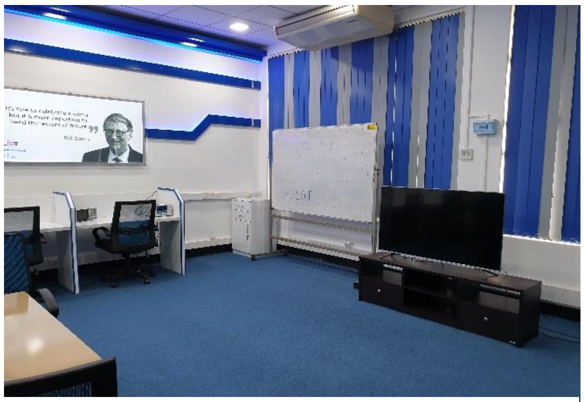


Figure 4.14 First Testing environment

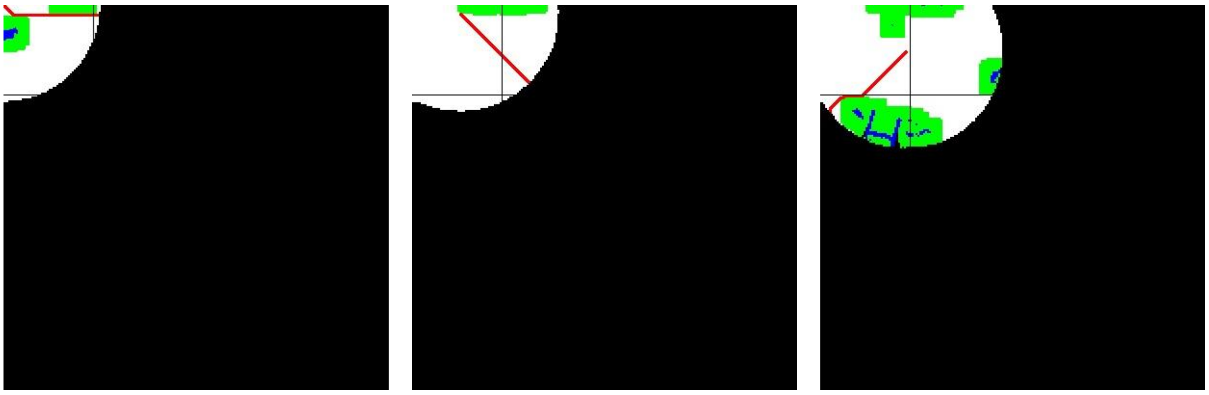


Figure 4.15 Navigation grid of the real environment testing

The robot performed much better in the second testing environment than the first due to the lack of a carpet in the second environment. It was able to map a large area and took a long time to fail than in the first testing environment. Figure 4.16 shows the second testing environment, figure 4.17 shows several stages of map building and figure 4.18 shows several stages of path planning until the robot failed.

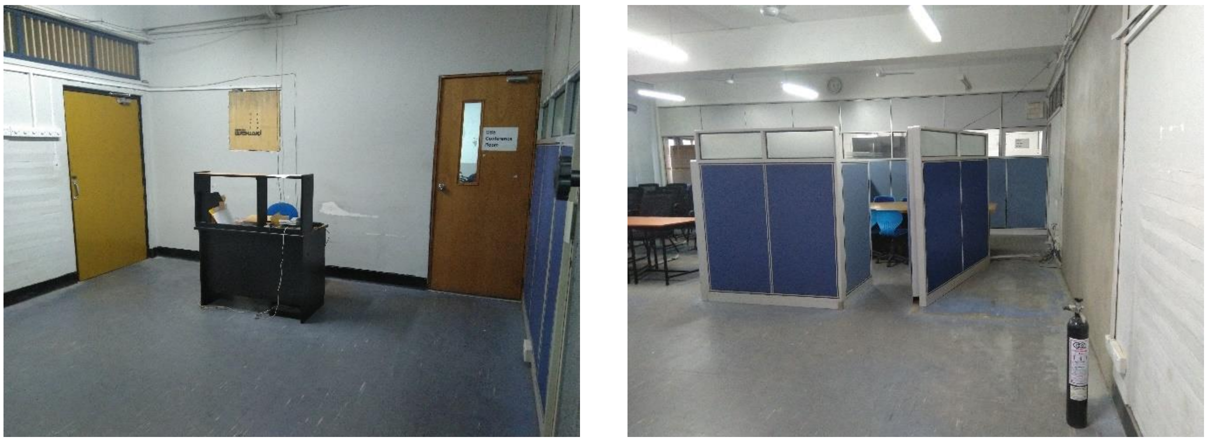


Figure 4.16 Second Testing Environment

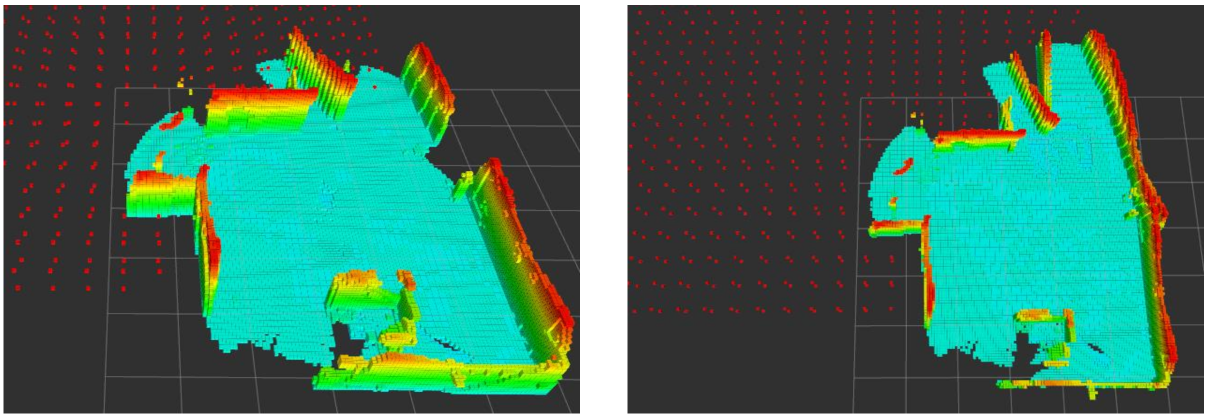


Figure 4.17 Stages of second environment mapping

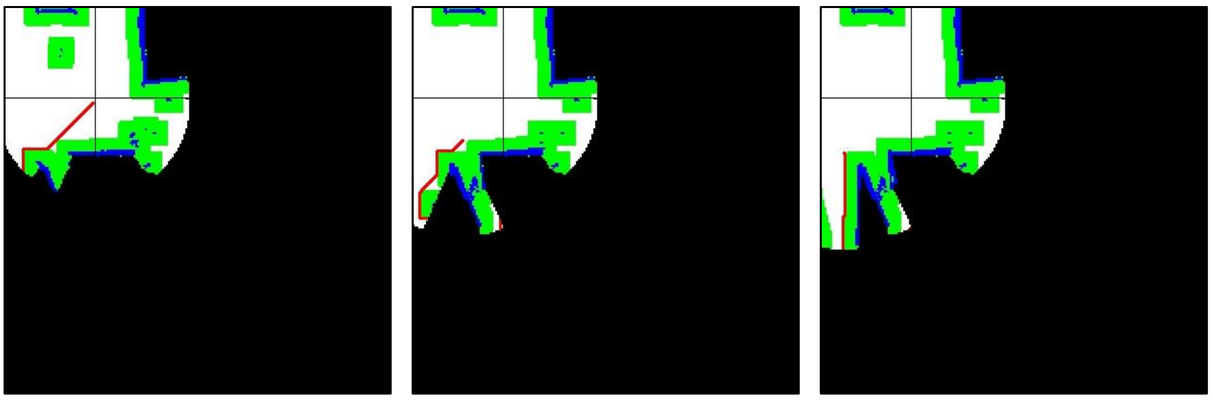


Figure 4.18 Navigation grid of the second environment

The solution to the odometry issues would be to use a localization method to correct the pose data of the robot but since a pre-built map of the environment is not available, the solution would be to develop a camera-based visual localization system.

5. Conclusion

The Goal Identifier subsystem was able to detect the environment and analyze it successfully through the Octomap framework. Velocity Controller subsystem also managed to move the robot successfully, but the speed was low due to the grid navigation approach. A better method needs to be designed, that is capable of filtering only the main points of a path, for the purpose of path simplification.

Goal Identifier, Path Planner and Velocity Controller subsystems combination successfully managed to navigate the simulated environment. But during real-world testing, it failed to complete the navigation due to the accumulation of odometry errors. Lack of localization was recognized as the main reason for the accumulation of errors. Developing or using a map independent localization system such as a visual localization can be proposed as a feasible solution.

The Ground evaluator subsystem which was left out due to lack of time and testing also needs to be incorporated into the motion planner. Capturing RGBD images along with the odometry could be a solution for the issue of mapping mud puddle areas onto the 2D grid which was used for path planning.

The main focus of this project, the development of a motion planner that uses point clouds to interpret the environment was successfully completed with good results. The opportunities it presented for future development work would make it much more efficient and usable in many other environments.

References

[1] C. Eldershaw and M. Yim, “Motion planning of legged vehicles in an unstructured environment,” in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 4, pp. 3383–3389.

[2] A. Elfes, “Using Occupancy Grids for Mobile Robot Perception and Navigation,” *Computer (Long. Beach. Calif).*, vol. 22, no. 6, pp. 46–57, 1989.

[3] F. Bourgault, A. A. Makarenko, S. B. Williams, B. Grocholsky, and H. F. Durrant-Whyte, “Information based adaptive robotic exploration,” in *IEEE/RSJ International Conference on Intelligent Robots and System*, 2002, vol. 1, pp. 540–545.

[4] R. Biswas, B. Limketkai, S. Sanner, and S. Thrun, “Towards object mapping in non-stationary environments with mobile robots,” in *IEEE/RSJ International Conference on Intelligent Robots and System*, vol. 1, pp. 1014–1019.

[5] X. Han, Y. Leng, H. Luo, and W. Zhou, “A novel navigation scheme in dynamic environment using layered costmap,” in *2017 29th Chinese Control And Decision Conference (CCDC)*, 2017, pp. 7123–7128.

[6] O. Khatib, “Real-Time Obstacle Avoidance for Manipulators and Mobile Robots,” *Int. J. Rob. Res.*, vol. 5, no. 1, pp. 90–98, Mar. 1986.

[7] K. Sabe, M. Fukuchi, J.-S. Gutmann, T. Ohashi, K. Kawamoto, and T. Yoshigahara, “Obstacle avoidance and path planning for humanoid robots using stereo vision,” in *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA ’04. 2004*, 2004, pp. 592-597 Vol.1.

[8] K. Konolige, M. Agrawal, R. C. Bolles, C. Cowan, M. Fischler, and B. Gerkey, “Outdoor Mapping and Navigation Using Stereo Vision,” in *Experimental Robotics*, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 179–190.

[9] A. L. Rankin, L. H. Matthies, and A. Huertas, “DAYTIME WATER DETECTION BY FUSING MULTIPLE CUES FOR AUTONOMOUS OFF-ROAD NAVIGATION,” in *Transformational Science and Technology for the Current and Future Force*, 2006, pp. 177–184.

[10] S. Siva, M. Wigness, J. Rogers, and H. Zhang, “Robot Adaptation to Unstructured Terrains by Joint Representation and Apprenticeship Learning,” *Robot. Sci. Syst.*, 2019.

[11] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard, “OctoMap: an efficient probabilistic 3D mapping framework based on octrees,” *Auton. Robots*, vol. 34, no. 3, pp. 189–206, Apr. 2013.

[12] D. Maier, A. Hornung, and M. Bennewitz, “Real-time navigation in 3D environments based on depth camera data,” in *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*, 2012, pp. 692–697.

[13] A. Hornung, M. Phillips, E. Gil Jones, M. Bennewitz, M. Likhachev, and S. Chitta, “Navigation in three-dimensional cluttered environments for mobile manipulation,” in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 423–429.

[14] M. Srinivasan Ramanagopal, A. P.-V. Nguyen, and J. Le Ny, “A Motion Planning Strategy for the Active Vision-Based Mapping of Ground-Level Structures,” *IEEE Trans. Autom. Sci. Eng.*, vol. 15, no. 1, pp. 356–368, Jan. 2018.

[15] K. Schmid, T. Tomic, F. Ruess, H. Hirschmuller, and M. Suppa, “Stereo vision based indoor/outdoor navigation for flying robots,” in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013, pp. 3955–3962.

[16] R. Dube, A. Gawel, C. Cadena, R. Siegwart, L. Freda, and M. Gianni, “3D localization, mapping and path planning for search and rescue operations,” in *2016 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, 2016, no. 1, pp. 272–273.

[17] A. Chilian and H. Hirschmuller, “Stereo camera based navigation of mobile robots on rough terrain,” in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2009, pp. 4571–4576.

[18] R. L. Klaser, F. S. Osorio, and D. Wolf, “Vision-Based Autonomous Navigation with a Probabilistic Occupancy Map on Unstructured Scenarios,” in *2014 Joint Conference on Robotics: SBR-LARS Robotics Symposium and Robocontrol*, 2014, pp. 146–150.

[19] C. Wang *et al.*, “Autonomous mobile robot navigation in uneven and unstructured

indoor environments,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 109–116.

[20] J. Guivant, E. Nebot, J. Nieto, and F. Masson, “Navigation and Mapping in Large Unstructured Environments,” *Int. J. Rob. Res.*, vol. 23, no. 4–5, pp. 449–472, Apr. 2004.

[21] M. Quigley *et al.*, “ROS: an open-source Robot Operating System,” in *ICRA workshop on open source software*, 2009, vol. 3, no. 3.2, p. 5.

[22] N. Koenig and A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, 2004, vol. 3, pp. 2149–2154.