JOMO KENYATTA UNIVERSITY OF AGRICULTURE

AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

BSc. Electronic and Computer Engineering

FINAL YEAR PROJECT REPORT

PROJECT TITLE:

SMART SOLAR GRASS CUTTER

Submitted by:

NAME	REGISTRATION NUMBER
CHELSEA KINYA GITONGA	EN 272-3689/2015
ISSAC ROBIN MACHUKI	EN 272-0535/2015

PROJECT SUPERVISOR:

MR. ASAPH MUHIA

A Final Year Project Proposal submitted to the Department of Electrical and Electronic Engineering in partial fulfillment of the requirements for the award of a Bachelor of Science Degree in Electronic and Computer Engineering.

NOVEMBER 2020

DECLARATION:

This project proposal is our original work, except where due acknowledgement is made in the text, and to the best of my knowledge has not been previously submitted to Jomo Kenyatta University of Agriculture and Technology or any other institution for the award of a degree or diploma.

SIGNATURE: .....

DATE: .....

NAME: CHELSEA GITONGA KINYA

REGISTRATION NUMBER: EN 272-3689/2015

SIGNATURE: .....

DATE: .....

NAME: ISSAC ROBIN MACHUKI

REGISTRATION NUMBER: EN 272-0535/2015

TITLE OF PROJECT: SMART SOLAR GRASS CUTTER

SUPERVISOR CONFIRMATION:

This project proposal has been submitted to the Department of Electrical and Electronic Engineering, Jomo Kenyatta University of Agriculture and Technology, with my approval as the University supervisor:

MR. ASAPH MUHIA

SIGNATURE: .....

DATE: .....

Table of Contents

LIST OF FIGURES

LIST OF TABLES:

ABBREVIATIONS:

AMCL:……Adaptive Monte Carlo Localization

BFS:……Breadth First Search

BLDC:……Brushless Direct Current

DFS:……Depth first Search

EKF:……Extended Kalman Filter

GPS:……Global Positioning System

IMU:……Inertial Measurement Unit

IPS:……Indoor Positioning System

KF:……..Kalman Filter

LIDAR:……...Light Detection and Ranging

MCL:……..Monte Carlo Localization

SLAM:……...Simultaneous Localization and Mapping

UCS:……..Uniform Cost Search

ROS……Robot Operating System

PROJECT ABSTRACT:

Grass cutting technologies in frequent use today are slashers, push lawn mowers, gasoline lawn mowers, cordless electric lawn mowers and robotic lawn mowers.

Cutting grass using a slasher is back-breaking work and the resultant cut is uneven. Accidents (e.g., breaking of window-panes) and injuries (e.g., sprained wrists and/or strained back muscles) are not uncommon while using slashers. Push lawn mowers are easier to use in comparison to slashers. However, they require hard pushing while cutting long and rough grass and while navigating corners and obstacles.

Gasoline lawn mowers release evaporative emissions such as hydrocarbons, oxides of nitrogen and carbon monoxide that contribute towards air pollution. On the other hand, cordless electric lawn mowers have limited run times of an hour or less with their re-charging time being longer than their run-time.

The grass cutting technologies mentioned above require human operators. A human operator is needed for the purposes of navigation and obstacle avoidance. The human operator has to spare time to perform menial tasks wasting both his/her time and energy. The work output as a result of human input is also subject to imperfection due to human error.

Robotic lawn mowers eliminate direct human input. The robotic lawn mowers that are currently available are contained by a border wire around the lawn that defines the areas to be mowed. Installation of the border wire is time consuming and requires several installation iterations to get it right. These mowbots also employ a heuristic path planning approach to mow the lawn. This approach is not optimal wasting finite resources such as time and battery energy. The resultant cut also does not have stripes.

A solar-powered autonomous grass cutter that performs straight line navigation will address the problems highlighted above. Solar energy is a green alternative eliminating evaporative emissions from gasoline lawn mowers. An on-board solar panel will also address the recharging problem present within the cordless electric lawn mowers. Finally, the problems associated with

robotic lawn mowers will be eliminated by developing and implementing mapping, localization, computational motion planning and obstacle avoidance algorithms.

CHAPTER ONE:

1. INTRODUCTION:

1.1. Background:

Advancements in lawn mowing technology have targeted three main areas: the cutting technique, the source of energy and control.

Two types of lawn mowers have resulted from the cutting technique: cylinder/reel lawn mowers and rotary lawn mowers.

The reel lawn mower was patented January 28, 1868 [1]. It was invented by Amariah Hills. A cylinder mower carries a fixed, horizontal cutting blade at the desired height of cut. Over this is a fast-spinning reel of blades which force the grass past the cutting bar. Of all the mowers, a properly adjusted cylinder mower makes the cleanest cut of the grass, and this allows the grass to heal more quickly. The machine also has a rear roller after the cutting cylinder which smooths the freshly cut lawn and minimizes wheel marks giving the best "stripped effect" across the mown lawn. Reel lawn mowers are best for flat lawns that you want to keep short and well-manicured. However, cylinder lawn mowers struggle to cut long grass and do not perform well when the lawn has steep banks.

Rotary mowers were not developed until engines were small enough and powerful enough to run the blades at sufficient speed. A rotary mower rotates about a vertical axis with the blade spinning at high speed, relying on impact to cut the grass. This tends to result in a rougher cut resulting in bruising and shredding of the grass leaf leading to discoloration of the leaf ends as the shredded portion dies. This is particularly prevalent if the blades become clogged or blunt. Frequent unclogging and sharpening of the blades almost entirely eliminates this problem. A rear roller can be separately attached to the rotary lawn mower achieving the same "stripped effect" from the reel lawn mower. Rotary lawn mowers are the most versatile type, coping very well with most types of grass and are therefore better than cylinder mowers at cutting longer and rougher grasses. Rotary lawn mowers can also be used on banks to a certain extent, provided that they are not too steep.

Four types of lawn mowers resulted from the type of energy source: by hand lawn mowers, gasoline/petrol lawn mowers, propane lawn mowers and electric lawn mowers.

In hand-powered lawn mowers, the reel is attached to the mower's wheels by gears, so that when the mower is pushed forward, the reel spins several times faster than the plastic or rubber-tired wheels turn. Mowing with hand-powered lawn mowers requires hard pushing while cutting long and rough grass and while navigating corners and obstacles.

The first internal combustion-engine lawn mower prototype was by WJ Stephenson Peach in 1869 [1]. The invention of motorized engines eliminated hard pushing since the lawn mowers could propel themselves forward. Gasoline lawn mowers have the advantages over electric mowers of greater power and distance range. However, they do create a significant amount of pollution due to the combustion in the engine, and their engines require periodic maintenance such as cleaning or replacement of spark plugs and air filters, and changing the engine oil. Gasoline lawn mowers are also quite noisy.

Propane lawn mowers emit 50 to 60% less CO_2 in comparison to gasoline lawn mowers.

Electric lawn mowers are further subdivided into corded and cordless electric models. Both are relatively quiet, typically producing less than 75 decibels, while a gasoline lawn mower can be 95 decibels or more.

Corded electric mowers are limited in range by their trailing power cord, which may limit their use with lawns extending outward more than 30-45 m from the nearest available power outlet. There is the additional hazard with these machines of accidentally mowing over the power cable, which stops the mower and may put users at risk of electric shock.

Cordless electric mowers are powered by a variable number (typically 1–4) of 12-to-80-volt rechargeable batteries. More batteries mean more run time and/or power. Cordless mowers have the maneuverability of a gasoline-powered mower and the environmental friendliness of a corded electric mower. Maintenance of corded electric mowers is limited to clearing of the deck and sharpening of the blades. However, they have limited run times of an hour or less and charging the batteries takes longer than how long you can run them for. The eventual disposal of worn-out batteries is also problematic (although some manufacturers offer to recycle them).

Installation of an on-board powering system (in the form of a solar panel) would greatly eliminate the limited run-time problem giving the cordless electric mowers great distance range.

Based on control, two types of lawn mowers exist: human controlled lawn mowers and robotic lawn mowers.

Push lawn mowers, self-propelled lawn mowers, ride on lawn mowers and remote-controlled lawn mowers fall under the human controlled category. A human operator is needed for the purposes of navigation and obstacle avoidance. The human operator has to spare time to perform menial tasks wasting both his/her time and energy. The work output as a result of human input is also subject to imperfection due to human error.

Robotic lawn mowers/ lawn-mowing bots/ mowbots eliminate direct human input. The lawn-mowing bots that are currently available are contained by a border wire around the lawn that defines the areas to be mowed. Installation of the border wire is time consuming and requires several installation iterations to get it right. These mowbots also employ a heuristic path planning approach to mow the lawn. When the mowbot approaches a random obstacle like a tree, it rotates to a random angle and just keeps going. As time increases the chance that the entire lawn is covered approaches 100%. The robot does not have a map of the lawn and is not aware of its location within the lawn. This approach is not optimal wasting finite resources such as time and battery energy. The resultant cut also does not have stripes.

1.2. Problem Statement:

Gasoline lawn mowers release evaporative emissions such as hydrocarbons, oxides of nitrogen and carbon monoxide that contribute towards air pollution. Cordless electric lawn mowers require frequent recharging which is a time-consuming process. In addition, the robotic lawn mowers in existence today employ a heuristic path planning approach that is not optimal, wasting finite resources such as time and battery energy. Therefore, there is a need to develop an autonomous grass cutter that performs straight-line navigation with an on-board solar panel that powers it during its operation.

1.3. Problem Justification:

This project seeks to come up with a solar-powered autonomous grass cutter that performs straight line navigation. Solar energy is clean energy and its use in this project will eliminate the

problem of pollution caused by gasoline lawn mowers. An on-board solar panel will also address the recharging problem present within cordless electric lawn mowers. Finally, this robotic grass cutter will achieve straight line navigation. This type of navigation is more time and energy efficient in comparison to the heuristic approach.

1.4. Objectives:

1.4.1. Main Objective

To design, fabricate and deploy a solar-powered autonomous grass cutter that performs straight-line navigation.

1.4.2. Specific Objectives

1. To design and fabricate a wheeled robotic grass cutter.
2. To develop and implement mapping algorithms for the grass cutter.
3. To develop and implement localization algorithms for the grass cutter.
4. To develop and implement path planning algorithms to implement straight line navigation for the grass cutter.
5. To develop and implement algorithms for dynamic obstacle avoidance.

1.5. Scope:

The mapped environment will be a 5m by 5m room consisting of the following obstacles: a table, a chair and a bed. The robot will have a 20cm by 20cm chassis. The solar panel and the kidnapped robot problem will not be considered.

1.6. Distribution of roles:

Student Name:	Specific objectives carried out:
Chelsea Kinya Gitonga	1,3,4
Issac Robin Machuki	1,2,5

Table 1.1: Distribution of roles.

CHAPTER TWO:

2. LITERATURE REVIEW

2.1. SUMMARY

This chapter provides a summary of commonly used algorithms in mapping, localization, path planning and obstacle avoidance.

2.2. MAPPING

Autonomous mobile robots need to achieve the following fundamental tasks: *localization, mapping and navigation* [2]. *Localization* is the ability of a robot to identify its absolute pose within an environment. *Mapping* is the process that involves a robot exploring its environment to identify, represent and store the features in it for future reference during navigation. *Navigation* is the process of moving from one point in an environment to the desired location. Navigation entails *path planning* and *obstacle avoidance* [3].

Standalone localization techniques used by mobile robots, such as Monte Carlo Localization are map-based. This means that a robot is fed with a map of its environment, for example a floor plan and it localizes itself within that map. Similarly, standalone mapping techniques used by mobile robots such as occupancy grids assume the robots pose is fully known before proceeding to generate a map. Systems such as GPS and IPS can be utilized to establish this absolute pose in outdoor and indoor environments respectively [4].

However, the first two tasks still form a chicken and egg problem in completely unknown environments [2]; the pose and map are unknown. This is because a robot needs to know its pose within an environment in order to build effective maps. At the same time a robot needs to have a map of its environment in order to identify its pose. This problem is often called the Simultaneous Localization and Mapping problem. There are several techniques and algorithms that have been developed over the years to handle the SLAM problem. In fact, most mobile robots in factories, offices, homes, for rescues missions, underwater and space exploration are implementing a form of SLAM algorithm and afterwards perform a mapping algorithm [5].

2.2.1. MAPPING WITH KNOWN POSES.

Historically, work in mapping first concentrated on robots operating in indoor environments as the models take advantage of the fact that the world can be represented as vertical structures on reference ground planes. These techniques have later been extended to outdoor environments [5]:

1. Occupancy grids.
2. Feature-based maps.
3. Topological maps.
4. Elevation maps,
5. 3-D grids.
6. Meshes

2.2.1.1. OCCUPANCY GRIDS

Occupancy grid maps were first introduced in the 1980s by Mavorec and Elfes [2] [4] [5] [6] [7] [8]. It is a probabilistic method of representing the environment and is popular to date. This representation involves dividing the area of vision of a sensor, for example a LIDAR sensor, into grids of fixed sizes. The size of a grid cell is also called its resolution. For each cell, given the sensory input obtained by a robot at the corresponding position, the occupancy grid map approach calculates a posterior probability, which is the probability of occupancy of that cell.

$P(m \mid x_{1:t}, z_{1:t})$

The above equation states, "What's the probability of occupancy of a grid cell given a single observation z (t) and the corresponding pose x (t) of the robot." Occupancy grid maps have been successfully applied in numerous installations of mobile robots and have been proven to be a powerful tool that supports various navigation tasks such as localization and path planning.

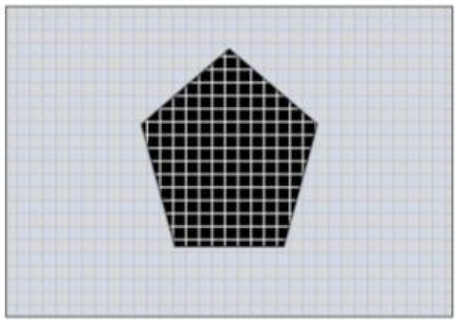


Figure 2.1 A grid representation

Advantages

1. It offers the most detailed map, and the resolution can be adjusted easily.
2. It doesn't require any predefined feature models.

Disadvantages

1. It is a discrete map hence prone to discretization errors.
2. High resolution will lead to high computation complexity.

2.2.1.2. FEATURE-BASED MAPS

The representation of the environment by classified features is a popular alternative to the grid-based approximations described above. The features used could be key points, lines, edges, corners, planes, etc. [4]

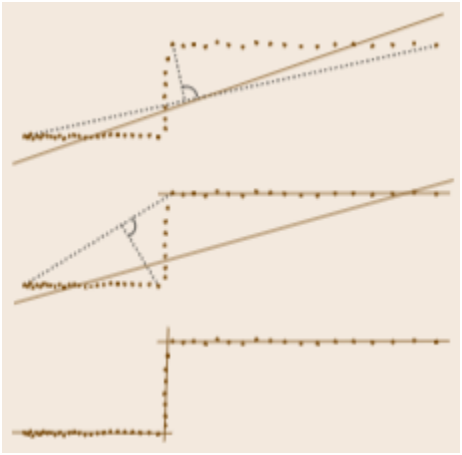


Figure 2.2 Feature map representation recognizing lines

Advantages

1. It is accurate and keeps the key features of the environments.
2. The map format is flexible; each feature could be handled independently and it is very easy to add or remove features from the map.
3. It is not limited to one type of sensor; it can be built by different sensors at the same time.

Disadvantages

1. The map only represents features; data not classified as features will be discarded.
2. A user has to build feature model before-hand.
3. It is hard to store and update map, when there is a lot of noise.

2.2.1.3. TOPOLOGICAL MAPS

In contrast to the above representations, which mainly focus on the geometric structure of the environment, the environment in this approach is represented by a graph-like structure, in which the nodes are locally distinguishable places and the edges are the paths between the places. Here, distinctive places are identified such as doors, stairways and corridors. [4] [5] [6].

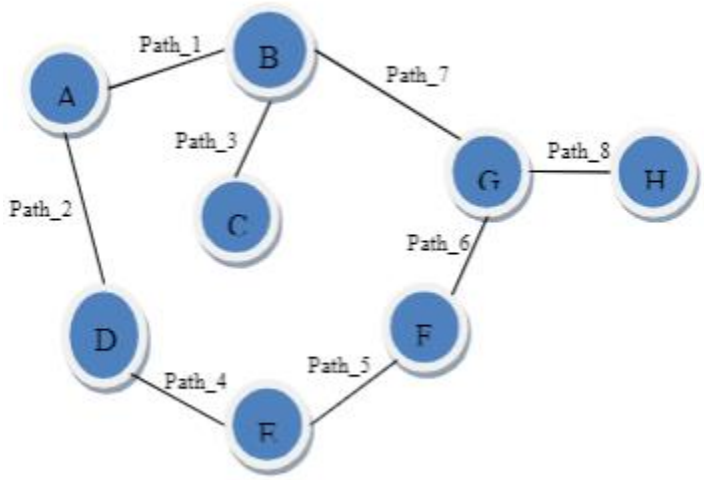


Figure 2.3 A topological map

A topological map is a simplified map in that it only keeps the key information and removes all the other information. It gives the relations of all the nodes but not the detailed information like scales, distances and directions. In order to use this map, a robot has to be able to detect and recognize all the nodes.

Advantages

1. It can be scaled easily with the size of environment.
2. Easy to build, since the map is formed by node.
3. Easy to use, especially for path planning.

Disadvantages

1. The map loses detailed position information of the obstacle. It can't be used directly in the applications which require exact positions of the environment.
2. It is hard to recognize the node, which need to be described.

2.2.1.4. ELEVATION GRIDS

This is a 3-D extension of Occupancy Grids. It assumes that the terrain can be represented as a function h=f (x, y), where x and y are the coordinates on a reference plane and h is the corresponding elevation. This digital elevation map stores the value of h, at discrete locations (x_i, y_i). These maps have been used extensively for mobile robots operating in natural environments with no vertical surfaces or overhangs [5].

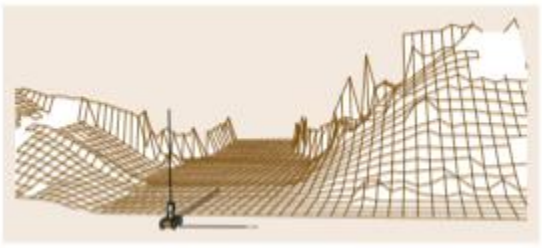


Figure 2.4 Example elevation map built by accumulating 3-D data from a range sensor

Advantages

1. Allows for robust autonomous navigation for mobile robots, considering the elevation of obstacles and other structures inside the environment.
2. Height information can be used to solve data association problem (distinguishing between two similar looing features) in robotic mapping.

Disadvantages

1. Has excessively high computational demands for building and storing and hence is not commonly used for direct application of mobile robots.

2.2.1.5. 3-D GRIDS AND POINT SETS

Elevation maps as described above assume a reference direction. In many cases, this assumption is violated. An alternative is to represent the data directly in 3-D without projecting it onto a reference 2-D plane.

Advantages

1. There is no restriction on the geometry of the environment.
2. It enables the computation of cost, the difficulty of traversing through a map, evaluated from the true local 3-D distribution of the data. This is important, for example in environments with vegetation scatter, which cannot be modeled as a surface.

Disadvantages

1. It is difficult to efficiently manage and manipulate very large sets of 3-D points [5].

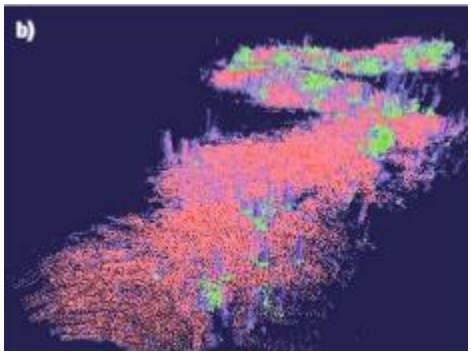


Figure 2.5 3-D Map accumulated over a large number of scans

2.2.1.6. MESHES

As described above, elevation maps are compact and easy to implement but they are restricted to a particular class of terrain. At the other extreme, using 3-D representations directly is more general but it is also more expensive computationally and it does not represent explicitly surface continuity. A compromise is to represent a map as a mesh [5].

Advantages

1. This approach is attractive because it can, in principle, represent any combination of surfaces.
2. It is also a compact representation, in that, even though the size of the mesh may be initially very large, efficient mesh simplification algorithms exist that can be used to reduce the map to a small number of vertices.

Disadvantages

1. The key issue with meshes is that extraction of the correct surfaces from raw data can be difficult in complex environments. In particular, the data is corrupted by sensor noise and by random clutter from other sources such as vegetation, which cannot be represented as a continuous surface.

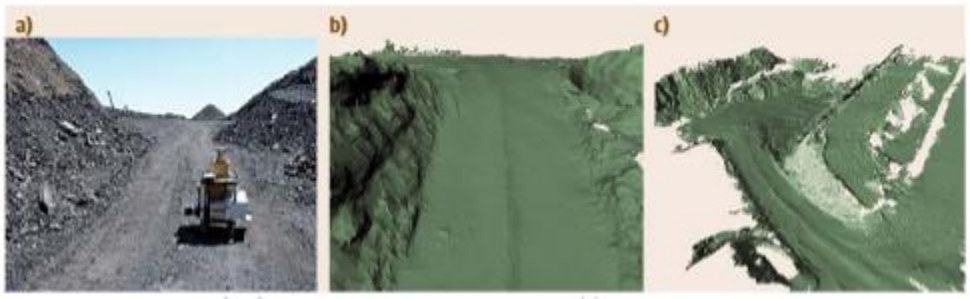


Figure 2.6 The mesh representation of a map

2.2.1.7. DYNAMIC ENVIRONMENTS

The majority of techniques for mapping have been developed for static environments. Certain approaches, like occupancy grids or elevation maps can in principle deal with dynamic environments in which the objects move. Their drawback lies in the fact that the time to unlearn that a cell is free or that the elevation has changed can require as many observations as the robot received with the same area being occupied or with different elevation. To resolve this problem, several alternatives have been developed in the past.

One popular technique is to track moving objects using feature-based tracking algorithms [5]. Such approaches are useful only when they type of dynamic object is known in advance. They have been successfully applied in the context of learning 3-D city maps from range data.

Alternative approaches to such tracking techniques include those that learn maps on different time scales. Other approaches explicitly learn different states of dynamic environment. Others only map static aspects.

Figure 2.7 Mapping in Dynamic Environments

2.2.2. SLAM- SIMULTANEOUS LOCALIZATION AND MAPPING

SLAM addresses the problem of a robot navigating in an unknown environment. While navigating the environment, the robot seeks to acquire a map thereof, and simultaneously localize itself using this map. SLAM can be seen as the main problem in truly autonomous mobile robots. Despite significant progress in this area, it still poses great challenges.

2.2.2.1. Taxonomy of the SLAM Problem

Literature distinguishes the SLAM algorithms in a number of ways [5].

Full/offline SLAM problem: estimating the posterior over the entire robot path together with the map from the available data.	Online SLAM problem: recovering the present robot location, instead of the entire path
Volumetric: the map is sampled at a resolution high enough to allow for photorealistic reconstruction of the environment.	Feature-based: extracts sparse features from the sensor stream. The map is then only comprised of features.
Topological: recover only a qualitative description of the environment, which characterizes the relation of basic locations.	Metric: provide metric information between the relations of such places.
Known correspondence: assuming a relation of identity of sensed things to other sensed things. (data association problem)	Unknown correspondence: do not have an assumption relating identity of sensed things to other sensed things.
Static: assume that the environment does not change over time.	Dynamic: allow for changes in the environment.
Small location uncertainty: allow only for small errors in the location estimate. (loop-closing problem)	Large location uncertainty: allow large errors in the location estimate.
Active: the robot actively explores its environment in pursuit of an accurate map.	Passive: some other entity is controlling the robot and the SLAM algorithm is purely observing.
Single robot.	Multi robot.

Table 2.1 Taxonomy of SLAM

Among the many SLAM algorithms, the three discussed below are the most basic and popular. They assume a static environment with a single robot.

1. Extended kalman filter SLAM
2. Graph-based optimization techniques
3. Particle methods

2.2.2.2. EXTENDED KALMAN FILTER

This formulation of SLAM is the earliest and still one of the most influential. EKF SLAM proposes the use of a single state vector t, to estimate the locations of the robot and a set of features in the environment, with an associated error covariance matrix representing the uncertainty in these estimates, including the correlations between the vehicle and the feature state estimates [5] [7] [8]. As the robot is moving through its environment taking measurements, the system state vector and covariance matrix are updated using the EKF. As new features are observed, new states are added to the system state vector and the size of the system covariance matrix grows quadratically. This approach assumes a feature- based environmental representation, in which the objects can be effectively represented as points in an appropriate parameter space. The position of the robot and the locations of the features form a network of uncertain spatial relationships. The development of appropriate representations is a critical issue in SLAM and is closely connected to how a robot perceives its environment (the sensors used).

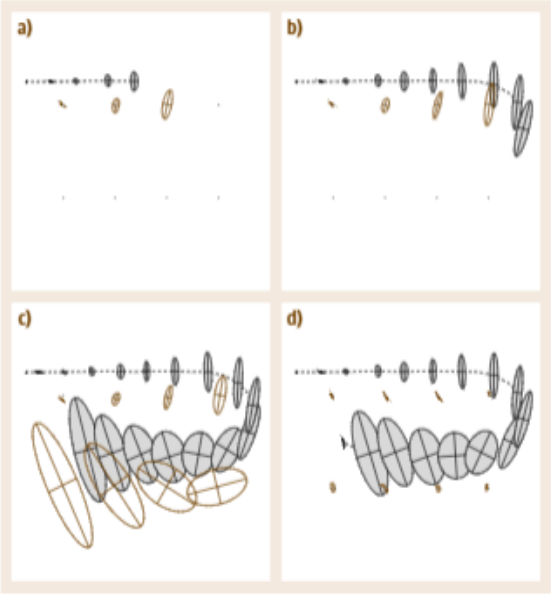


Figure 2.8 EKF SLAM

Advantages

1. Able to deal with non-linear models.
2. Can provide good loop closure references (easily identify previously observed features).

Disadvantages

1. It has high computational complexity and thus is difficult to meet large-scale maps.

EKF SLAM has been applied successfully to a large range of navigation problems involving airborne, underwater, indoor and various other vehicles.

2.2.2.3. GRAPH-BASED OPTIMIZATION TECHNIQUE

A second family of algorithms solves the SLAM problem through nonlinear sparse optimization [5] [7] [8]. They draw their intuition from a graphical representation of the SLAM problem. Landmarks and robot locations can be thought of as nodes in a graph. Every consecutive pair of location (coordinate) is tied together by an arc that represents the information conveyed by the odometry reading.

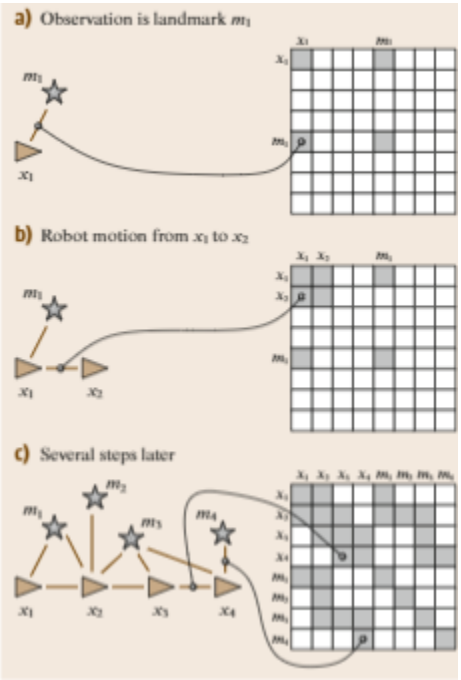


Figure 2.9 Graph based SLAM construction

Advantages

1. They scale to much higher-dimensional maps than EKF SLAM methods, since the constraint of the covariance matrix that scales quadratically does not exist here.
2. The update time of the graph is constant and the amount of memory required is linear, allowing for large scaling.
3. Provide better accuracy and consistency in its estimating technique (matrix).

Disadvantages

1. Most graphical SLAM methods are offline/full, meaning they optimize for the entire robot path. If the robot path is long, the optimization may become cumbersome.

2.2.2.4. PARTICLE METHODS

The third principal SLAM paradigm is based on particle filters. Particle filter represent a posterior through a set of particles [5] [7] [8]. Each particle is best thought of as a concrete guess as to what the true value of the state may be. By collecting such many guesses, particles, into a set, the particle filter captures a representative sample from the posterior distribution. The particle filter has been shown under mild conditions to approach the true posterior as the particle set size goes to infinity. In recent years, with the advent of extremely efficient microprocessors, particle filter has become a popular algorithm.

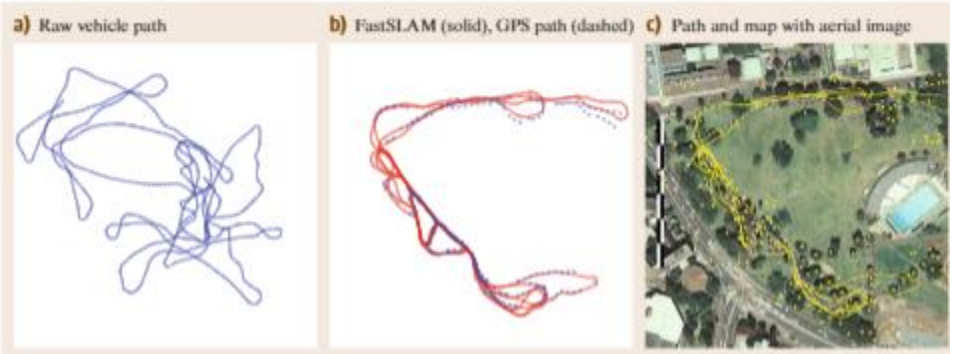


Figure 2.10 Particle mapping

Advantages

1. It solves both the full and online SLAM problems. This makes it a filter, similar to EKF.
2. It makes it easy to pursue multiple data association hypotheses. This means that the method samples the correct posterior even for SLAM problems with unknown data association- something neither of the first two algorithms could do.
3. It can be implemented very efficiently with advanced tree methods to represent the map estimates.

Disadvantages

1. Suffers from degeneracy due to the process when sampling the proposal distribution that requires the history of the particle.
2. Particle depletion during sampling process.

2.2.3. SUMMARY

Graph-Based SLAM algorithm will be used because it is the most accurate, it is more memory efficient in comparison to EKF and it does not suffer from degeneracy. Binary occupancy grid mapping algorithm will be used to generate the map because it provides the most detailed map, it does not rely on predefined features and is computationally inexpensive.

2.3. LOCALIZATION

Localization is the challenge of determining your robot's pose (position and orientation) in a mapped environment. We do this by implementing a probabilistic algorithm to filter noisy sensor measurements and track the robot's position and orientation. There are four very popular localization algorithms:

- Monte Carlo Localization Algorithm
- Extended Kalman Filter Localization Algorithm
- Grid Localization Algorithm
- Markov Localization Algorithm

The list above takes popularity and ease of implementation into account with the Monte Carlo Localization Algorithm topping the list.

Before we dive into the specific localization algorithms, it is important to define the localization problem. There are three different types of localization problems [9]. These problems vary in complexity. The amount of information present and the nature of the environment that a robot is operating in determine the difficulty of the localization task.

The easiest localization problem is called **position tracking** or **local localization.** In this problem, the robot knows its initial pose and the localization challenge entails estimating the robot's pose as it moves around the environment. This problem is not as trivial as you might think since there is always some uncertainty in the robot motion. However, the uncertainty is limited to regions surrounding the robot.

A more complicated localization challenge is called **global localization.** In this case, the robot's initial pose is unknown and the robot must determine its pose relative to the ground truth map. The amount of uncertainty in global localization is much greater than that in position tracking.

The most challenging localization problem is the **kidnapped robot** problem. In robotics, the **kidnapped robot problem** commonly refers to a situation where an autonomous robot in operation is carried to an arbitrary location. The kidnapped robot problem creates significant issues with the robot's localization system, and only a subset of localization algorithms can successfully deal with the uncertainty created; it is commonly used to test a robot's ability to recover from catastrophic localization failures.

The four popular localization algorithms:

2.3.1. Markov Localization Algorithm

Markov localization algorithm is a realization of the Bayes Filter. The algorithm maintains a probability distribution over the set of all possible positions and orientations the robot might be located at. The algorithm addresses the position tracking problem, the global localization problem and the kidnapped robot problem in static environments [10]. However, Markov localization algorithm performs poorly if too many aspects of the environment are not covered in the world model.

2.3.2. Grid Localization Algorithm

Grid localization algorithm is referred to as histogram filter since it is capable of estimating the robot's pose using grids. The algorithm can solve the position tracking problem and the global localization problem. However, a number of issues arise when implementing grid localization:

- With a fine-grained grid, the computation required for a naïve implementation may make the algorithm intolerably slow.
- With a coarse grid, the additional information loss through the discretization negatively affects the filter and if not properly treated, may even prevent the filter from working [10].

2.3.3. Extended Kalman Filter (EKF) Localization Algorithm

The EKF is a variation of the Kalman Filter (KF). Therefore, we need to understand the KF before diving into the EKF.

Kalman Filter (KF)

The Kalman Filter can take data with a lot of uncertainty and noise and provide a very accurate estimate of the real value. The Kalman Filter works in a two-step process. The first step is a **measurement update**. We use the newly recorded measurement to update our last state. The second step is a **state prediction**. We use the information that we have about the current state to predict what the future state will be. At the start we use an initial guess. We continue to iterate through these two steps and it doesn't take many iterations for the estimate to converge on the real value.

Advantages of KF

- The Kalman Filter can very quickly develop an accurate estimate of the true value of the variable being measured
- The Kalman Filter does not require a lot of data to make an estimate

Disadvantages of KF

- The Kalman Filter cannot be used to solve non-linear problems

The EKF was developed to handle non-linear problems by linearizing the non-linear problem functions using first-order Taylor series expansion. The resultant linear problem functions are then implemented using an algorithm similar to the Kalman Filter.

Advantages of EKF

- EKF has lower computational cost in comparison to higher order approaches to non-linear filtering

Disadvantages of EKF

- EKFs are prone to linearization errors. A linearization error is the difference between a linear approximation and a non-linear function as illustrated in the figure below:

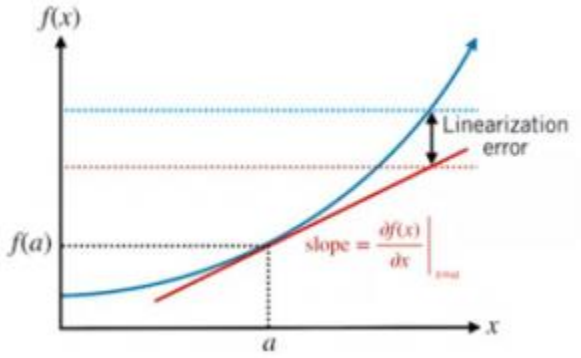


Figure 2.11 Linearization Error

- Computation of the Jacobian (used during the linearization of non-linear problem function) is prone to error
- KF and EKF can only solve the local localization problem [9]

2.3.4. Monte Carlo Localization (MCL) Algorithm

MCL is also known as particle filter localization algorithm because it estimates the robot's pose using particles. A particle can be thought of as a virtual element representing a robot. Each particle has a position and orientation, representing a guess of where a robot might be located. These particles are re-sampled each time the robot moves and senses its environment using range finder sensors. Eventually the algorithm zeroes in on the actual location of the robot.

MCL is limited to local and global localization problems and cannot recover from robot kidnapping. The Adaptive MCL (a variation of the MCL), on the other hand, is capable of solving the kidnapped robot problem. [9]

2.3.4.1. Advantages of MCL over EKF

- MCL represents non-Gaussian distributions and can approximate any other practical important distribution. MCL is not restricted by the linear Gaussian state space assumption as is the case of EKF.
- In MCL, one can control the computational memory and resolution of your solution by changing the number of particles distributed uniformly and randomly throughout the map.

Our project seeks to solve position tracking and global localization problems. EKF only solves the position tracking problem. Grid, Markov and MCL localization algorithms solve both. However, grid localization algorithm is intolerably slow when dealing with a fine-grained grid and discretization negatively affects the histogram filter when dealing with coarse grids. Markov localization algorithm performs well in static environments but the performance is poor if too many aspects of the environment are not covered in the world model. MCL algorithm is not affected by the problem mentioned above and can represent any probability distribution making it the best fit.

2.4. Computational motion planning

Computational motion planning is the ability of a robot to build a path between a starting point and a goal point taking into account both static and dynamic obstacles within the environment. There are several motion planning algorithms that are used to achieve computational motion planning. A motion planning algorithm scans through the configuration space and generates a sequence of motion commands or a sequence of states that the robot can visit in order to reach the target location.

Certain inputs need to be fed to the motion planning algorithm. These inputs include:

- The start pose/ configuration of the robot – This can be limited to position and orientation but, in some cases, the full state (the robot's starting velocity) is also taken into account.
- A desired goal pose / configuration – This can also be limited to position and orientation but, in some cases, the robot needs to arrive at the goal at a certain velocity. The goal configuration can also be multiple states.
- A geometric description of the robot – This is used by the planning algorithm to identify valid states avoiding collision with the environment or with other objects moving around.

- A geometric representation of the environment.

The outcome of feeding these inputs to a path planning algorithm is a collision free path from the start to the goal. The path provided can be the shortest path or the fastest path or an obstacle free path depending on the robot's application.

2.4.1. Classification of motion planning algorithms

Motion planning algorithms are primarily classified based on the technique used to scan the configuration space into uninformed motion planning algorithms and informed motion planning algorithms.

2.4.1.1. Uninformed motion planning algorithms

Uninformed motion planning algorithms blindly search through the configuration space [11]. They simply expand different nodes with the hope of reaching the goal at some point. Examples of uninformed motion planning algorithms include:

- Breadth-first search algorithm
- Depth-first search algorithm
- Uniform cost search algorithm etc.

Breadth-first search (BFS)

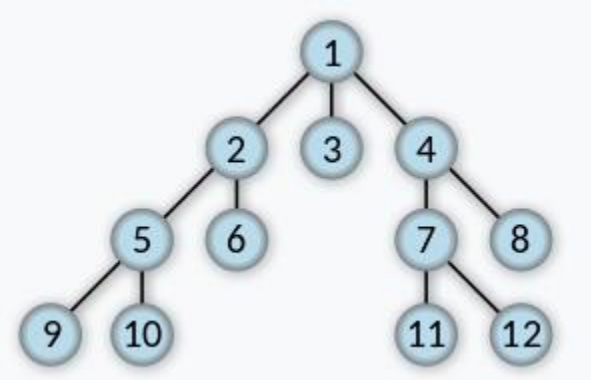


Figure 2.12 Breadth-first search

Breadth-first search starts at the root node and explores all the nodes at the present depth prior to moving on to the nodes at the next depth level. The figure above illustrates this.

Advantages

- With breadth-first search a path to the goal will be found if one exists. This makes breadth-first search a complete solution.
- If there is more than one path to the goal configuration, then BFS will provide the solution that requires the least number of steps to get to the goal (offers an optimal solution)

Disadvantages

- It requires a lot of memory since each level of the search tree must be saved into memory to expand the next level
- BFS needs a lot of time if the solution is far away from the root node.

Depth-first search (DFS)

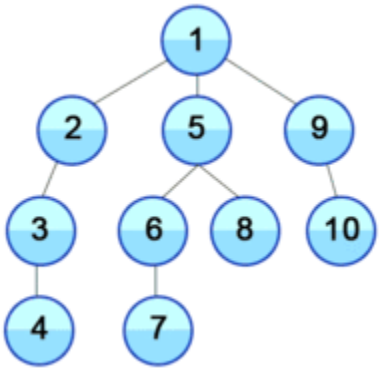


Figure 2.13 Depth-first search

Depth-first search starts at node 1. Assuming that the left edges are chosen before the right edges and assuming the search remembers previously visited nodes and will not repeat them, the nodes will be visited in the following order: 1,2,3,4,5,6,7,8,9,10 as illustrated in the figure above.

Depth-first search explores the node branch as far as possible before backtracking and expanding other nodes. Depth-first search is a complete solution if the robot is operating within a finite space.

Advantages

- DFS requires less memory as it only needs to store a stack of the nodes on the path from root node to the current node
- It takes less time to reach the goal than BFS algorithm (if it traverses in the right path)

Disadvantages

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution
- DFS algorithm may get lost in an infinite branch and never make it to the solution node.

Uniform cost search (UCS)

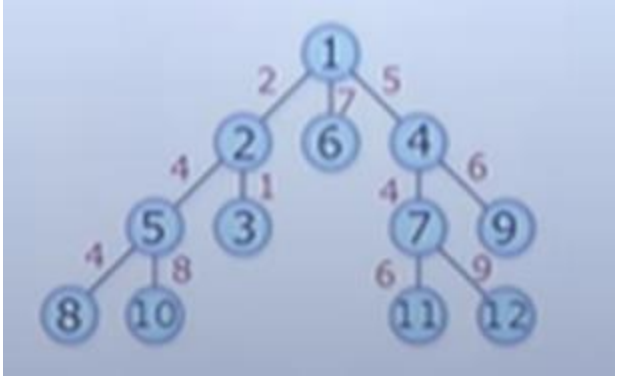


Figure 2.14 Uniform cost search

Is a cost sensitive motion planning algorithm. Transitions from one node to another are marked by different costs.

The algorithm takes into account the accumulated path cost (g). UCS expands the node which has the smallest accumulated path cost which leads to a behavior similar to breadth-first search if the transition costs were uniform. If we fully expand the search tree, taking into account the transition costs then we get the same result as **Dijkstra's algorithm**.

Advantages

- The solution obtained eventually is both complete and optimal (optimal because at every state the path with the least cost is chosen).

Disadvantages

- It does not care about the number of steps involved in searching and only concerned about path cost. Due to this, the algorithm may get stuck in an infinite loop.

In uninformed motion planning algorithms, time complexity (the amount of time taken to find a solution) and space complexity (the amount of memory needed to perform the search) are both relatively high (especially if the configuration space is large). Therefore, informed motion planning algorithms are needed.

2.4.1.2. Informed motion planning algorithms

Informed motion planning algorithms find the goal configuration faster by exploiting additional information about the goal. The additional information is encoded within a heuristic function [11]. A heuristic function estimates the cost from any configuration to the goal configuration. The better the estimate, the faster the goal configuration can be found. Certain bounds can be provided to limit the heuristic estimate. Examples of informed motion planning algorithms include:

- Greedy best-first search
- A*

Greedy best-first search

The cost estimate is the only thing taken into account in the simplest form of greedy search. Greedy search greedily explores the node that seems closest to the goal under the given heuristic.

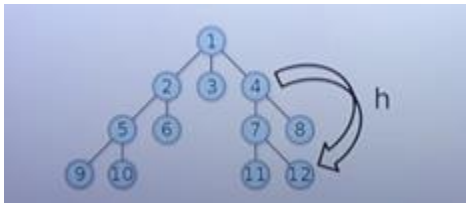


Figure 2.15 Greedy best first search

From the figure above:

Having expanded a certain number of states with the current state being state number four and the goal state being state number 12, the heuristic provides an estimate of the cost to get to the goal configuration.

The search algorithm ignores the accumulated path cost (how expensive it was to reach node four) and explores the nodes that promise the shortest distance to the goal under the provided heuristic.

Advantages

- Greedy best-first search finds the solution quickly provided that the heuristic estimate given is good.

Disadvantages

- Not guaranteed to find the optimal solution since the heuristic provided might not be the optimal solution

A*

A* is a combination of uniform-cost search and greedy search. A* jointly considers the accumulated path cost (g) and the heuristic (h) computing their sum to generate the fitness number (f). A* motion planning algorithm finds the shortest path through the configuration space.

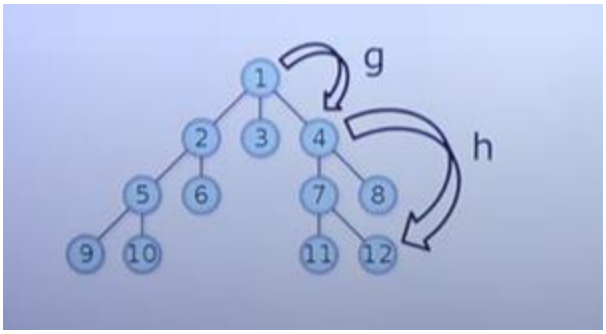


Figure 2.16 A search algorithm*

Advantages

- A* motion planning algorithm is both optimal and complete
- A* motion planning algorithm can solve very complex problems

Disadvantages

- The main drawback of A* is memory requirement as it keeps all generated nodes in memory

After exploring the uninformed and informed motion planning algorithms, it is important to note that an algorithm will primarily be chosen based on the robot's application. Our robot aims to achieve straight line navigation. Motion planning algorithms that can be exploited to achieve this goal are: breadth-first search, Dijkstra's algorithm and A*. Basing our evaluation on optimality, completeness, time complexity and space complexity of the motion planning algorithms, A* is the best fit [11].

2.5. OBSTACLE AVOIDANCE

Obstacle avoidance is one of the prime issues related to autonomous navigation of mobile robots. In order to maneuver in dynamic environments, a robot has to be equipped with an obstacle avoidance algorithm to deal with hurdles which are not known beforehand [12]. Obstacle avoidance can be classified in two sub stages called obstacle detection and collision avoidance. Different algorithms use different kinds of sensors for obstacle detection. Data received from the sensor is processed and the controller sends a signal to an end effector in order to avoid obstacles. The different techniques for obstacle avoidance are:

i. Bug Algorithm
ii. Artificial Potential Field Method
iii. Vector Field Histogram
iv. Elastic Band Concept
v. Fixed Sonar Method
vi. Optical Flow
vii. Collision Cone Method
viii. Fuzzy Logic Algorithm
ix. Neural Networks

2.5.1. BUG ALGORITHM

Bug Algorithm is the simplest algorithm to implement. It can further be divided into:

i. Bug 1 algorithm
ii. Bug 2 algorithm
iii. Distance Bug algorithm
iv. Tangent Bug algorithm

2.5.1.1. BUG 1 ALGORITHM

This method can be illustrated in three steps [12] [13]. One, head towards the goal. Two, if an obstacle is encountered, circumnavigate it and remember how close you get to the goal. Three, return to that closest point and continue. It is the simplest of all algorithms discussed in this work. It reaches to the goal almost all the times giving high reliability. But the matter of concern with this method is efficiency.

2.5.1.2. BUG 2 ALGORITHM

When the robot encounters an obstacle, it starts moving along the edge of the obstacle until it finds a point with the same slope as the goal [12], [13]. It starts moving on the line joining point of departure and goal. This algorithm is easy to program and provides better performance than Bug 1algorithm in majority of cases. Robot does not need to encircle the entire object as in the Bug 1 algorithm.

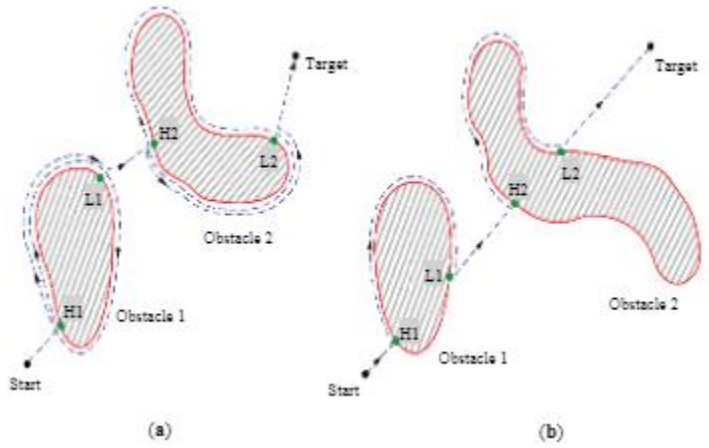


Figure 2.17 Bug algorithm

2.5.1.3. DISTANCE BUG ALGORITHM

The robot travels a shorter distance as compared to the earlier versions of Bug algorithm [12]. Its obstacle avoidance approach is different. When it encounters an obstacle, it follows the edge of the obstacle like the previous versions, but in addition it calculates and stores the distance from its current and next position to the goal. The departure point is selected if the condition that the distance from the goal and the next destination is greater than corresponding distance from the current distance, is met.

2.5.1.4. TANGENT BUG ALGORITHM

This approach finds tangents to the obstacle and calculates the distance of the goal from the point where the tangents touch the obstacles [12]. At times this algorithm can behave like bug 1, following the edges of an obstacle. A value d_{min} which is the shortest distance observed so far between the sensed boundary of the obstacle and the goal and d_{leave} which is the shortest distance between any point in the currently sensed environment and the goal are continuously updated. It terminates boundary following behavior when $d_{leave} < d_{min}$.

2.5.2. ARTIFICIAL POTENTIAL FIELD

Artificial Potential Field assumes robot and obstacle have the same polarity of electric potential whereas goal has an opposite polarity [12], [13]. This implies that robot is repelled by obstacle and attracted by goal. Robot moves in the direction of the resultant of all force vectors acting on it. This is simpler to implement and is an easy way to find the shorter edge of the obstacle. However, a local minima problem can cause process failure; a point other than the destination where the resultant forces are zero, hence falling into a trap.

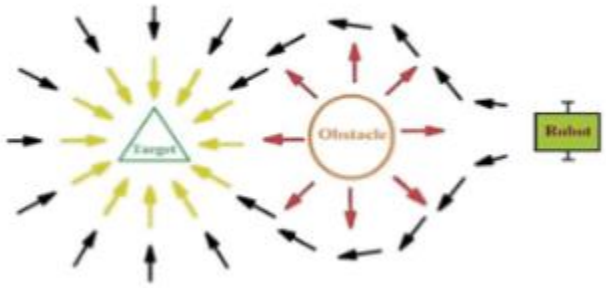


Figure 2.18 Artificial potential field

2.5.3. VECTOR FIELD HISTOGRAM

Vector Field Histogram constructs a polar histogram based on knowledge acquired through range sensors [12], [13]. Histogram is a graph between probabilities of presence of obstacle to angle associated to a sensor. The probabilities are obtained by creating a local occupancy grid map of the environment. The histogram is used to discover all the passages large enough to allow the robot to pass through. Selection of path is based on cost function which is a function of alignment of the robot's path with the goal, and on the difference between the current wheel orientation and the new direction. Minimum cost function is desirable.

It is a better method for detecting obstacle's shapes However, it requires longer time to 2-D map the obstacle. It also has high computational requirements and doesn't consider the vehicle's dynamics.

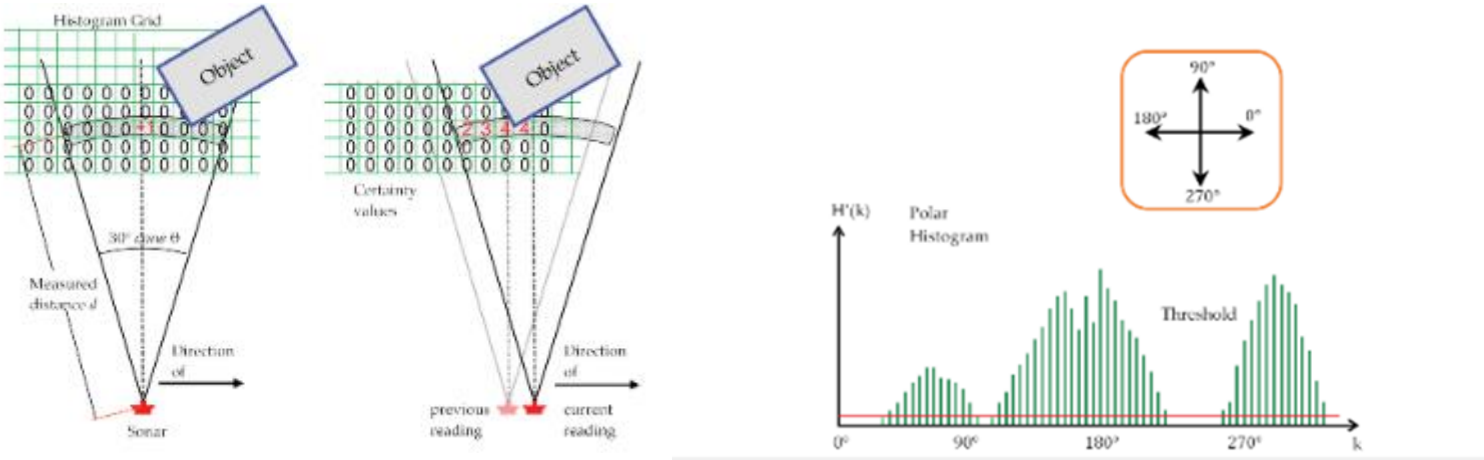


Figure 2.19 Histogram grid map and converted 1-D polar histogram

2.5.4. ELASTIC BAND TECHNIQUE

This method defines a bubble around the robot containing maximum free space, which can be travelled in any direction [12]. Shape and size of bubble are function of model of robot's geometry and range sensor information. A band of such bubbles can be formed for navigation of robot avoiding collision. It comprises distance sensors placed at fixed angular distance. Concept of bubble is used to judge whether the obstacle will cause a possible collision. Combining the profile of the boundary of the obstacle with the bubble concept, we can determine the maneuvers to avoid collisions.

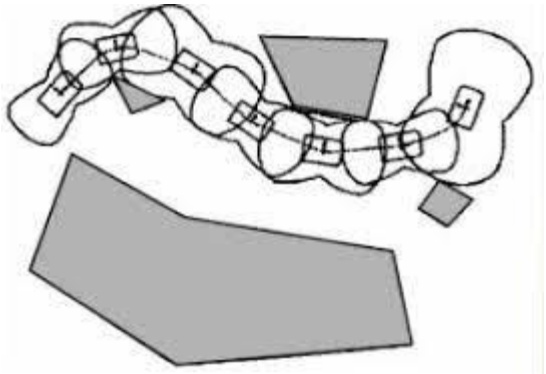


Figure 2.20 Elastic band

2.5.5. FIXED SONAR METHOD

Fixed sonar sensors are used to measure distance of obstacle from robot's current position [12]. Two sonar sensors are mounted on the front part of robot. Position of obstacle is classified in different zones based on combinations of sensor readings. Distance can be obtained by velocity formulae, where the velocity of sound in air = 340 m/s is used. In this method care has to be taken in placing sonar sensors so that they do not detect ground as obstacle. Ease of implementation and less computation requirements are important merits of this method.

2.5.6. OPTICAL FLOW

Is used to detect dynamic objects only. It uses a camera to detect such obstacles [12]. As the camera moves through the environment, the pattern of light reflected to the obstacle changes continuously, creating an optical flow. Optic flow is the pattern of apparent motion of objects in a visual scene caused by relative motion between the camera and the scene. The aim of creating an optical flow pattern is to calculate the relative motion between two image frames taken consecutively at each position. This flow indicates any moving object in the way of the robot.

2.5.7. COLLISION CONE METHOD

The collision cone concept was first proposed for a 2- D movement scenario [13]. It assumes any obstacle as a circular area, and the distance from the robot's position to the obstacle area is calculated within a collision cone using the robot's velocity vector. This concept works for any irregular-shaped unknown obstacle and prevents collision between two irregular-shaped objects or vehicles. Later on, this method has been extended to detect moving obstacles in 3-D space.

This approach creates simpler avoidance paths. It uses vehicle dynamics to create avoidance paths. It uses the minimum effort for guidance control and doesn't consider the shape of the obstacle.

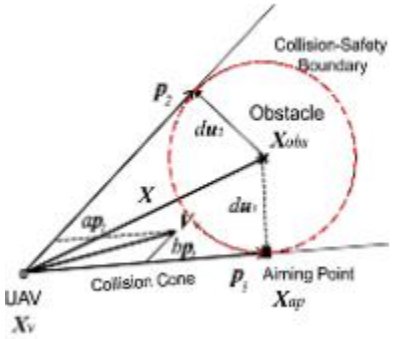


Figure 2.21 Collision cone

2.5.8. *FUZZY LOGIC ALGORITHM*

Fuzzy logic uses the fuzzy controller. To use fuzzy logic in any system, the operator needs to assign a set of data or knowledge to create fuzzy sets that will be used to avoid obstacles or navigate the mobile robot. This process of assigning fuzzy input sets is called fuzzification [13]. The set value usually can be anywhere between two traditional logics, such as (0, 1) or (Low, High) or (Cold, Hot), etc. This is why, usually, those vehicles that use fuzzy logic for navigation and avoidance use one kind of multiple sensors or sensor fusion.

It is robust and suitable for dynamic environments. However, it needs multisensory systems and polyhedral shapes may increase computational calculation.

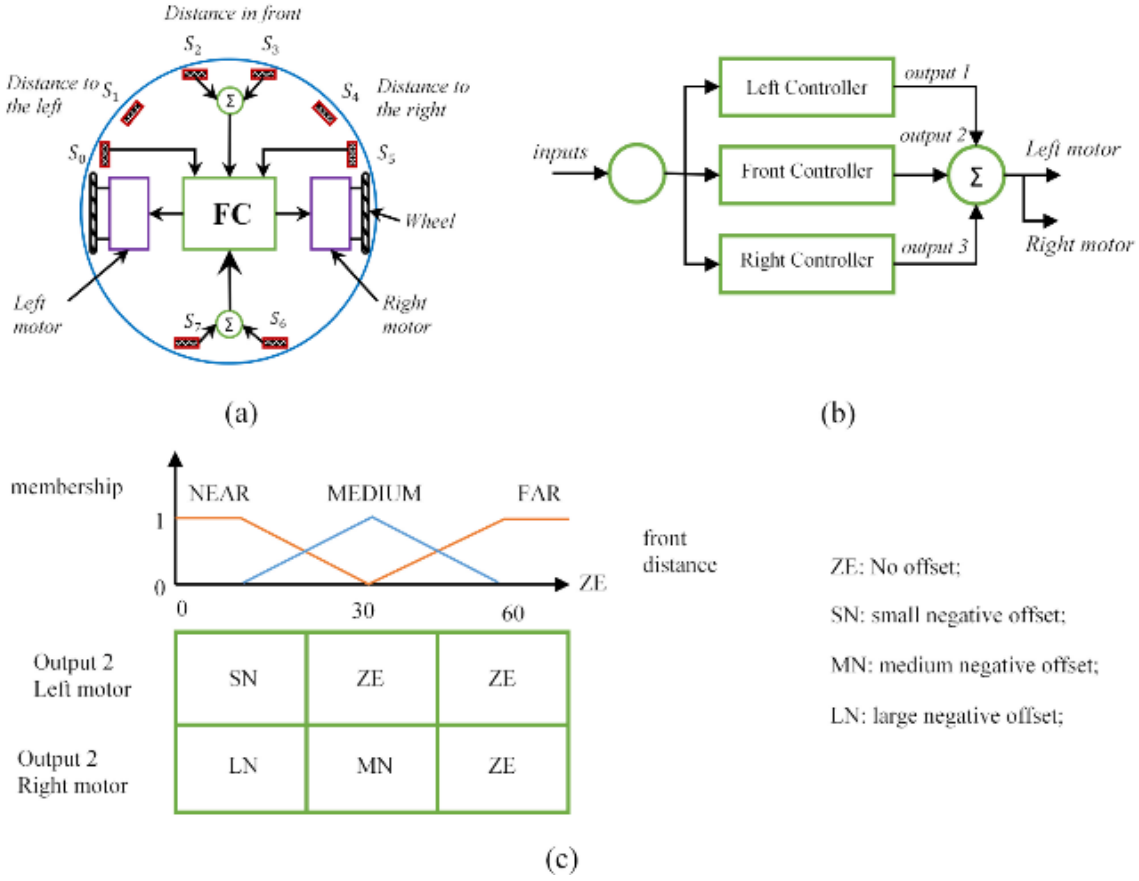


Figure 2.22 Usage of fuzzy logic control

2.5.9. NEURAL NETWORK

Neural network algorithms are inspired by the brain [13]. The neural network take in data, train themselves to recognize the patterns in the data, and then predict the outputs for a new set of similar data. A computational model repeats training or functions with a biological neural network system until the best result comes out. The dynamic neural network is capable of automatically adjusting its structure, following the complication of the vehicle's environment, understanding the mapping connection amongst the vehicle's state and its obstacle avoidance decision in real-time, and efficiently decreasing the vehicle's computational load.

It is good for known obstacle environment and has better performance for real-time avoidance. However, it needs lots of training data before performance.

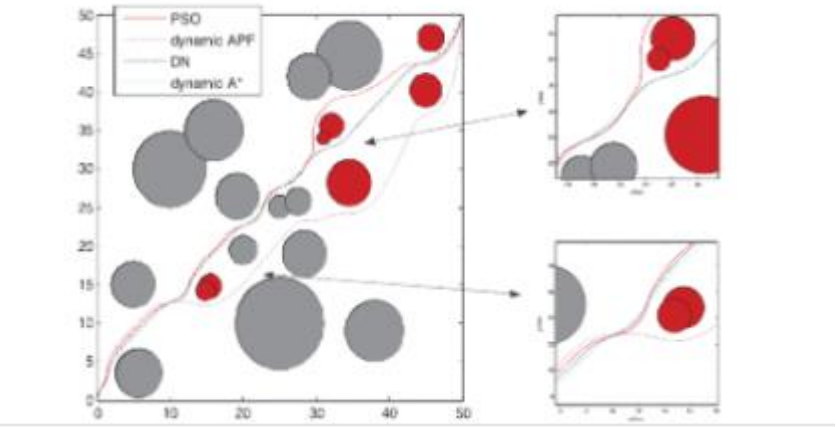


Figure 2.23 Collision avoidance performance using training data

2.5.10. SUMMARY

This project will limit itself to implementing the Bug 1 algorithm for obstacle avoidance. Using this approach, the robotic lawn mower will mow around an obstacle before exiting to continue following its path.

2.6. ROS

The Robot Operating System (ROS) is a set of software libraries and tools that help build robot applications from drivers to state-of-the-art algorithms and powerful developer tools for robotics projects.

2.6.1. ROS CONCEPTS - PACKAGES

All ROS software is organized into packages. A ROS package is a coherent collection of files, generally including both executables and supporting files, that serves a specific purpose.

Each package is defined by amanifest, which is a file called package.xml. This file defines some details about the package, including its name, version, maintainer, and dependencies. The directory containing package.xml is called the package directory. (In fact, this is the definition of a ROS package: Any directory that ROS can find that contains a file named package.xml is a package directory.

You can obtain a list of all of the installed ROS packages using this command:

```
rospack list
```

2.6.2. ROS CONCEPTS - MASTER

One of the basic goals of ROS is to enable roboticists to design software as a collection of small, mostly independent programs called nodes that all run at the same time. For this to work, those nodes must be able to communicate with one another. The part of ROS that facilitates this communication is called the ROS master. To start the master, use this command:

```
roscore
```

2.6.3. ROS CONCEPTS - NODES

Once you've started roscore, you can run programs that use ROS. A running instance of a ROS program is called a node.

The basic command to create a node (also known as "running a ROS pro-gram") is rosrun:

```
rosrunpackage-name executable-name
```

There are two required parameters to rosrun. The first parameter is a package name. The second parameter is simply the name of an executable file within that package.

2.6.4. ROS CONCEPTS – TOPICS AND MESSAGES

The primary mechanism that ROS nodes use to communicate is to send messages. Messages in ROS are organized into named topics. The idea is that a node that wants to share information will publish messages on the appropriate topic or topics; a node that wants to receive information will subscribe to the topic or topics that it's interested in. The ROS master takes care of ensuring that publishers and subscribers can find each other; the messages themselves are sent directly from publisher to subscriber.

2.6.4.1. TOPICS AND MESSAGES: VIEWING THE GRAPH

This idea is probably easiest to see graphically, and the easiest way to visualize the publish-subscribe relationships between ROS nodes is to use this command:

rqt_graph

In this name, the r is for ROS, and the qt refers to the qt GUI toolkit used to implement the program.

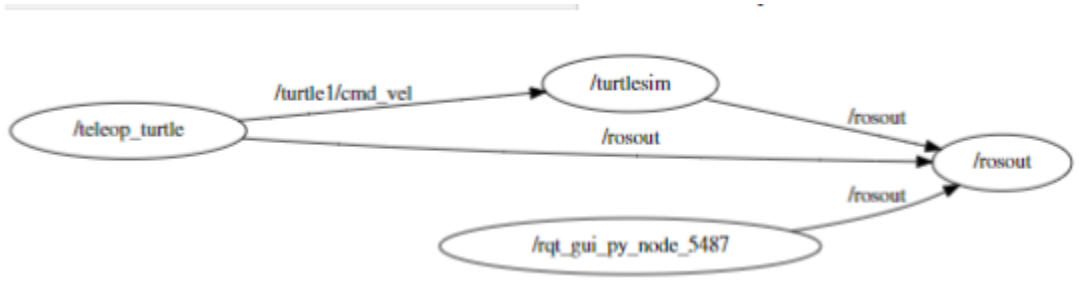


Figure 2.24 Collision avoidance performance using training data

2.6.4.2. TOPICS AND MESSAGES: MESSAGES AND MESSAGE TYPES

To get a list of active topics, use this command:

rostopic list

You can see the actual messages that are being published on a single topic using the rostopic command:

rostopic echo topic-name

There are also two commands for measuring the speed at which messages are published and the bandwidth consumed by those messages:

rostopic hz topic-name

rostopic bw topic-name

You can learn more about a topic using the rostopic info command:

rostopic info topic-name

To see details about a message type, use a command likethis:

rosmsg show message-type-name

Like everything else in ROS, every message type belongs to a specific package. Message type names always contain a slash, and the part before the slash is the name of the containing package:

package-name/type-name

The main idea here is that topics and messages are used for many-to-many communication. Many publishers and many subscribers can share a single topic.

ROS does provide a mechanism, called services, for slightly more direct, one-to-one communication. This secondary technique is much less common, but does have its uses.

2.6.5. ROS CONCEPTS – WRITING ROS PROGRAMS

We've introduced a few core ROS features, including packages, nodes, topics, and messages. Now we write the ROS programs. Before we write any programs, the first steps are to create a workspace to hold our packages, and then to create the package itself.

2.6.5.1. WRITING ROS PROGRAMS – CREATING A WORKSPACE

Packages that you create should live together in a directory called a workspace. We use the normal command and refer to this new directory as our workspace directory:

mkdir

2.6.5.2. WRITING ROS PROGRAMS – CREATING A PACKAGE

The command to create a new ROS package, which should be run from the src directory of your workspace, looks like this:

```
catkin_create_pkgpackage-name
```

It creates a directory to hold the package and creates two configuration files inside that directory:

2. The first configuration file, calledpackage.xml
3. The second file, called CMakeLists.txt, is a script for an industrial-strength cross-platform build system called CMake. It contains a list of build instructions including what executables should be created, what source files to use to build each of them, and where to find the include files and libraries needed for those executables. CMake is used internally by catkin.

2.6.5.3. WRITING ROS PROGRAMS – COMPILING THE PROGRAM

Compiling and running of our programs is done by ROS' build system called catkin. There are four steps

1. Declaring dependencies.
   First, we need to declare the other packages on which ours depends. For C++ programs, this step is needed primarily to ensure that catkin provides the C++compiler with the appropriate flags to locate the header files and libraries that it needs. To list dependencies, edit the CMakeLists.txt in your package directory.
2. Declaring an executable.
   Next, we need to add two lines to CMakeLists.txt declaring the executable we would like to create. The general form is:

   ```
   add_executable(executable-name source-files)
   target_link_libraries(executable-name${catkin_LIBRARIES})
   ```

   The first line declares the name of the executable we want, and a list of source files that should be combined to form that executable. If you have more than one source file, list them all here, separated by spaces. The second line tells CMake to use the appropriate library flags (defined by the find_packageline above) when linking this executable. If your package contains more than one executable, copy and modify these two lines for each executable you have.

3. Building the workspace.

   Once your CMakeLists.txt is set up, you can build your work-space—including compiling all of the executables in all of its packages—using this command:

   catkin_make

   Because it's designed to build all of the packages in your workspace, this command must be run from your workspace directory. It will perform several configuration steps (especially the first time you run it) and create subdirectories called devel and build within your workspace. These two new directories contain build-related files like automatically-generated make files, object code, and the executables themselves. If you like, the devel and build subdirectories can safely be deleted when you've finished working on your package.

4. Sourcing setup.bash

   The final step is to execute a script called setup.bash, which is created by catkin_make inside the devel subdirectory of your workspace:

   source devel/setup.bash

   This automatically-generated script sets several environment variables that enable ROS to find your package and its newly-generated executables.

2.6.5.4. WRITING ROS PROGRAMS – EXECUTING THE PROGRAM

Execute:

Roscore

Then since all of those build steps are complete, your new ROS program is ready to execute using:

rosrun

CHAPTER THREE:

3. METHODOLOGY:

3.1.SIMULATION SETUP

3.1.1. ROS ESSENTIALS (GO CHASE IT)

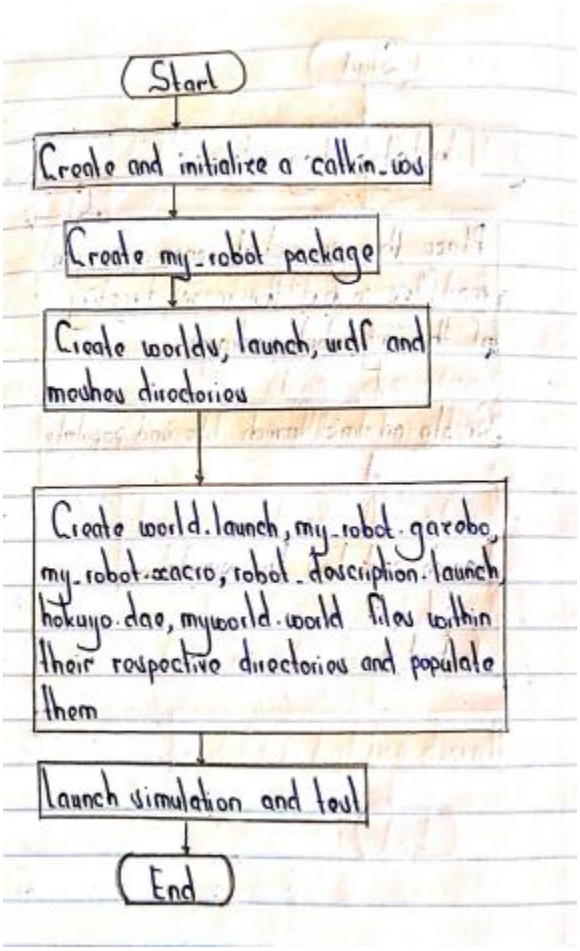


Figure 3.1 Ros essentials flowchart

1. Created and initialized a catkin_ws:

```
$ mkdir catkin_wsre
$ cd catkin_wsre
$ mkdir src
$ cd src
$ catkin_init_workspace
```

2. Within the src directory of the catkin_ws, we created my_robot package:

```
$ catkin_create_pkg my_robot
```

3. Created a worlds and a launch directory within the my_robot package:

```
$ cd my_robot
$ mkdir launch
$ mkdir worlds
```

4. Created the world.launch file:

```
$ cd launch
$ touch world.launch
```

5. Added the following to world.launch:

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
<!-- Robot pose -->
  <arg name="x" default="-4.638987"/>
  <arg name="y" default="-0.980767"/>
  <arg name="z" default="0.100000"/>
  <arg name="roll" default="-0.000017"/>
  <arg name="pitch" default="0.000669"/>
  <arg name="yaw" default="1.506235"/>
  <!-- Launch other relevant files-->
  <include file="$(find my_robot)/launch/robot_description.launch"/>
  <!-- World File -->
  <arg name="world_file" default="$(find my_robot)/worlds/myworld.world"/>
  <!-- Launch Gazebo World -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="use_sim_time" value="true"/>
    <arg name="debug" value="false"/>
    <arg name="gui" value="true" />
```

```
    <arg name="world_name" value="$(arg world_file)"/>
  </include>
<!-- Find my robot Description-->
  <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find
my_robot)/urdf/my_robot.xacro'"/>
  <!-- Spawn My Robot -->
<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn="false"
output="screen"
args="-urdf -param robot_description -model my_robot
-x $(arg x) -y $(arg y) -z $(arg z)
-R $(arg roll) -P $(arg pitch) -Y(arg yaw)"/>
<!--launch rviz-->
<node name="rviz" pkg="rviz" type="rviz" respawn="false"/>
</launch>
```

6. Created an urdf directory:

```
$ cd ..
$ mkdir urdf
```

7. Within the urdf directory, we added the downloaded my_robot.gazebo file.

8. Created the robot's xacro file inside the urdf directory:

```
$ cd urdf
$ touch my_robot.xacro
```

9. Added the following code into the my_robot.xacro file:

```
<?xml version='1.0'?>
<robot name="my_robot" xmlns:xacro="https://www.ros.org/wiki/xacro">
<xacro:include filename="$(find my_robot)/urdf/my_robot.gazebo" />
<link name="robot_footprint"></link>
<joint name="robot_footprint_joint" type="fixed">
```

```
<origin xyz="0 0 0" rpy="0 0 0" />
<parent link="robot_footprint"/>
<child link="chassis" />
</joint>
<link name='chassis'>
<pose>0 0 0.1 0 0 0</pose>
<inertial>
<mass value="15.0"/>
<origin xyz="0.0 0 0" rpy="0 0 0"/>
<inertia
ixx="0.1" ixy="0" ixz="0"
iyy="0.1" iyz="0"
izz="0.1"
/>
</inertial>
<collision name='collision'>
<origin xyz="0 0 0" rpy="0 0 0"/>
<geometry>
<box size=".4 .2 .1"/>
</geometry>
</collision>
<visual name='chassis_visual'>
<origin xyz="0 0 0" rpy="0 0 0"/>
<geometry>
<box size=".4 .2 .1"/>
</geometry>
</visual>
<collision name='back_caster_collision'>
<origin xyz="-0.15 0 -0.05" rpy="0 0 0"/>
<geometry>
<sphere radius="0.0499"/>
```

```
</geometry>
</collision>
<visual name='back_caster_visual'>
<origin xyz="-0.15 0 -0.05" rpy="0 0 0"/>
<geometry>
<sphere radius="0.05"/>
</geometry>
</visual>
<collision name='front_caster_collision'>
<origin xyz="0.15 0 -0.05" rpy="0 0 0"/>
<geometry>
<sphere radius="0.0499"/>
</geometry>
</collision>
<visual name='front_caster_visual'>
<origin xyz="0.15 0 -0.05" rpy="0 0 0"/>
<geometry>
<sphere radius="0.05"/>
</geometry>
</visual>
</link>
<gazebo reference="chassis">
<material>Gazebo/Blue</material>
</gazebo>
<link name="left_wheel">
<inertial>
<origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
<mass value="5"/>
<inertia ixx="0.1" ixy="0" ixz="0" iyy="0.1" iyz="0" izz="0.1"/>
</inertial>
<visual>
```

```
<origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
<geometry>
<cylinder length="0.05" radius="0.1"/>
</geometry>
</visual>
<collision>
<origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
<geometry>
<cylinder length="0.05" radius="0.1"/>
</geometry>
</collision>
</link>
<gazebo reference="left_wheel">
<material>Gazebo/Red</material>
</gazebo>
<link name="right_wheel">
<inertial>
<origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
<mass value="5"/>
<inertia ixx="0.1" ixy="0" ixz="0" iyy="0.1" iyz="0" izz="0.1"/>
</inertial>
<visual>
<origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
<geometry>
<cylinder length="0.05" radius="0.1"/>
</geometry>
</visual>
<collision>
<origin xyz="0 0 0" rpy="0 1.5707 1.5707"/>
<geometry>
<cylinder length="0.05" radius="0.1"/>
```

```
</geometry>
</collision>
</link>
<gazebo reference="right_wheel">
<material>Gazebo/Red</material>
</gazebo>
<joint type="continuous" name="left_wheel_hinge">
<origin xyz="0 0.15 0" rpy="0 0 0"/>
<child link="left_wheel"/>
<parent link="chassis"/>
<axis xyz="0 1 0" rpy="0 0 0"/>
<limit effort="1000" velocity="1000"/>
<dymanic damping="1.0" friction="1.0"/>
</joint>
<joint type="continuous" name="right_wheel_hinge">
<origin xyz="0 -0.15 0" rpy="0 0 0"/>
<child link="right_wheel"/>
<parent link="chassis"/>
<axis xyz="0 1 0" rpy="0 0 0"/>
<limit effort="1000" velocity="1000"/>
<dymanic damping="1.0" friction="1.0"/>
</joint>
<link name="camera">
<inertial>
<origin xyz="0 0 0" rpy="0 0 0"/>
<mass value="0.1"/>
<inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6"/>
</inertial>
<visual>
<origin xyz="0 0 0" rpy="0 0 0"/>
<geometry>
```

```
<box size=".05 .05 .05"/>
</geometry>
</visual>
<collision>
<origin xyz="0 0 0" rpy="0 0 0"/>
<geometry>
<box size=".05 .05 .05"/>
</geometry>
</collision>
<box_inertia sizeX="0.05" sizeY="0.05" sizeZ="0.05" mass="0.1">
<origin xyz="0 0 0" rpy="0 0 0"/>
</box_inertia>
</link>
<gazebo reference="camera">
<material>Gazebo/Red</material>
</gazebo>
<joint type="fixed" name="camera_joint">
<origin xyz="0.2 0 0" rpy="0 0 0"/>
<child link="camera"/>
<parent link="chassis"/>
<axis xyz="0 1 0" rpy="0 0 0"/>
</joint>
<link name="hokuyo">
<inertial>
<origin xyz="0 0 0" rpy="0 0 0"/>
<mass value="1e-5"/>
<inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6"/>
</inertial>
<visual>
<origin xyz="0 0 0" rpy="0 0 0"/>
<geometry>
```

<mesh filename="package://my_robot/meshes/hokuyo.dae"/>
</geometry>
</visual>
<collision>
<origin xyz="0 0 0" rpy="0 0 0"/>
<geometry>
<box size=".1 .1 .1"/>
</geometry>
</collision>
</link>
<joint type="fixed" name="hokuyo_joint">
<origin xyz="0.15 0 .1" rpy="0 0 0"/>
<child link="hokuyo"/>
<parent link="chassis"/>
<axis xyz="0 1 0" rpy="0 0 0"/>
</joint>
</robot>

10. Created a launch file for the robot:

```
$ cd ..
$ cd launch
$ touch robot_description.launch
```

11. Added the following to robot_description.launch file:

<?xml version="1.0"?>
<launch>
<!-- send urdf to param server -->
<param name="robot_description" command="$(find xacro)/xacro --inorder '$(find my_robot)/urdf/my_robot.xacro'" />
<!-- Send fake joint values-->
<node name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher">

```
  <param name="use_gui" value="false"/>
 </node>
 <!-- Send robot states to tf -->
 <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" respawn="false" output="screen"/>
</launch>
```

12. Created a meshes directory (within the my_robot package directory) to hold the downloaded hokuyo.dae file.
13. Navigated to the worlds directory and added myworld.world file.

```
$ cd ..
$ cd worlds
```

14. Opened a new terminal and ran the following commands:

```
$ cd catkin_wsre
$ catkin_make
$ source devel/setup.bash
$ roslaunch my_robot world.launch
```

3.1.2. LOCALIZATION (WHERE AM I?)

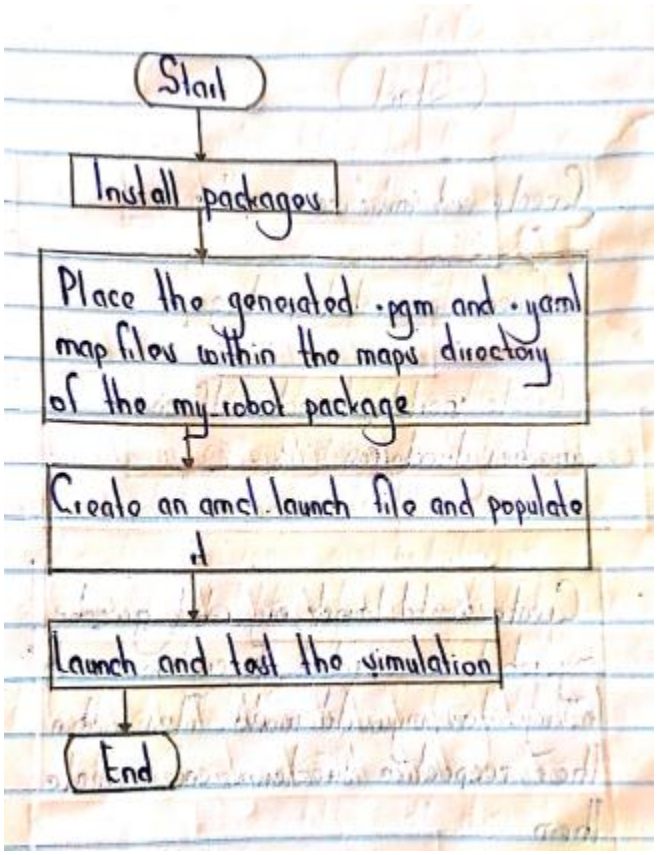


Figure 3.2 Localization flowchart

1. Packages installed for native ROS installation or virtual machine:

```
$ sudo apt-get install ros-kinetic-navigation
$ sudo apt-get install ros-kinetic-map-server
$ sudo apt-get install ros-kinetic-move-base
$ sudo apt-get install ros-kinetic-amcl
```

2. Navigated to my_robot folder and created a maps folder to store the generated pgm map file:

```
$ cd catkin_wsl/src/my_robot
$ mkdir maps
```

3. Installed dependencies to compile the pgm_map_creator:

```
sudo apt-get install libignition-math2-dev protobuf-compiler
```

4. Cloned the pgm_map_creator package to the src folder and built the package:

```
cd ..
git clone https://github.com/udacity/pgm_map_creator.git
cd ..
catkin_make
```

5. Copied the gazebo world created and pasted it in the world folder within the pgm_map_creator package folder.

6. Inserted the map creator plugin into our world file. We then opened the world file using gedit and added the following tag towards the end of the file, but just before `</world>` tag:

```
<plugin filename="libcollision_map_creator.so" name="collision_map_creator"/>
```

7. Opened a new terminal and ran gzserver with the world file:

```
cd catkin_wsl
source devel/setup.bash
gzserver src/pgm_map_creator/world/myworld.world
```

8. Opened another terminal and ran the request_publisher node:

```
cd catkin_wsl
source devel/setup.bash
```

```
roslaunch pgm_map_creator request_publisher.launch
```

We waited for the plugin to generate a map. It was located within the map folder of the `pgm_map_creator` package folder.

9. Added the map to my_robot package folder within the maps folder.
10. Created a yaml file to provide metadata about the map:

```
cd catkin_wsl/src/my_robot/maps
touch map.yaml
```

11. Opened the `yaml` file and added the following lines to it:

```
image: map.pgm
resolution: 0.01
origin: [-15, -15, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 0
```

12. Created a launch file for the AMCL nodes:

```
$ cd catkin_wsl/src/my_robot/launch/
$ gedit amcl.launch
```

13. Obtained the following config files that were later loaded into the AMCL launch file:

```
$ cd ..
$ mkdir config
$ cd config

wget https://s3-us-west-1.amazonaws.com/udacity-robotics/Resource/where_am_i/config.zip
```

```
unzip config.zip

rm config.zip
```

14. Added the following tags to amcl.launch:

```
<?xml version="1.0"?>
<launch>
 <!-- Map Server -->
<arg name="map_file" default="$(find my_robot)/maps/map.yaml"/>
<node name="map_server" pkg="map_server" type="map_server" args="$(arg
map_file)" />
<!-- AMCL Node -->
  <node name="amcl" pkg="amcl" type="amcl" output="screen">
  <remap from="scan" to="/scan"/>
  <param name="odom_frame_id" value="odom"/>
  <param name="odom_model_type" value="diff-corrected"/>
  <param name="base_frame_id" value="robot_footprint"/>
  <param name="global_frame_id" value="map"/>
  <param name="initial_pose_x" value="-0.999"/>
  <param name="initial_pose_y" value="4.723"/>
  <param name="initial_pose_a" value="-1.560"/>
<!-- Tune parameters -->
<param name="transform_tolerance" value="0.2"/>
<param name="min_particles" value="100"/>
<param name="max_particles" value="3000"/>
<param name="update_min_a" value="pi/6.0"/>
<param name="update_min_d" value="0.1"/>
<param name="laser_min_range" value="0.1"/>
<param name="laser_max_range" value="10"/>
<param name="laser_max_beams" value="30"/>
```

```
<param name="laser_z_hit" value="0.95"/>
<param name="laser_z_rand" value="0.05"/>
<param name="odom_alpha1" value="0.02"/>
<param name="odom_alpha2" value="0.02"/>
<param name="odom_alpha3" value="0.02"/>
<param name="odom_alpha4" value="0.02"/>
<param name="odom_alpha5" value="0.02"/>
  </node>
  <!-- Move Base -->
  <node name="move_base" pkg="move_base" type="move_base" respawn="false"
output="screen">
<remap from="scan" to="/scan"/>
<param name="base_global_planner" value="navfn/NavfnROS" />
  <param name="base_local_planner"
value="base_local_planner/TrajectoryPlannerROS"/>
<rosparam file="$(find my_robot)/config/costmap_common_params.yaml"
command="load" ns="global_costmap" />
  <rosparam file="$(find my_robot)/config/costmap_common_params.yaml"
command="load" ns="local_costmap" />
  <rosparam file="$(find my_robot)/config/local_costmap_params.yaml"
command="load" />
  <rosparam file="$(find my_robot)/config/global_costmap_params.yaml"
command="load" />
  <rosparam file="$(find my_robot)/config/base_local_planner_params.yaml"
command="load" />
  </node>
</launch>
```

15. Cloned the ros-teleop package to the src folder, built the package and sourced the setup script:

```
cd catkin_wsl/src
```

```
git clone https://github.com/ros-teleop/teleop_twist_keyboard

cd ..

catkin_make

source devel/setup.bash
```

16. Launched the simulation:

```
$ cd catkin_wsl/
$source devel/setup.bash
$ roslaunch my_robot world.launch
```

17. In a new terminal, we launched the amcl launch file:

```
$source devel/setup.bash
$ roslaunch my_robot amcl.launch
```

18. RViz configuration set:
    - Selected map for fixed frame
    - Clicked the "Add" button and:
        - added RobotModel
        - added Map and selected first topic/map
        - added PoseArray and selected topic/particlecloud
19. Sent a 2D Nav Goal from RViz and observed the robot localize itself in the environment.
20. The robot could also be controlled using the teleop node. Ran the teleop script as is described in the README file:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

3.1.3. MAPPING (MAP MY WORLD)

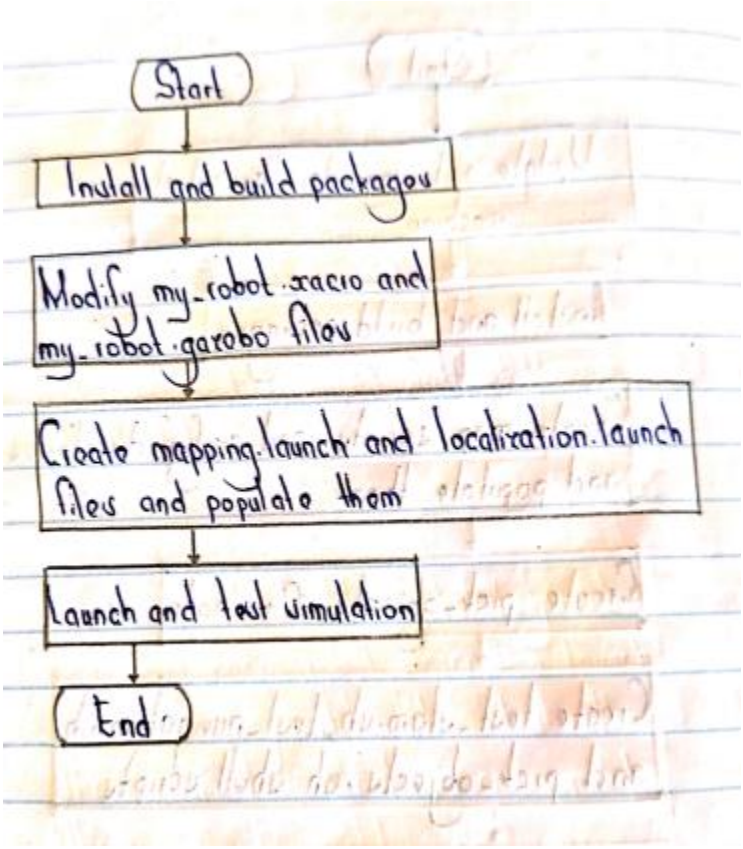


Figure 3.3 Mapping flowchart

1. Used the commands below to install and build rtabmap and rtabmap-ros packages for the native ROS installation or virtual machine:

```
$ sudo apt-get install ros-kinetic-rtabmap-ros
$ source /opt/ros/kinetic/setup.bash
$ source ~/catkin_wsm/devel/setup.bash
$ sudo apt install ros-kinetic-rtabmap
$ cd ~
$ git clone https://github.com/introlab/rtabmap.git rtabmap
$ cd rtabmap/build
$ cmake ..
$ make
$ sudo make install
```

$ cd ~/catkin_wsm

$ git clone https://github.com/introlab/rtabmap_ros.git src/rtabmap_ros

$ catkin_make -j4

2. Added the following to the robot's xacro file:

```
<joint name="camera_optical_joint" type="fixed">
  <origin xyz="0 0 0" rpy="-1.5707 0 -1.5707"/>
  <parent link="camera"/>
  <child link="camera_link_optical"/>
 </joint>
 <link name="camera_link_optical">
 </link>
```

3. Replaced the existing camera and its shared object file within my_robot.gazebo file:

```
<!-- RGBD Camera -->
 <gazebo reference="camera">
  <sensor type="depth" name="camera1">
    <always_on>1</always_on>
    <update_rate>20.0</update_rate>
    <visualize>true</visualize>
    <camera>
      <horizontal_fov>1.047</horizontal_fov>
      <image>
        <width>640</width>
        <height>480</height>
        <format>R8G8B8</format>
      </image>
      <depth_camera>
      </depth_camera>
      <clip>
        <near>0.1</near>
```

```
            <far>20</far>
          </clip>
        </camera>
        <plugin name="camera_controller" filename="libgazebo_ros_openni_kinect.so">
          <alwaysOn>true</alwaysOn>
          <updateRate>10.0</updateRate>
          <cameraName>camera</cameraName>
          <frameName>camera_link_optical</frameName>
          <imageTopicName>rgb/image_raw</imageTopicName>
          <depthImageTopicName>depth/image_raw</depthImageTopicName>
          <pointCloudTopicName>depth/points</pointCloudTopicName>
          <cameraInfoTopicName>rgb/camera_info</cameraInfoTopicName>
          <depthImageCameraInfoTopicName>depth/camera_info</depthImageCameraInfoTopic
Name>
          <pointCloudCutoff>0.4</pointCloudCutoff>
            <hackBaseline>0.07</hackBaseline>
            <distortionK1>0.0</distortionK1>
            <distortionK2>0.0</distortionK2>
            <distortionK3>0.0</distortionK3>
            <distortionT1>0.0</distortionT1>
            <distortionT2>0.0</distortionT2>
          <CxPrime>0.0</CxPrime>
          <Cx>0.0</Cx>
          <Cy>0.0</Cy>
          <focalLength>0.0</focalLength>
          </plugin>
    </sensor>
  </gazebo>
```

4. Created a mapping.launch file and added the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<launch>
  <!-- Arguments for launch file with defaults provided -->
  <arg name="database_path"     default="$(find my_robot)/rtabmap.db"/>
  <arg name="rgb_topic"   default="/camera/rgb/image_raw"/>
  <arg name="depth_topic" default="/camera/depth/image_raw"/>
  <arg name="camera_info_topic" default="/camera/rgb/camera_info"/>
  <!-- Mapping Node -->
  <group ns="rtabmap">
    <node name="rtabmap" pkg="rtabmap_ros" type="rtabmap" output="screen" args="--delete_
db_on_start">
      <!-- Basic RTAB-Map Parameters -->
      <param name="database_path"       type="string" value="$(arg database_path)"/>
      <param name="frame_id"            type="string" value="robot_footprint"/>
      <param name="odom_frame_id"       type="string" value="odom"/>
      <param name="subscribe_depth"     type="bool"   value="true"/>
      <param name="subscribe_scan"      type="bool"   value="true"/>
      <!-- RTAB-Map Inputs -->
      <remap from="scan" to="/scan"/>
      <remap from="rgb/image" to="$(arg rgb_topic)"/>
      <remap from="depth/image" to="$(arg depth_topic)"/>
      <remap from="rgb/camera_info" to="$(arg camera_info_topic)"/>
      <!-- RTAB-Map Output -->
      <remap from="grid_map" to="/map"/>
      <!-- Rate (Hz) at which new nodes are added to map -->
      <param name="Rtabmap/DetectionRate" type="string" value="1"/>
      <!-- 2D SLAM -->
      <param name="Reg/Force3DoF" type="string" value="true"/>
      <!-- Loop Closure Detection -->
      <!-- 0=SURF 1=SIFT 2=ORB 3=FAST/FREAK 4=FAST/BRIEF 5=GFTT/FREAK 6=GFTT
/BRIEF 7=BRISK 8=GFTT/ORB 9=KAZE -->
      <param name="Kp/DetectorStrategy" type="string" value="0"/>
```

```
    <!-- Maximum visual words per image (bag-of-words) -->
    <param name="Kp/MaxFeatures" type="string" value="400"/>
    <!-- Used to extract more or less SURF features -->
    <param name="SURF/HessianThreshold" type="string" value="100"/>
    <!-- Loop Closure Constraint -->
    <!-- 0=Visual, 1=ICP (1 requires scan)-->
    <param name="Reg/Strategy" type="string" value="0"/>
    <!-- Minimum visual inliers to accept loop closure -->
    <param name="Vis/MinInliers" type="string" value="15"/>
    <!-- Set to false to avoid saving data when robot is not moving -->
    <param name="Mem/NotLinkedNodesKept" type="string" value="false"/>
  </node>
 </group>
<!-- visualization with rtabmapviz -->
  <node pkg="rtabmap_ros" type="rtabmapviz" name="rtabmapviz" args="-d $(find rtabmap_r
os)/launch/config/rgbd_gui.ini" output="screen">
      <param name="subscribe_depth"            type="bool" value="true"/>
      <param name="subscribe_scan"             type="bool" value="true"/>
      <param name="frame_id"                   type="string" value="base_footprint"/>
      <remap from="rgb/image"       to="$(arg rgb_topic)"/>
      <remap from="depth/image"     to="$(arg depth_topic)"/>
      <remap from="rgb/camera_info" to="$(arg camera_info_topic)"/>
      <remap from="scan"            to="/scan"/>
  </node>
</launch>
```

5. Launched the gazebo world and RViz:

```
roslaunch my_robot world.launch
```

6. Launched the teleop node:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

7. Launched our mapping node:

```
roslaunch my_robot mapping.launch
```

8. Launched rtabmap-databaseViewer:
   rtabmap-databaseViewer /home/robond/catkin_wsm/src/my_robot/rtabmap.db
9. Added some windows to get a better view of the relevant information:

- Said yes to using the database parameters
- View - Constraint View
- View - Graph View

10. Duplicated our `mapping.launch` file and renamed the duplicated file as `localization.launch`.
11. Made the following changes to the `localization.launch` file:
    - Removed the `args="--delete_db_on_start"` from our node launcher since the database was needed to localize too.
    - Removed the `Mem/NotLinkedNodesKept` parameter.
    - Added the `Mem/IncrementalMemory` parameter of type `string` and set it to `false` to finalize the changes needed to put the robot into localization mode.
12. Launched the gazebo world and RViz:

```
roslaunch my_robot world.launch
```

13. Launched the `teleop` node:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

14. Launched our localization node:

```
roslaunch my_robot localization.launch
```

3.1.4. PATH PLANNING (GRASS CUTTER)

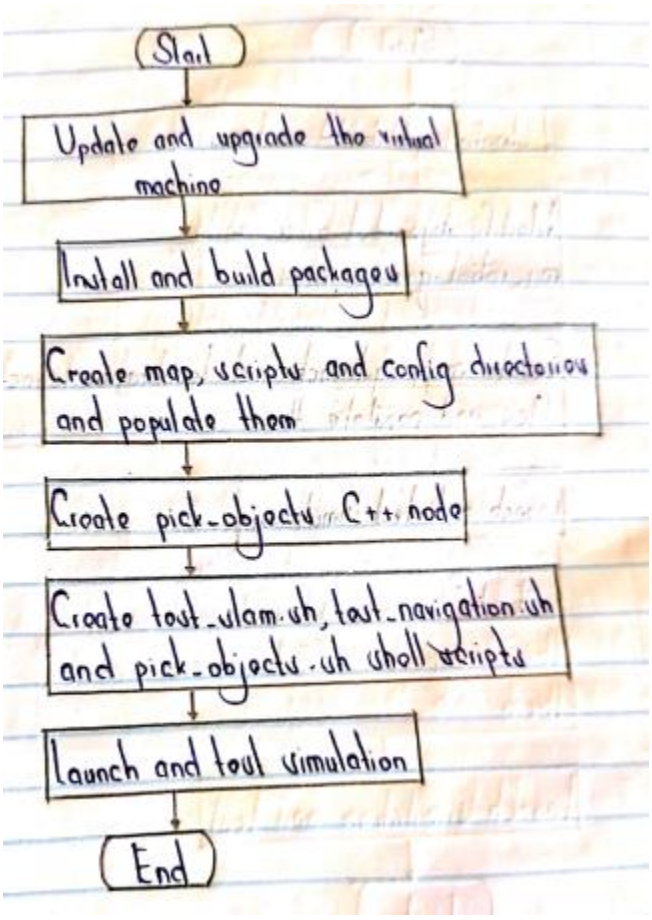


Figure 3.4 Path planning flowchart

1. Updated and upgraded the virtual machine:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install --fix-missing
```

2. Installed xterm:

   `sudo apt-get install xterm`

3. Created and initialized a catkin_ws:

```
$ mkdir catkin_wspp
$ cd catkin_wspp
```

```
$ mkdir src
$ cd src
$ catkin_init_workspace
```

4. Packages installed within the src folder using the following commands:
    - git clone https://github.com/ros-perception/slam_gmapping.git
    - git clone https://github.com/turtlebot/turtlebot.git
    - git clone https://github.com/turtlebot/turtlebot_interactions.git
    - git clone https://github.com/turtlebot/turtlebot_simulator.git
    - sudo apt install ros-kinetic-turtlebot*
    - sudo apt install ros-kinetic-turtlebot*-description
5. After installation the packages were ran using the following commands:

```
cd ..
catkin_make
```

6. Created the following directories using the commands shown below:
    - cd src
    - mkdir map
    - mkdir scripts
7. Placed the myworld.world file within the map directory of the current workspace.
8. Created test_slam.sh and added the following code:

```
#!/bin/sh
# launch turtlebot_world.launch to deploy a turtlebot within your environment
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
export ROBOT_INITIAL_POSE='-x -5 -y -2 -z 0 -R 0 -P 0 -Y 0';
roslaunch turtlebot_gazebo turtlebot_world.launch
world_file:=$(pwd)/../../src/map/myworld.world" &
sleep 20
# launch gmapping_demo.launch to perform SLAM
```

```
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
rosparam set /slam_gmapping/iterations 10;
rosparam set /slam_gmapping/linearUpdate 0.05;
rosparam set /slam_gmapping/angularUpdate 0.1;
rosparam set /slam_gmapping/map_update_interval 0.25;
rosparam set /slam_gmapping/srr 0.01;
rosparam set /slam_gmapping/srt 0.01;
rosparam set /slam_gmapping/str 0.02;
rosparam set /slam_gmapping/stt 0.02;
roslaunch turtlebot_gazebo gmapping_demo.launch" &
sleep 20
# launch view_navigation.launch to observe the map in rviz
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
roslaunch turtlebot_rviz_launchers view_navigation.launch" &
sleep 20
# launch keyboard_teleop.launch to manually control the robot
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
roslaunch turtlebot_teleop keyboard_teleop.launch"
```

9. Turned the script into an executable script using the following command:

   `chmod +x test_slam.sh`

10. Launched the shell script file:

    `./test_slam.sh`

11. Created `test_navigation.sh` shell script and added the following code:

```
#!/bin/sh
# launch turtlebot_world.launch to deploy a turtlebot within your environment
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
export ROBOT_INITIAL_POSE='-x -5 -y -2 -z 0 -R 0 -P 0 -Y 0';
```

```
roslaunch turtlebot_gazebo turtlebot_world.launch
world_file:=$(pwd)/../../src/map/myworld.world " &
sleep 20
# launch amcl_demo.launch for localization
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
roslaunch turtlebot_gazebo amcl_demo.launch
map_file:=$(pwd)/../../src/map/myworld.yaml " &
sleep 20
# launch view_navigation.launch to observe the map in rviz
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
roslaunch turtlebot_rviz_launchers view_navigation.launch"
```

12. Turned the script into an executable script using the following command:

```
chmod +x test_navigation.sh
```

13. Launched the shell script file:

```
./test_navigation.sh
```

14. Created a pick_objects package:

```
catkin_create_pkg pick_objects move_base_msgs actionlib roscpp
```

15. Created a pick_objects C++ node:

```
$ cd pick_objects
$ cd src
$ gedit pick_objects.cpp
```

16. Added the following code:

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
// Define a client to send goal requests to the move_base server through a
SimpleActionClient
```

```
typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
MoveBaseClient;
int main(int argc, char** argv){
 // Initialize the pick_objects node
 ros::init(argc, argv, "pick_objects");
 //tell the action client that we want to spin a thread by default
 MoveBaseClient ac("move_base", true);
 // Wait 5 sec for move_base action server to come up
 while(!ac.waitForServer(ros::Duration(5.0))){
  ROS_INFO("Waiting for the move_base action server to come up");
 }
 move_base_msgs::MoveBaseGoal goal;
 // set up the frame parameters
 goal.target_pose.header.frame_id = "map";
 goal.target_pose.header.stamp = ros::Time::now();
 ros::NodeHandle n; // node-handle for parameters
 // Define a position and orientation for the robot to reach
 n.getParam("/location_one/tx", goal.target_pose.pose.position.x);
 n.getParam("/location_one/ty", goal.target_pose.pose.position.y);
 n.getParam("/location_one/tz", goal.target_pose.pose.position.z);
 n.getParam("/location_one/qx", goal.target_pose.pose.orientation.x);
 n.getParam("/location_one/qy", goal.target_pose.pose.orientation.y);
 n.getParam("/location_one/qz", goal.target_pose.pose.orientation.z);
 n.getParam("/location_one/qw", goal.target_pose.pose.orientation.w);
 // Send the goal position and orientation
 ROS_INFO("Sending goal");
 ac.sendGoal(goal);
 // Wait an infinite time for the results
 ac.waitForResult();
 // Check if the robot reached its goal
 if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
```

```
    ROS_INFO("Robot reached location");
    ros::Duration(5.0).sleep();
   }
   else {
    ROS_INFO("The robot failed to reach location");
   }
 // Define a position and orientation for the robot to reach
   n.getParam("/location_two/tx", goal.target_pose.pose.position.x);
   n.getParam("/location_two/ty", goal.target_pose.pose.position.y);
   n.getParam("/location_two/tz", goal.target_pose.pose.position.z);
   n.getParam("/location_two/qx", goal.target_pose.pose.orientation.x);
   n.getParam("/location_two/qy", goal.target_pose.pose.orientation.y);
   n.getParam("/location_two/qz", goal.target_pose.pose.orientation.z);
   n.getParam("/location_two/qw", goal.target_pose.pose.orientation.w);
   // Send the goal position and orientation
   ROS_INFO("Sending goal");
   ac.sendGoal(goal);
   // Wait an infinite time for the results
   ac.waitForResult();
   // Check if the robot reached its goal
   if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
    ROS_INFO("Robot reached location");
    ros::Duration(5.0).sleep();
   }
   else {
    ROS_INFO("The robot failed to reach location");
   }
 // Define a position and orientation for the robot to reach
   n.getParam("/location_three/tx", goal.target_pose.pose.position.x);
   n.getParam("/location_three/ty", goal.target_pose.pose.position.y);
   n.getParam("/location_three/tz", goal.target_pose.pose.position.z);
```

```
  n.getParam("/location_three/qx", goal.target_pose.pose.orientation.x);
  n.getParam("/location_three/qy", goal.target_pose.pose.orientation.y);
  n.getParam("/location_three/qz", goal.target_pose.pose.orientation.z);
  n.getParam("/location_three/qw", goal.target_pose.pose.orientation.w);
  // Send the goal position and orientation
  ROS_INFO("Sending goal");
  ac.sendGoal(goal);
  // Wait an infinite time for the results
  ac.waitForResult();
  // Check if the robot reached its goal
  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
    ROS_INFO("Robot reached location");
    ros::Duration(5.0).sleep();
  }
  else {
    ROS_INFO("The robot failed to reach location");
  }
 // Define a position and orientation for the robot to reach
  n.getParam("/location_four/tx", goal.target_pose.pose.position.x);
  n.getParam("/location_four/ty", goal.target_pose.pose.position.y);
  n.getParam("/location_four/tz", goal.target_pose.pose.position.z);
  n.getParam("/location_four/qx", goal.target_pose.pose.orientation.x);
  n.getParam("/location_four/qy", goal.target_pose.pose.orientation.y);
  n.getParam("/location_four/qz", goal.target_pose.pose.orientation.z);
  n.getParam("/location_four/qw", goal.target_pose.pose.orientation.w);
  // Send the goal position and orientation
  ROS_INFO("Sending goal");
  ac.sendGoal(goal);
  // Wait an infinite time for the results
  ac.waitForResult();
  // Check if the robot reached its goal
```

```
  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
    ROS_INFO("Robot reached location");
    ros::Duration(5.0).sleep();
  }
  else {
    ROS_INFO("The robot failed to reach location");
  }
// Define a position and orientation for the robot to reach
  n.getParam("/location_five/tx", goal.target_pose.pose.position.x);
  n.getParam("/location_five/ty", goal.target_pose.pose.position.y);
  n.getParam("/location_five/tz", goal.target_pose.pose.position.z);
  n.getParam("/location_five/qx", goal.target_pose.pose.orientation.x);
  n.getParam("/location_five/qy", goal.target_pose.pose.orientation.y);
  n.getParam("/location_five/qz", goal.target_pose.pose.orientation.z);
  n.getParam("/location_five/qw", goal.target_pose.pose.orientation.w);
  // Send the goal position and orientation
  ROS_INFO("Sending goal");
  ac.sendGoal(goal);
  // Wait an infinite time for the results
  ac.waitForResult();
  // Check if the robot reached its goal
  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
    ROS_INFO("Robot reached location");
    ros::Duration(5.0).sleep();
  }
  else {
    ROS_INFO("The robot failed to reach location");
  }
  // Define a position and orientation for the robot to reach
  n.getParam("/location_six/tx", goal.target_pose.pose.position.x);
  n.getParam("/location_six/ty", goal.target_pose.pose.position.y);
```

```
  n.getParam("/location_six/tz", goal.target_pose.pose.position.z);
  n.getParam("/location_six/qx", goal.target_pose.pose.orientation.x);
  n.getParam("/location_six/qy", goal.target_pose.pose.orientation.y);
  n.getParam("/location_six/qz", goal.target_pose.pose.orientation.z);
  n.getParam("/location_six/qw", goal.target_pose.pose.orientation.w);
  // Send the goal position and orientation
  ROS_INFO("Sending goal");
  ac.sendGoal(goal);
  // Wait an infinite time for the results
  ac.waitForResult();
  // Check if the robot reached its goal
  if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {
    ROS_INFO("Robot reached location");
  }
  else {
    ROS_INFO("The robot failed to reach location");
  }
  return 0;
}
```

17. Edited the CMakeLists.txt file. Added directories, executable and target link libraries.
18. Built the catkin_wspp.
19. Created a pick_objects.sh script file and added the following code:

```
#!/bin/sh
# launch turtlebot_world.launch to deploy a turtlebot within your environment
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
export ROBOT_INITIAL_POSE='-x -5 -y -2 -z 0 -R 0 -P 0 -Y 0';
roslaunch turtlebot_gazebo turtlebot_world.launch
world_file:=$(pwd)/../../src/map/myworld.world " &
sleep 20
# launch amcl_demo.launch for localization
```

```
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
roslaunch turtlebot_gazebo amcl_demo.launch
map_file:=$(pwd)/../../src/map/myworld.yaml " &
sleep 20
# launch view_navigation.launch to launch RViz
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
roslaunch turtlebot_rviz_launchers view_navigation.launch" &
sleep 30 # keeping large to enable visualization
# launch pick_objects node
xterm -e "cd $(pwd)/../..;
source devel/setup.bash;
rosparam load $(pwd)/../config/position_config.yaml;
rosrun pick_objects pick_objects"
```

20. Turned the script into an executable script using the following command:

```
chmod +x pick_objects.sh
```

21. Launched the shell script file:

```
./pick_objects.sh
```

3.2. IMPLEMENTATION

This chapter discusses the overall system architecture of the solar-powered autonomous grass cutter and the techniques used for mapping, localization, motion-planning and obstacle avoidance. The system's flowchart is also provided.

3.2.1. Overall system architecture:

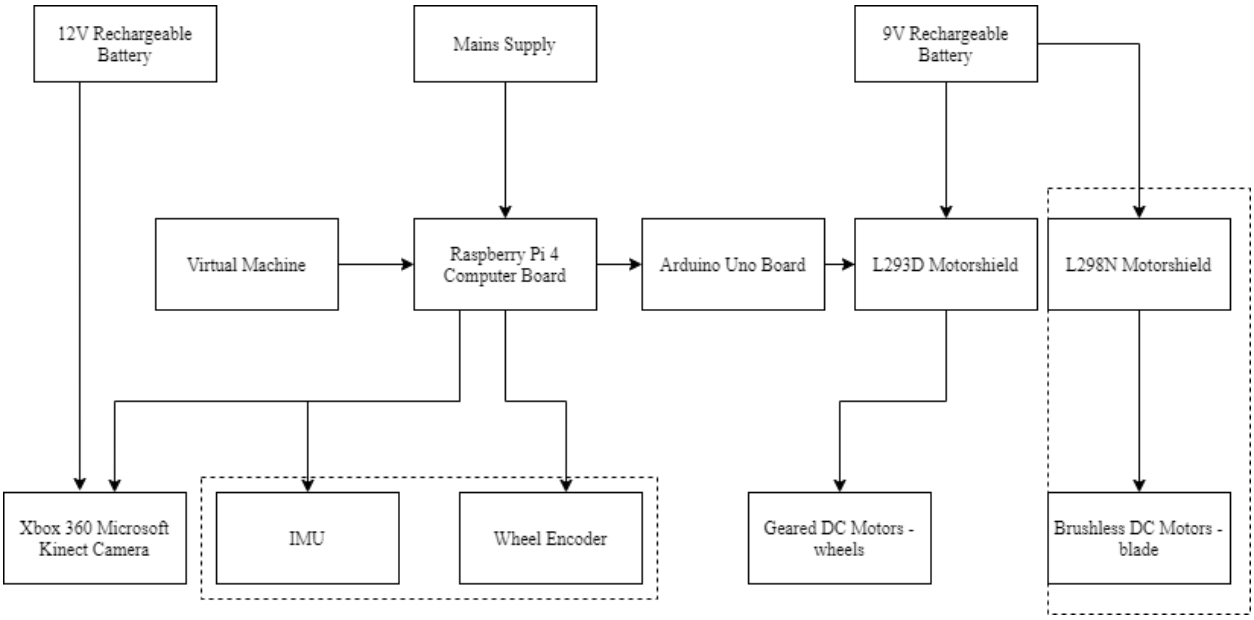


Figure 3.5 Overall system architecture

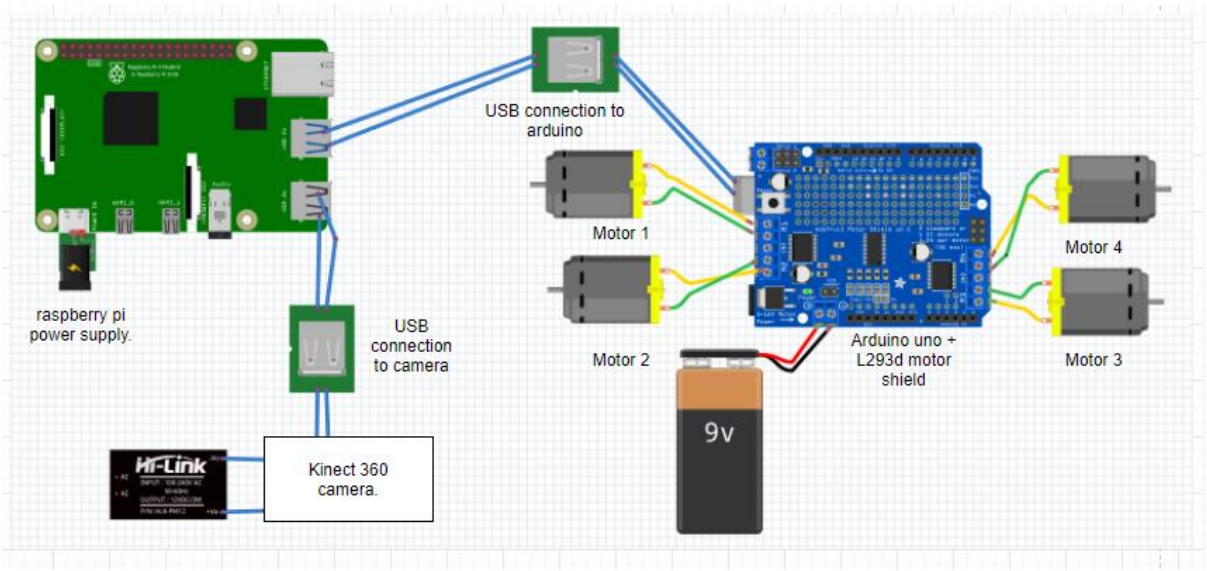


Figure 3.6 Circuit diagram

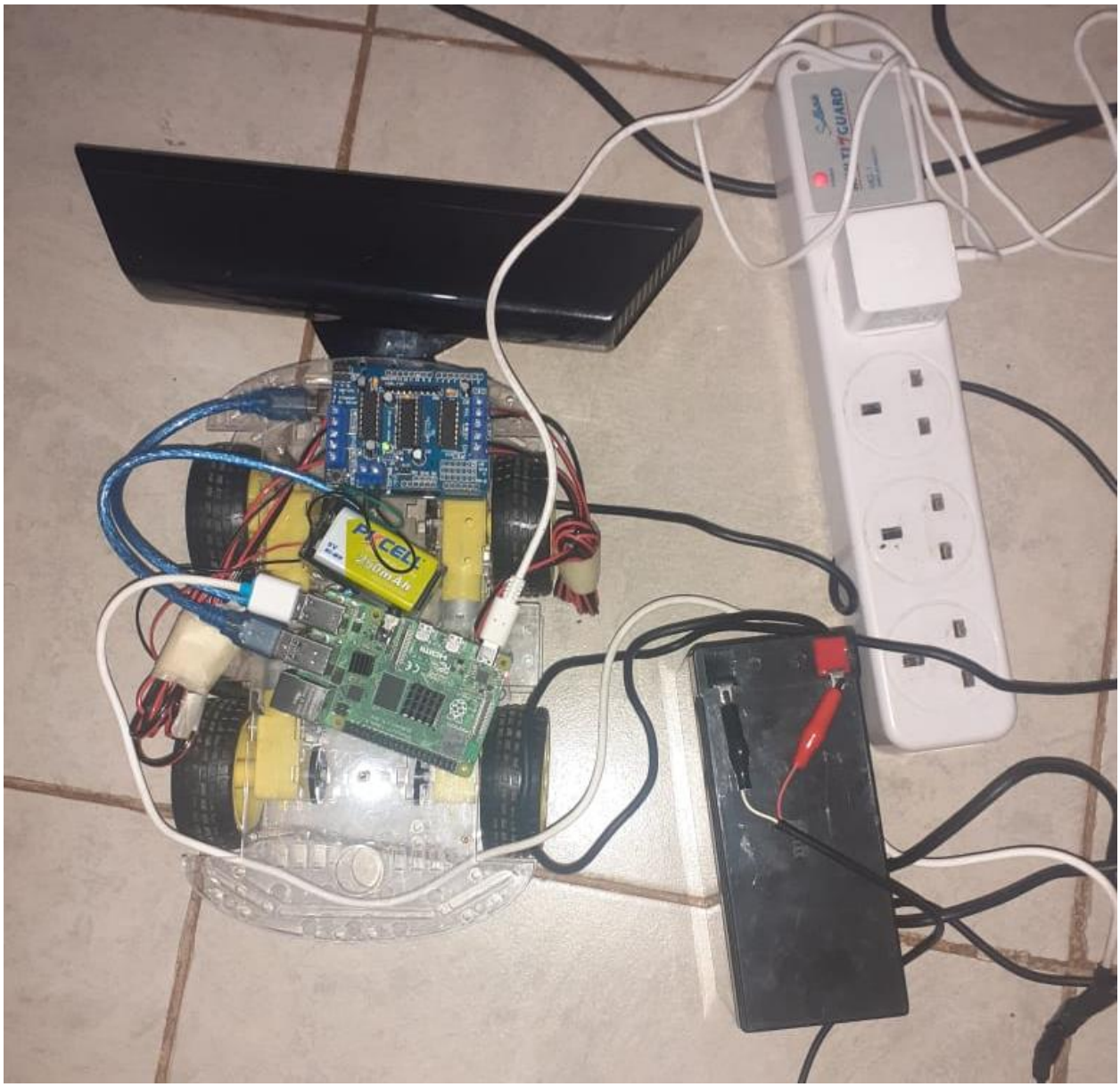


Figure 3.7 Circuit diagram

3.2.2. System Power:

The Raspberry Pi microcontroller was powered directly from the mains supply.

The motor driver mounted on Arduino was powered by a 9V rechargeable battery.

Finally, the Xbox 360 Kinect camera sensor was powered by a rechargeable 12V battery.

3.2.3. Raspberry Pi

The Raspberry Pi 4 Computer Model B 4GB RAM was used for this project. This is because of its superior memory capacity and processing speed.

3.2.4. Arduino

The Arduino Uno board was used to run the motors due to its compatibility with the L293D Motor shield.

3.2.5. Linux Virtual Machine

On a laptop, a ROS enabled Linux Virtual Machine to help us build the software for the robot. The steps for setting up the Virtual Machine were as follows:

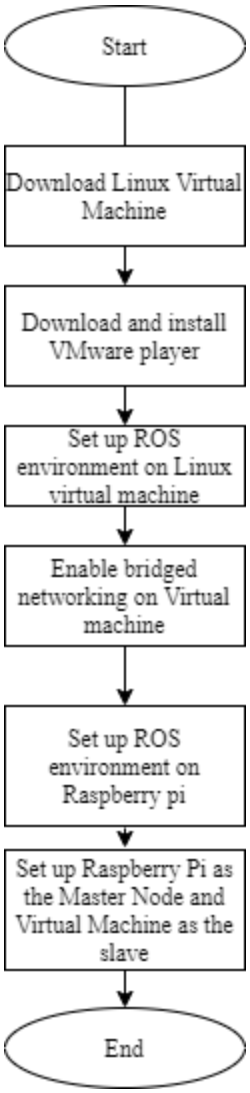


Figure 3.8 Setting up ROS environment

1. Downloading Linux Image:
   i) Download the compressed VM disk image from the link: https://s3-us-west-1.amazonaws.com/udacity-robotics/Virtual+Machines/Lubuntu_071917/RoboVM_V2.1.0.zip
   ii) Extract the compressed image on our Windows laptop.
2. Download and install VMware Player:
   i) Download and install VMware Player from: http://www.vmware.com/products/player/playerpro-evaluation.html
   ii) Open VMware
   iii) Press "Open a Virtual Machine"

iv) Navigate to the folder that contains the VM files

v) Select the larger .ova file

vi) Import the files

3. Setting up the ROS environment on our Linux Virtual Machine using the following commands:

i) Setup sources.list:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

ii) Set up keys:

```
sudo apt install curl
```

```
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add –
```

iii) Installation:

```
sudo apt update
```

```
sudo apt-get install ros-kinetic-desktop-full
```

iv) Environment setup:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
```

```
source ~/.bashrc
```

v) Dependencies for building packages:

```
sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool build-essential
```

vi) Initializing rosdep:

```
sudo apt install python-rosdep
```

```
sudo rosdep init
```

```
rosdep update
```

vii) Next we upgrade the cmake version on our ROS environment using the following commands:

```
apt install build-essential git

git clone https://github.com/Kitware/CMake/; cd CMake

./bootstrap && make && sudo make install
```

vii) Next, we setup our catkin workspace using the following commands:

```
mkdir -p /home/ubuntu/catkin_ws/src

cd /home/workspace/catkin_ws/src

catkin_init_workspace

catkin_make
```

4. To enable the ROS enabled virtual machine and the ROS enabled Raspberry pi to communicate:
   i) On Virtual Machine setting we added a Bridged Network adapter.
   ii) Executed the following command the find the VM IPV4 address:

```
ifconfig

sudo nano .bashrc
```

   iii) Added the following lines of code:

```
export ROS_IP= “VM IPV4 address”

export ROS_MASTER_URI=http://”VM IPV4 address or name”:11311
```

3.2.6. Raspberry pi

We used a Raspberry Pi 4 Computer Model B 4GB RAM.

Software Setup:

1. First, we installed a Raspberry Pi Image that came with pre-installed ROS from: https://ubiquity-pi-image.sfo2.cdn.digitaloceanspaces.com/2020-11-07-ubiquity-xenial-lxde-raspberry-pi.img.xz
2. Next, we used etcher software to flash the image onto the Micro SD.
3. Next, we connect to the Raspberry Pi network. The Raspberry Pi on the initial boot up comes up as an access point with an SSID of ubiquityrobotXXXX where XXXX is part of the MAC address. The wifi password is robotseverywhere.
4. Next, on the Linux Virtual Machine terminal we connect to the Raspberry Pi using the command (10.42.0.1 is the standard IP address of ubiquityrobots):

   ssh ubuntu@ubiquityrobot or ssh ubuntu@10.42.0.1

   password=Ubuntu

5. Next we used the following pifi commands to change the robot's name and to add a network name it cannot connect to upon reboot, using the following commands:

   sudo pifi set-hostname grasscutter

   sudo pifi add "Network_Name" "Network_Password"

   sudo reboot

6. Next, we set up our ROS environment using the following commands:

   i) Setup sources.list:

   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

   ii) Set up keys:

   sudo apt install curl

   curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add –

iii) Installation:

```
sudo apt update

sudo apt-get install ros-kinetic-desktop-full
```

iv) Environment setup:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

v) Dependencies for building packages:

```
sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool build-essential
```

vi) Initializing rosdep:

```
sudo apt install python-rosdep

sudo rosdep init

rosdep update
```

vii) Next we upgrade the cmake version on our ROS environment using the following commands:

```
apt install build-essential git

git clone https://github.com/Kitware/CMake/; cd CMake

./bootstrap && make && sudo make install
```

viii) Next, we setup our catkin workspace using the following commands:

```
mkdir -p /home/ubuntu/catkin_ws/src

cd /home/workspace/catkin_ws/src

catkin_init_workspace
```

```
catkin_make
```

7. To enable the ROS enabled Raspberry Pi and the ROS enabled VM to communicate:
   i) Executed the following command the find the Raspberry Pi IPV4 address:

```
ifconfig

sudo nano .bashrc
```

   ii) Added the following lines of code:

```
export ROS_IP= “Raspberry pi IPV4 address”

export ROS_MASTER_URI=http://”Raspberry Pi IPV4 address or name”:11311
```

3.2.7. Arduino and Differential Drive

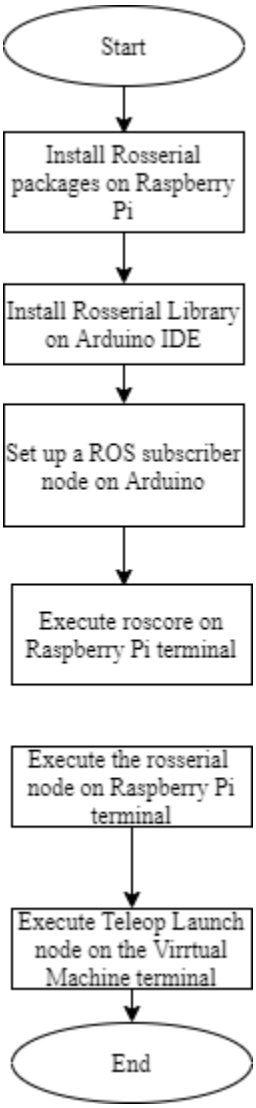


Figure 3.9 Setting up serial communication with the Arduino

Differential drive:

Robots can use a drive mechanism known as differential drive. It consists of 2 drive wheels mounted on a common axis, and each wheel can independently be driven either forward or backward. There are two types of turning:

1. Single-wheel turning: involves leaving one wheel stationary and turning the other.

2. Double-wheel turning: involves rotating both wheels in opposite directions. This is preferred as it distributes the work of turning on both wheels.

ROS uses geometry messages called geometry_msgs/Twist.msg to control the motors. This expresses velocity in free space broken into its linear and angular parts.

The following three steps are taken to come up with double wheel turn differential equations:

i. Calculate wheel speeds from Twist Message

The robot moves in a specific direction (forward or backward) at a specific speed (in m/s). The linear velocity (x) from the twist message relates to the wheel linear velocity (v) as follows:

$$v = x \tag{3.1}$$

The circumference(c) of the circle:

$$c = 2\pi r \tag{3.2}$$

The ratio of the angle the robot needs to turn, in combination with circumference and angular velocity (z) to obtain the angular wheel angular speed (v) shown below:

$$v = 2\pi r \left(\frac{z}{360}\right) \tag{3.3}$$

The radius, r, above should be half the distance between the two wheels called the track (t). Therefore, the above equation becomes:

$$v = (2\pi)\left(\frac{t}{2}\right)\left(\frac{z}{360}\right) \tag{3.4}$$

The twist message specifies angular velocity in radians per second. The equation above reduces to:

$$v = \frac{zt}{2} \tag{3.5}$$

ROS conventions follow the right-hand rule, which dictate that a positive angular velocity means a counter-clockwise turn, and a negative angular velocity means a clockwise turn. Using these, we obtain both wheel speeds with the formulas shown below:

$$lv = x - \frac{zt}{2} \qquad 3.6$$
$$rv = x + \frac{zt}{2}$$

$lv \Rightarrow$ left wheel velocity (meters per second)
$rv \Rightarrow$ right wheel velocity (meters per second)
$t \Rightarrow$ track (meters)
$x \Rightarrow$ linear velocity commanded (meters per second)
$z \Rightarrow$ angular velocity commanded (radians per second)

ii. Convert wheel speeds into duty cycles

Robot speed is controlled using duty cycles between 0 to 100. The motor's maximum speed is assigned a duty cycle of 100.

The equations in step 1 above give us wheel velocities in meters per second. Therefore, the equation shown below converts the speed into duty cycle using 0.5m/s as the maximum speed.

$$d = \frac{100\text{v}}{0.5} \qquad 3.7$$

$d \Rightarrow$ duty cycle (percentage, $0 - 100$)
$v \Rightarrow$ wheel velocity (meters per second)

iii. Convert duty cycle into PWM signals

The L293D motor controller uses pulse width signals between 0 and 255 to control motor speeds. The equation below is used to convert duty cycle into a pulse width value.

$$p = \frac{255\text{d}}{100} \qquad 3.8$$

$d \Rightarrow$ duty cycle (percentage, $0 - 100$)
$p \Rightarrow$ pulse width modulation (0-255)

If the pulse width value is less than 0, the motors turn backwards. Otherwise, they turn forwards.

Software Setup:

1. We started by installing the following packages on our Raspberry Pi ROS to enable serial communication using the following commands:

```
sudo apt-get install ros-kinetic-rosserial-arduino

sudo apt-get install ros-kinetic-rosserial

sudo apt-get install ros-kinetic-rosserial-python

git clone https://github.com/ros-drivers/rosserial.git

cd catkin_ws

catkin_make

catkin_make install
```

2. Next, we setup a ROS node on the Arduino using the following code:

```
#include <AFMotor.h>

#include <ros.h>

#include <geometry_msgs/Twist.h>

// Pin variables for motors.

AF_DCMotor motor1(1);

AF_DCMotor left_motor(1, MOTOR12_64KHZ);

AF_DCMotor motor2(2);

AF_DCMotor motor3(3);

AF_DCMotor right_motor(3, MOTOR12_64KHZ);
```

```
AF_DCMotor motor4(4);
double w_r = 0, w_l = 0;
//wheel_rad is the wheel radius , wheel_sep is
double wheel_rad = 0.0325, wheel_sep = 0.295;
//setting up a ROS node
ros::NodeHandle nh;
int lowSpeed = 200;
int highSpeed = 50;
double speed_ang = 0, speed_lin = 0;
//Defining the message type that the node will subscribe to
void messageCb( const geometry_msgs::Twist& msg) {
  speed_ang = msg.angular.z;
  speed_lin = msg.linear.x;
  w_r = (speed_lin / wheel_rad) + ((speed_ang * wheel_sep) / (2.0 * wheel_rad));
  w_l = (speed_lin / wheel_rad) - ((speed_ang * wheel_sep) / (2.0 * wheel_rad));
}
//Setting up an instance of a ROS node, a subscriber node
ros::Subscriber<geometry_msgs::Twist> sub("cmd_vel", &messageCb );
void Motors_init();
void MotorL(int Pulse_Width1);
void MotorR(int Pulse_Width2);
```

```
void setup() {
  Motors_init();
  nh.initNode();
  nh.subscribe(sub);
}
void loop() {
  MotorL(w_l * 10);
  MotorR(w_r * 10);
  nh.spinOnce();
}
void Motors_init() {
  motor1.setSpeed(200);
  motor1.run(RELEASE);
  motor2.setSpeed(200);
  motor2.run(RELEASE);
  motor3.setSpeed(200);
  motor3.run(RELEASE);
  motor4.setSpeed(200);
  motor4.run(RELEASE);
}
void MotorL(int Pulse_Width1) {
```

```
if (Pulse_Width1 > 0) {
  //analogWrite(EN_L, Pulse_Width1);
  //digitalWrite(IN1_L, HIGH);
  //digitalWrite(IN2_L, LOW);
  motor1.run(FORWARD);
  motor2.run(FORWARD);
  motor3.run(FORWARD);
  motor4.run(FORWARD);
}
if (Pulse_Width1 < 0) {
  Pulse_Width1 = abs(Pulse_Width1);
  motor1.run(BACKWARD);
  motor2.run(BACKWARD);
  motor3.run(BACKWARD);
  motor4.run(BACKWARD);
}
if (Pulse_Width1 == 0) {
  //analogWrite(EN_L, Pulse_Width1);
  //digitalWrite(IN1_L, LOW);
  //digitalWrite(IN2_L, LOW);
  motor1.run(RELEASE);
```

```
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
  }
}
void MotorR(int Pulse_Width2) {
  if (Pulse_Width2 > 0) {
    //analogWrite(EN_R, Pulse_Width2);
    //digitalWrite(IN1_R, LOW);
    //digitalWrite(IN2_R, HIGH);
    motor1.run(FORWARD);
    motor2.run(FORWARD);
    motor3.run(FORWARD);
    motor4.run(FORWARD);
  }
  if (Pulse_Width2 < 0) {
    Pulse_Width2 = abs(Pulse_Width2);
    motor1.run(BACKWARD);
    motor2.run(BACKWARD);
    motor3.run(BACKWARD);
    motor4.run(BACKWARD);
```

```
  }
  if (Pulse_Width2 == 0) {
    motor1.run(RELEASE);
    motor2.run(RELEASE);
    motor3.run(RELEASE);
    motor4.run(RELEASE);
  }
}
```

3. Next, to start the serial communication on the Raspberry Pi using the following command:

i) On one terminal:

```
roscore
```

ii) On another terminal:

```
rosrun rosserial_python serial_node.py /dev/ttyACM0
```

iii) On a third terminal, to control the robot:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

3.2.6. Sensors:

The following sensors were used on the robotic grass cutter: an Xbox 360 Kinect camera sensor.

Xbox 360 Kinect Camera

Kinect is a line of depth and motion sensing input devices produced by Microsoft. The devices generally contain RGB cameras and infrared projectors and detectors that map depth through either structured light or time of flight.

The Kinect contains three vital pieces that work together to detect your motion and create your physical image on the screen: an RGB color VGA video camera, a depth sensor, and a multi-array microphone.

The Xbox 360 contains a monochrome CMOS sensor uses structured light for depth sensing using a near-infrared pattern projected across the space in front of the Kinect, while an infrared sensor captures the reflected light pattern. The light pattern is deformed by the relative depth of the objects in front of it, and the mathematics can be used to estimate the depth.

Due to its affordability, this camera became the perfect imaging tool for our robot.

It was utilized for mapping, localization and motion planning.

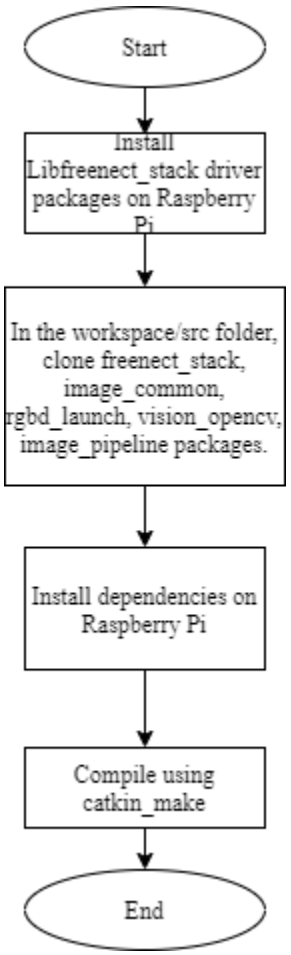


Figure 3.10 Setting up the Xbox 360 Microsoft Kinect camera

Software Setup

1. On the raspberry pi we installed the following ROS drivers:

libfreenect drivers - freenect_stack package for ROS

2. The procedure involved running the following commands on the terminal of our raspberry pi:

```
sudo apt-get update

sudo apt-get install cmake build-essential libusb-1.0-0-dev

git clone https://github.com/OpenKinect/libfreenect.git

cd libfreenect

mkdir build && cd build

cmake -L ..

make

sudo make install
```

3. Next, we moved to our catkin_ws, src folder and installed the following packages:

```
git clone https://github.com/ros-drivers/freenect_stack.git

git clone https://github.com/ros-perception/image_common.git

git clone https://github.com/ros-drivers/rgbd_launch.git

git clone https://github.com/ros-perception/vision_opencv.git

git clone https://github.com/ros-perception/image_pipeline.git
```

4. Next, move back to the catkin_ws folder and install system dependencies for the packages installed using the following commands:

```
rosdep install --from-paths src --ignore-src

sudo apt-get install libbullet-dev libharfbuzz-dev libgtk2.0-dev libgtk-3-dev
```

5. Next, compile the system using the command:

```
catkin_make -j2
```

3.2.7. Techniques:

3.2.7.1. RTABMAP SLAM Algorithm:

RTAB-Map (Real-Time Appearance-Based Mapping) is a RGB-D, Stereo and Lidar Graph-Based SLAM approach based on an incremental appearance-based loop closure detector. The loop closure detector uses a bag-of-words approach to determinate how likely a new image comes from a previous location or a new location. When a loop closure hypothesis is accepted, a new constraint is added to the map's graph, then a graph optimizer minimizes the errors in the map. A memory management approach is used to limit the number of locations used for loop closure detection and graph optimization, so that real-time constraints on large-scale environnements are always respected. RTAB-Map can be used alone with a handheld Kinect, a stereo camera or a 3D lidar for 6DoF mapping, or on a robot equipped with a laser rangefinder for 3DoF mapping. We settled on this SLAM algorithm because of its memory management ability and ease of use and affordability, since we could utilize an Xbox 360 Microsoft Kinect camera.

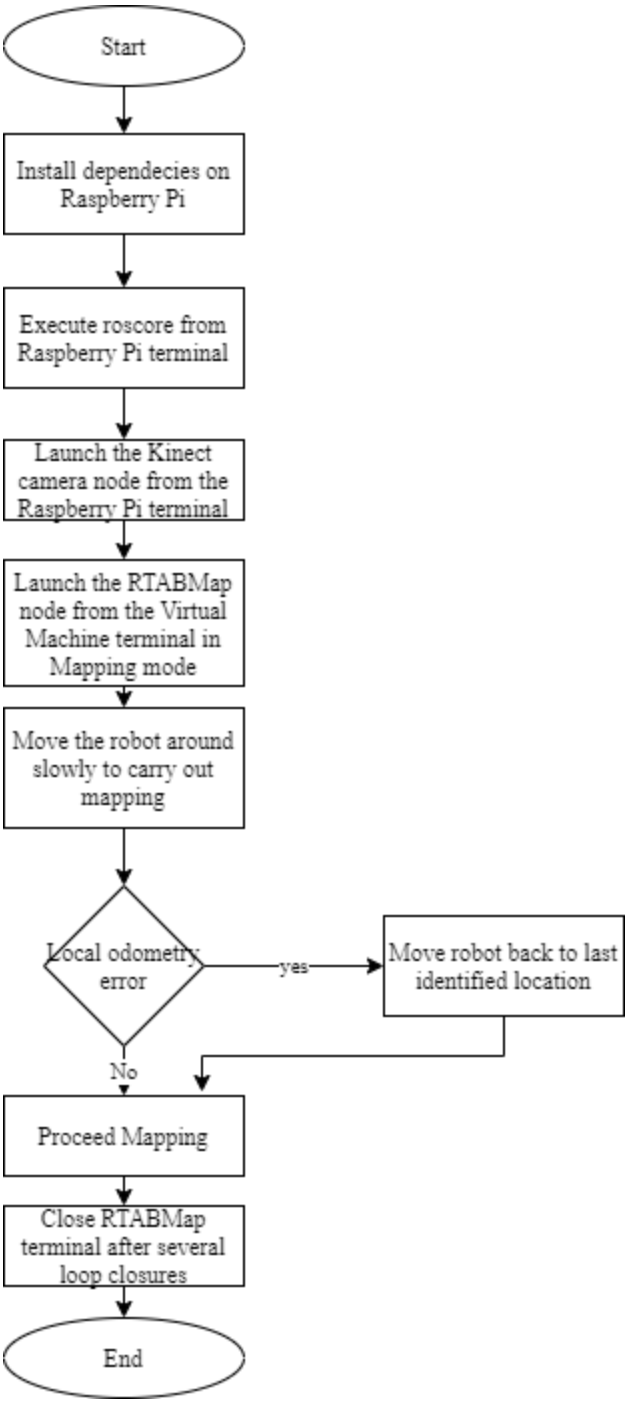


Figure 3.11 RTAB-Mapping algorithm

1. To setup RTABMAP SLAM on our ROS enabled Raspberry Pi we executed the following commands on our VM:

   i) Install dependencies:

```
sudo apt install ros-kinetic-rtabmap ros-kinetic-rtabmap-ros
```

ii) Install standalone libraries:

```
cd ~

git clone https://github.com/introlab/rtabmap.git rtabmap

cd rtabmap/build

cmake ..  [<---double dots included]

make

sudo make install
```

iii) Install RTAB-Map ros-pkg in our src folder of our catkin_ws:

```
cd ~/catkin_ws

git clone https://github.com/introlab/rtabmap_ros.git src/rtabmap_ros

catkin_make -j4
```

2. Next we set up a launch file to launch the camera using the following command:

i) Make Launch file:

```
gedit /home/robond/catkin_ws/src/rtabmap_ros/launch/freenect_throttle.launch
```

ii) Added the following code:

```
<launch>
  <include file="$(find freenect_launch)/launch/freenect.launch">
    <arg name="depth_registration" value="True" />
  </include>
  <arg name="rate"  default="5"/>
  <arg name="approx_sync" default="true" /> <!-- true for freenect driver -
->
```

```
  <arg name="rgbd_sync" default="true"/>

  <!-- Use same nodelet used by Freenect/OpenNI -->

  <group ns="camera">

    <node if="$(arg rgbd_sync)" pkg="nodelet" type="nodelet" name="rgbd_sync" args="load rtabmap_ros/rgbd_sync camera_nodelet_manager" output="screen">

      <param name="compressed_rate"  type="double" value="$(arg rate)"/>

      <param name="approx_sync" type="bool" value="$(arg approx_sync)"/>

      <remap from="rgb/image"       to="rgb/image_rect_color"/>

      <remap from="depth/image"     to="depth_registered/image_raw"/>

      <remap from="rgb/camera_info" to="rgb/camera_info"/>

      <remap from="rgbd_image"      to="rgbd_image"/>

    </node>

    <node unless="$(arg rgbd_sync)" pkg="nodelet" type="nodelet" name="data_throttle" args="load rtabmap_ros/data_throttle camera_nodelet_manager" output="screen">

      <param name="rate"        type="double" value="$(arg rate)"/>

      <param name="approx_sync" type="bool" value="$(arg approx_sync)"/>

      <remap from="rgb/image_in"       to="rgb/image_rect_color"/>

      <remap from="depth/image_in"     to="depth_registered/image_raw"/>

      <remap from="rgb/camera_info_in" to="rgb/camera_info"/>
```

```
    <remap from="rgb/image_out" to="throttled/rgb/image_rect_color"/>
    <remap from="depth/image_out" to="throttled/depth_registered/image_raw"/>
    <remap from="rgb/camera_info_out" to="throttled/rgb/camera_info"/>
  </node>
 </group>
</launch>
```

3. Next, on our master, the ROS enabled Raspberry Pi, we execute:

```
roscore
```

4. Next, we launched the camera with the following command:

```
roslaunch rtabmap_ros freenect_throttle.launch rate:=5
```

5. On our ROS enabled VM we execute:

```
roslaunch rtabmap_ros rgbd_mapping.launch subscribe_rgbd:=true rgbd_topic:=/camera/rgbd_image compressed:=true rtabmap_args:="--delete_db_on_start"
```

6. We the carried out mapping and generated a 3-D point cloud map and a 2-D occupancy grid map as shown below:

```
rtabmap-databaseViewer ~/.ros/rtabmap.db
```

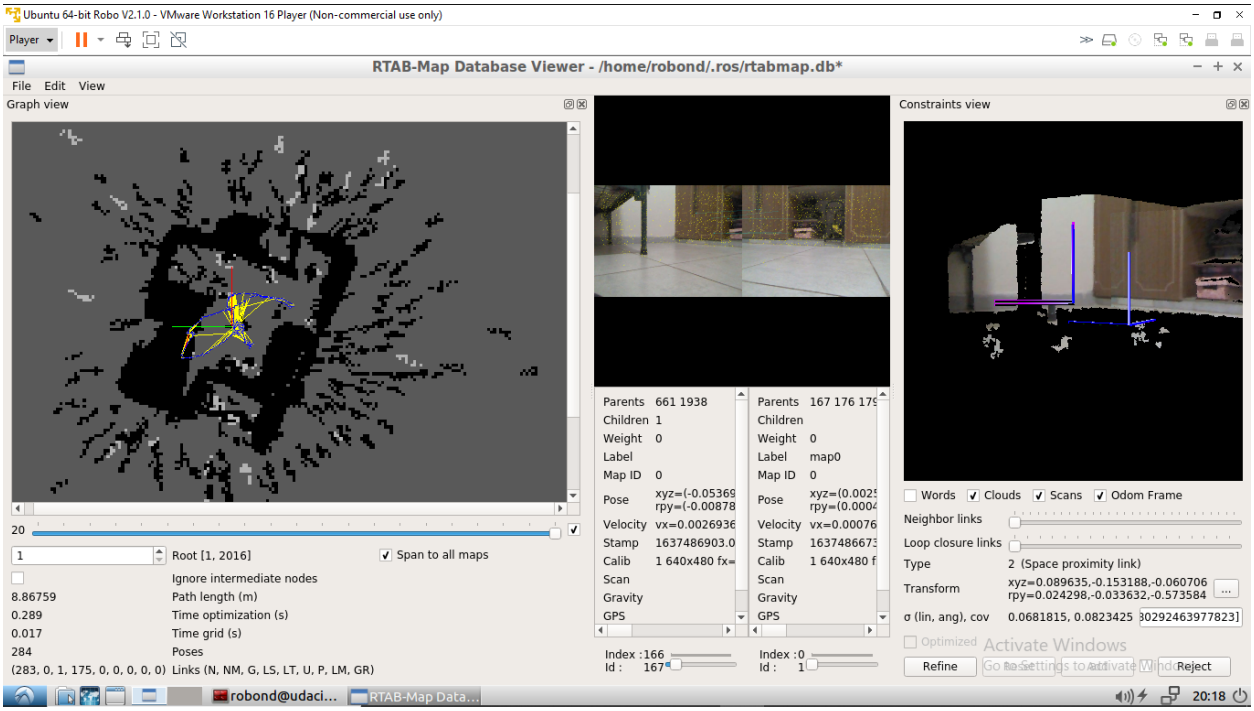


Figure 3.12 rtabmap.db

3.2.7.2. Localization:

We used the RTAB-Map localization feature to localize the robot in our mapped environment, using the following commands:

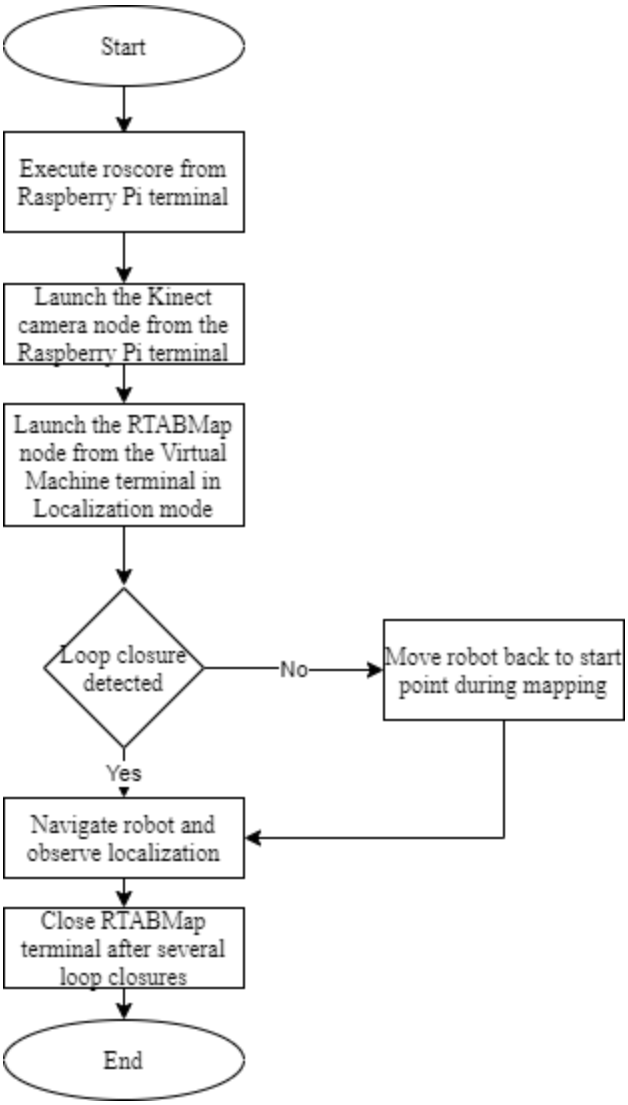


Figure 3.13 RTABMAP localization

1. On one terminal of our ROS enabled Raspberry Pi:

   roscore

2. On another terminal of our ROS enabled Raspberry Pi:

   roslaunch rtabmap_ros freenect_throttle.launch rate:=5

3. On another terminal of our ROS enabled VM:

   roslaunch rtabmap_ros rgbd_mapping.launch subscribe_rgbd:=true rgbd_topic:=/camera/rgbd_image compressed:=true localization:=true

CHAPTER FOUR:

4. RESULTS:

4.1. SIMULATION DISCUSSION

4.1.1. MAPPING

Mapping performed using two ROS packages.

The first ROS package used was the RTAB_Map ROS package. RTAB_Map ROS package uses graph SLAM algorithm to perform mapping. Graph SLAM uses nodes and constraints to generate a map. The robot's pose and features within the environment are treated as nodes. Motion constraints are used to tie poses together while measurement constraints are used to tie poses and features together. Loop closers occur when a sensor reading taken matches a previous sensor reading. Constraints are then adjusted generating a map of best fit.

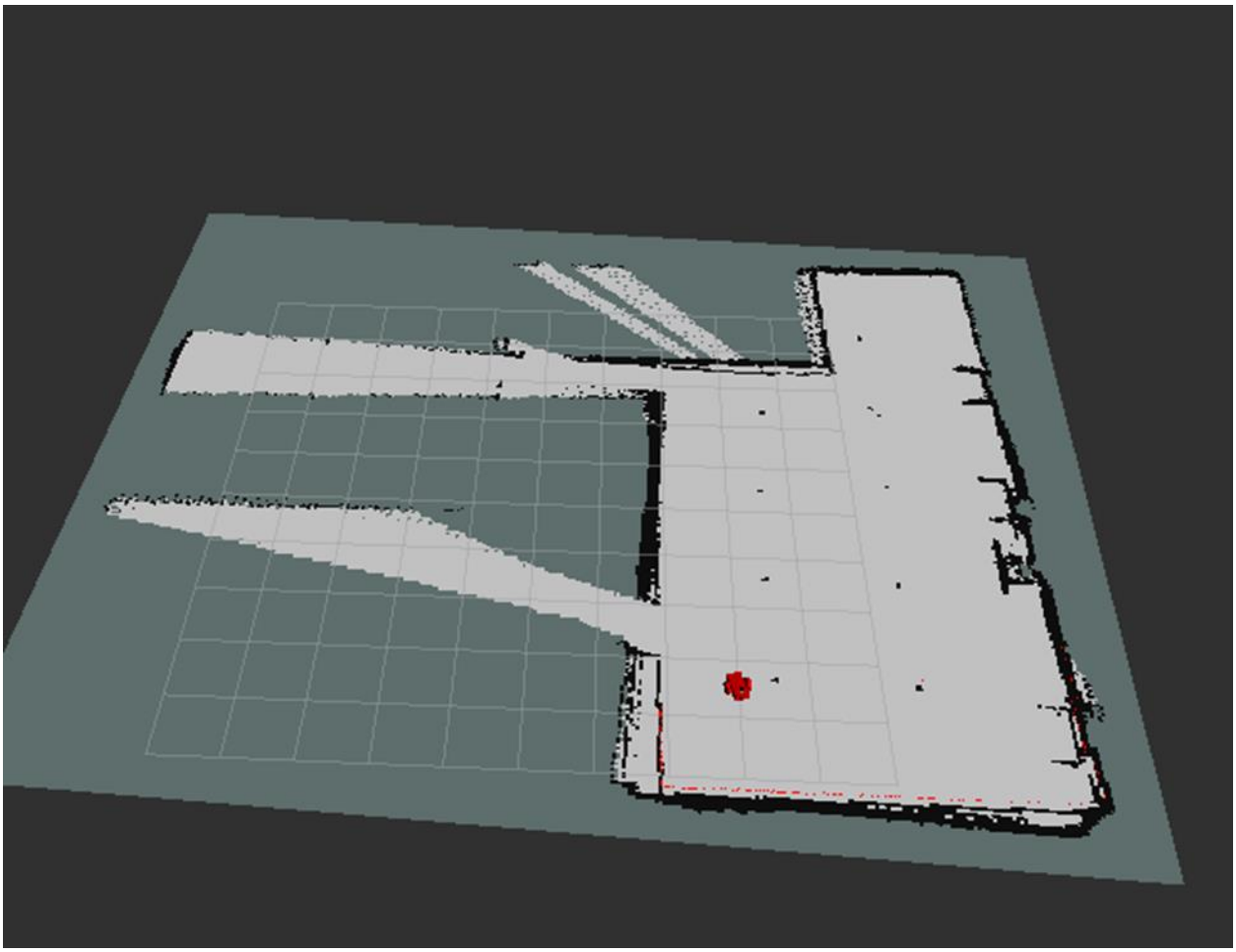


Figure 4.1 2D map generated using RTAB_map ros package

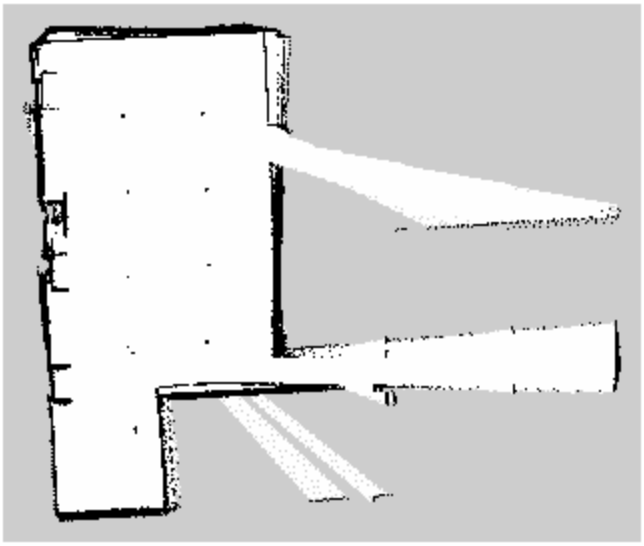


Figure 4.2 db image generated using RTAB_map ros package

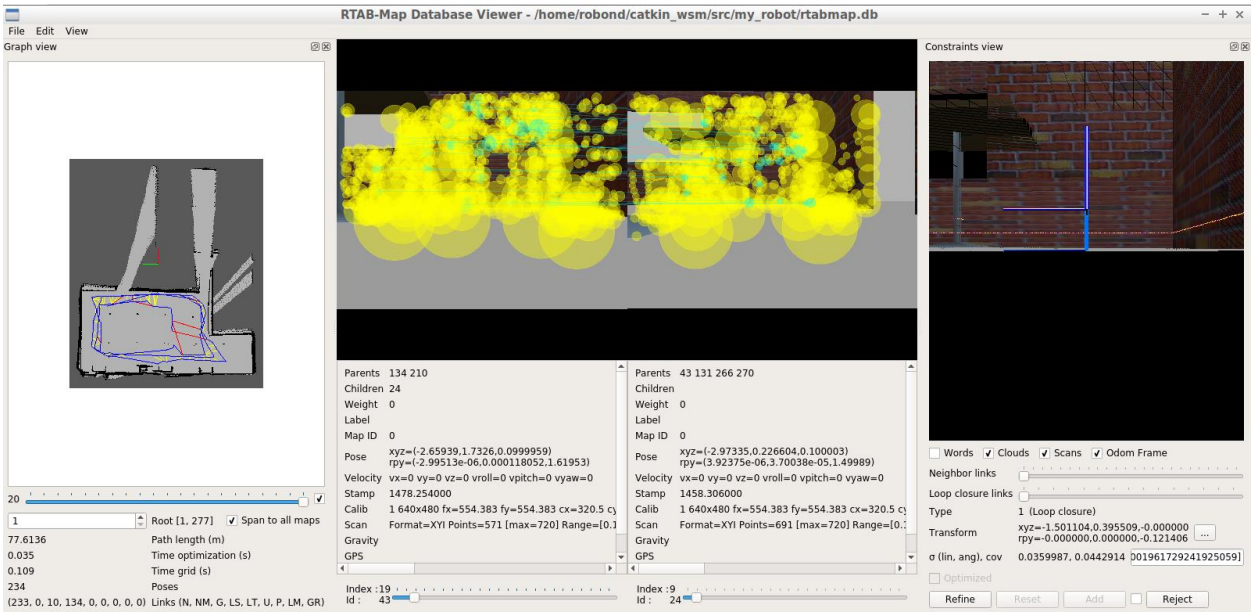


Figure 4.3 Visualization on RTAB database viewer

Getting three loop closures will be sufficient for mapping the entire environment. Loop closures can be maximized by going over similar paths two or three times. This allows for the maximization of feature

detection, facilitating faster loop closures. From the image above we were able to obtain 10 global loop closures.

In the image above we have our 2D grid map on the left in all its updated iterations and the path of the robot. In the middle we have images from the mapping process. The images in yellow are generated when features are detected by the detection algorithm. On the right we have the constraint view. This is where we can identify where and how the neighboring links and loop closures were created.

The second package used is the SLAM_gmapping package. The package uses a Raoblackwelized particle filter. Each particle carries an individual map of the environment. The robot navigates within its environment updating the information held within the particles. Loop closures occur when a current sensor reading matches with a sensor reading taken earlier. Loop closure is characterized by reduced particles that eventually converge. A map of best fit is generated following a loop closure.

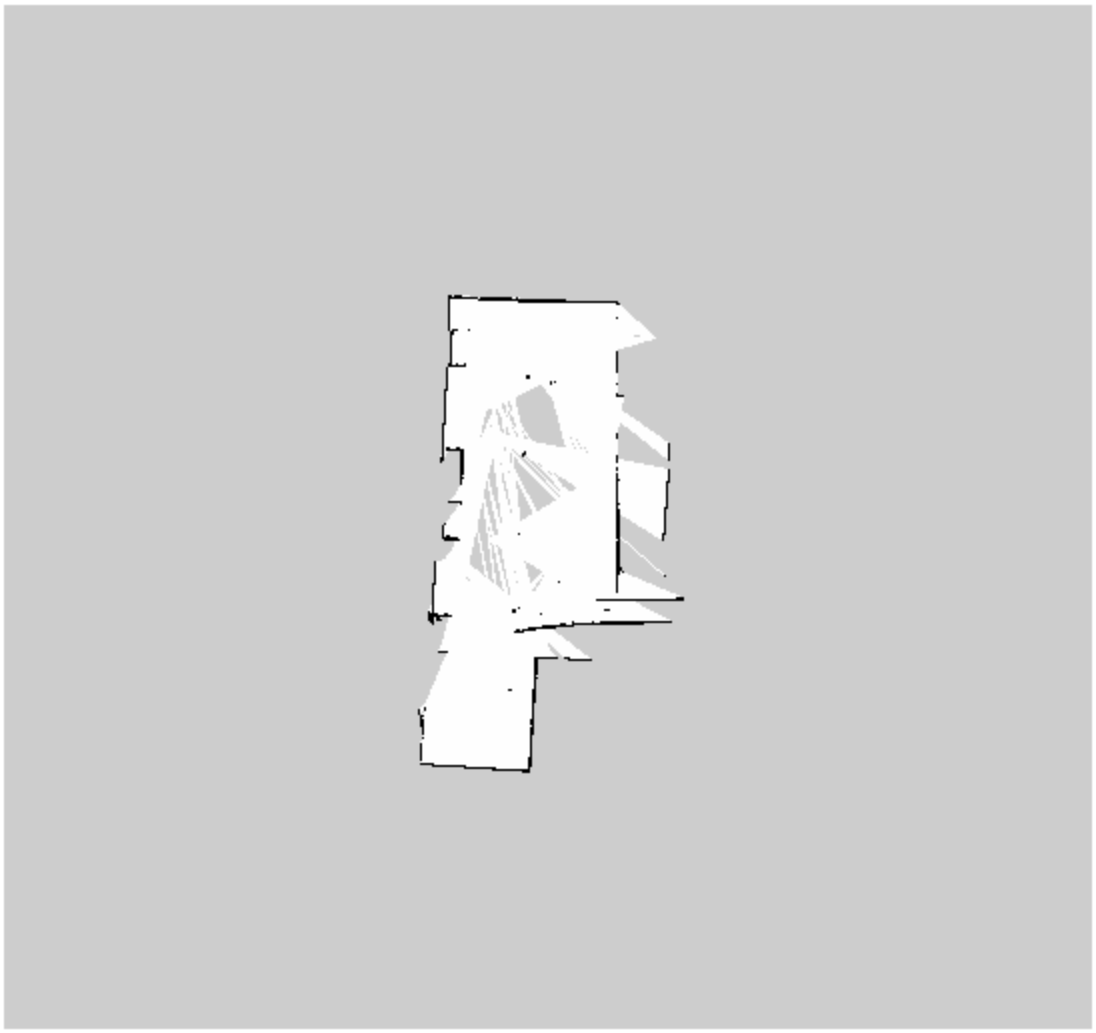


Figure 4.4 pgm image generated using SLAM_gmapping

```
image: map.pgm
resolution: 0.050000
origin: [-12.200000, -12.200000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196

```

Figure 4.5 yaml file containing metadata on the pgm image

If the robot continues to move around within its environment, different views of the same feature are obtained improving the quality of the map. The .yaml file generated provides metadata on the .pgm file. .pgm files are mostly used within AMCL packages.

4.1.2. LOCALIZATION

Two ROS packages used to perform localization.

The first ROS package is the Adaptive Monte Carlo Localization package. The AMCL package generates particles within the robot's environment. The particles represent guesses of the robot's pose. As the robot moves around the environment, the particles disperse since the robot's exact position is unknown. Updates in sensor measurements and the motion model result in particle convergence.

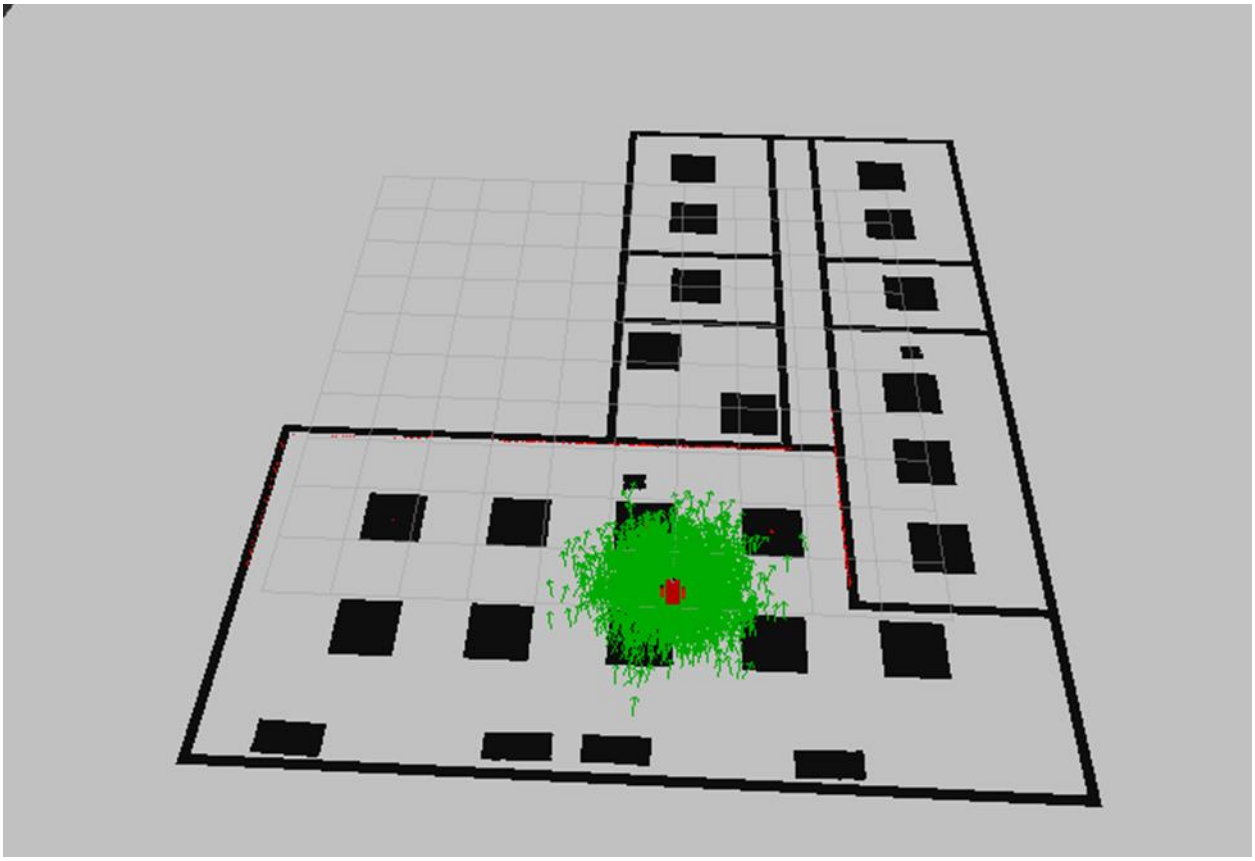


Figure 4.6 Robot's initial pose

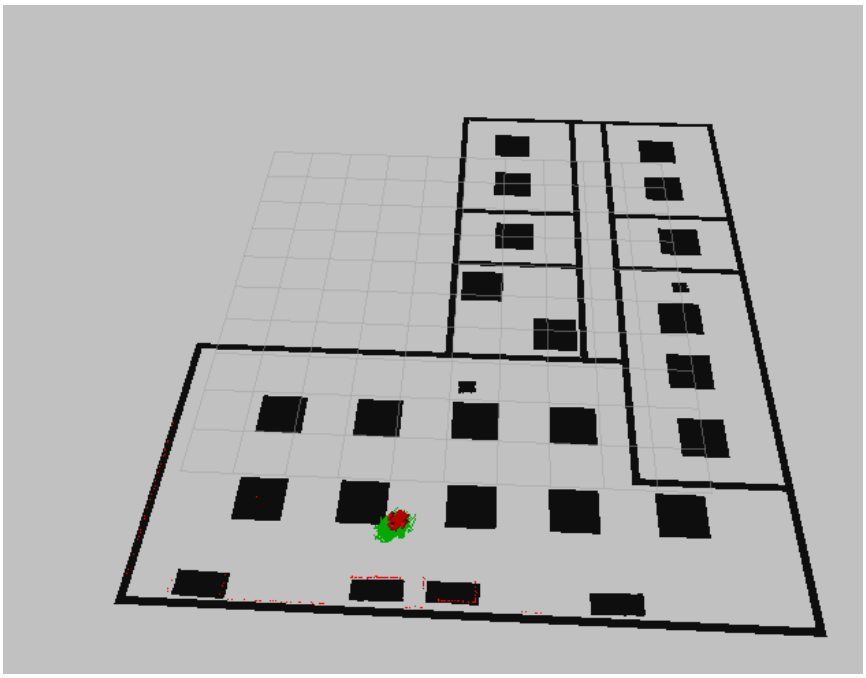


Figure 4.7 Robot's final pose

AMCL uses kullback-leibler divergence sampling to reduce the number of regenerated particles during the resampling stage. This significantly reduces the computational cost.

The second package used is the RTAB_map ROS package.

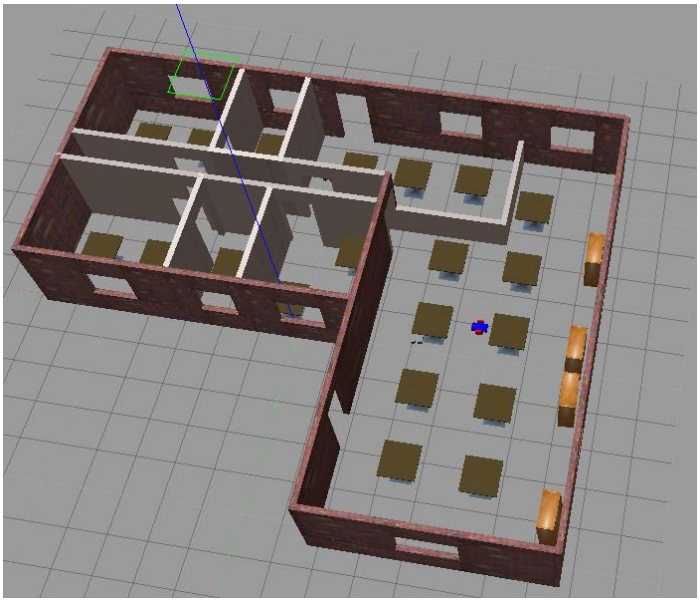


Figure 4.8 Initial robot pose

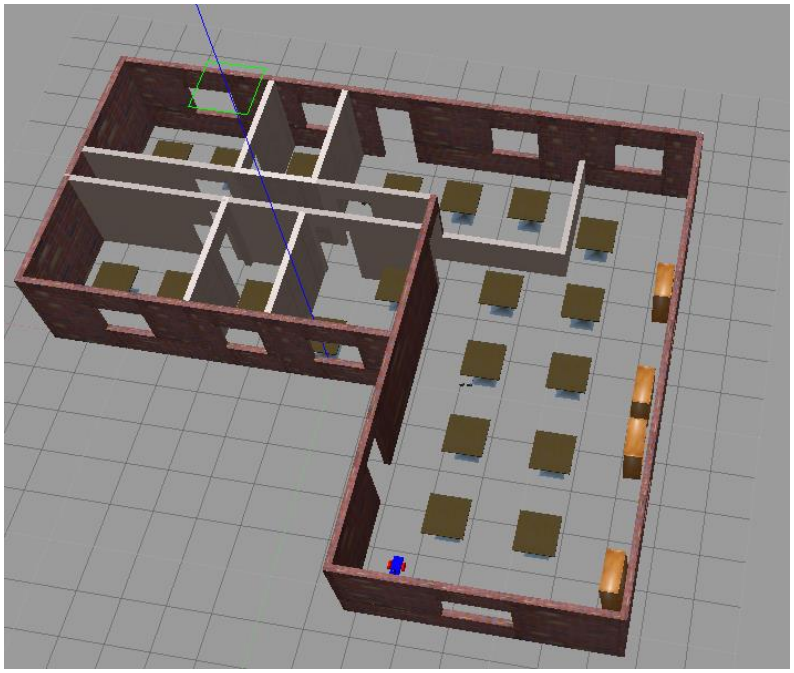


Figure 4.9 Final robot pose

The axes in the image below represent the initial and final robot poses.

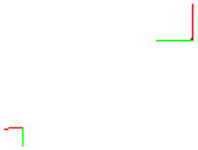


Figure 4.10 Visualization on RTAB database viewer

4.1.3. PATH PLANNING AND OBSTACLE AVOIDANCE

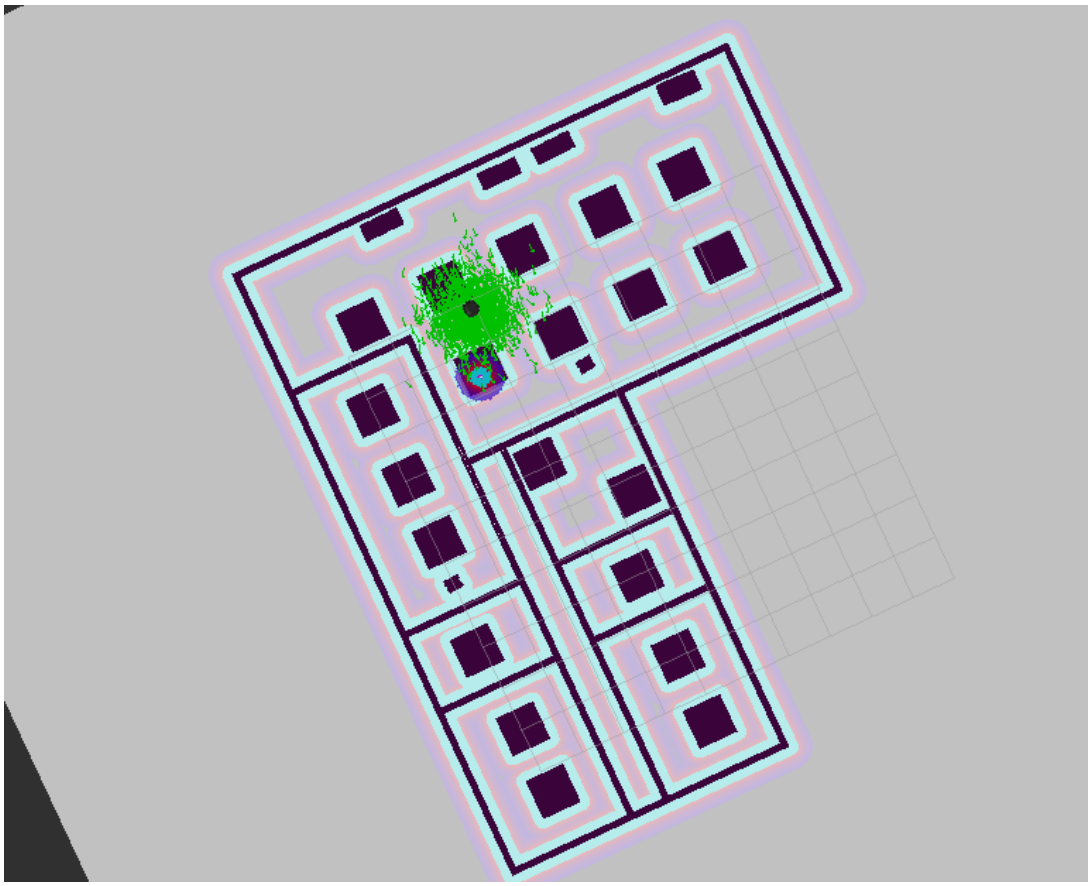


Figure 4.11 Initial robot pose

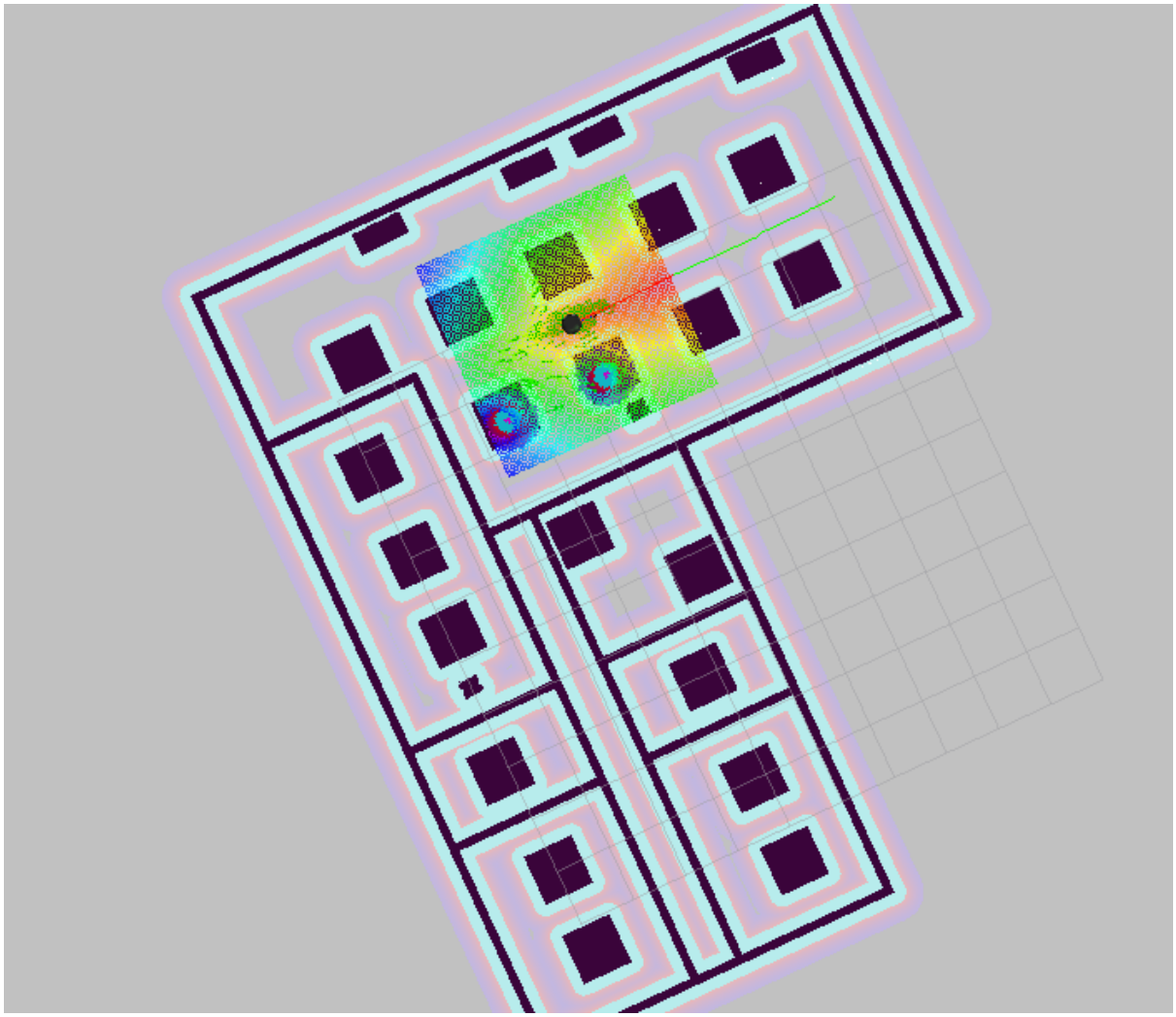


Figure 4.12 Robot navigating within its environment

The robot begins to move within its world after planning its trajectory as seen in the image above. ROS Navigation_stack and Move_base packages are used to receive a 2D navigational goal and to plan an obstacle free path between the initial/starting configuration and the goal configuration.

4.2. REAL WORLD IMPLEMENTATION RESULTS:

4.2.1. ROBOT:

We assembled a four wheeled robot that could be teleoped from a ROS-enabled Virtual Machine. The robot was powered directly from the Mains supply. It consisted of a Raspberry Pi 4 Model B, an Arduino Uno, a motor shield, an Xbox 360 Microsoft Kinect sensor, a 9V rechargeable battery for the motor and 12V rechargeable battery for the camera.

Figure 4.13 Grass cutter

4.2.2. CAMERA:

We set up the drivers for the camera and were able to get a feed from it, through the Raspberry Pi and to our Virtual Machine.

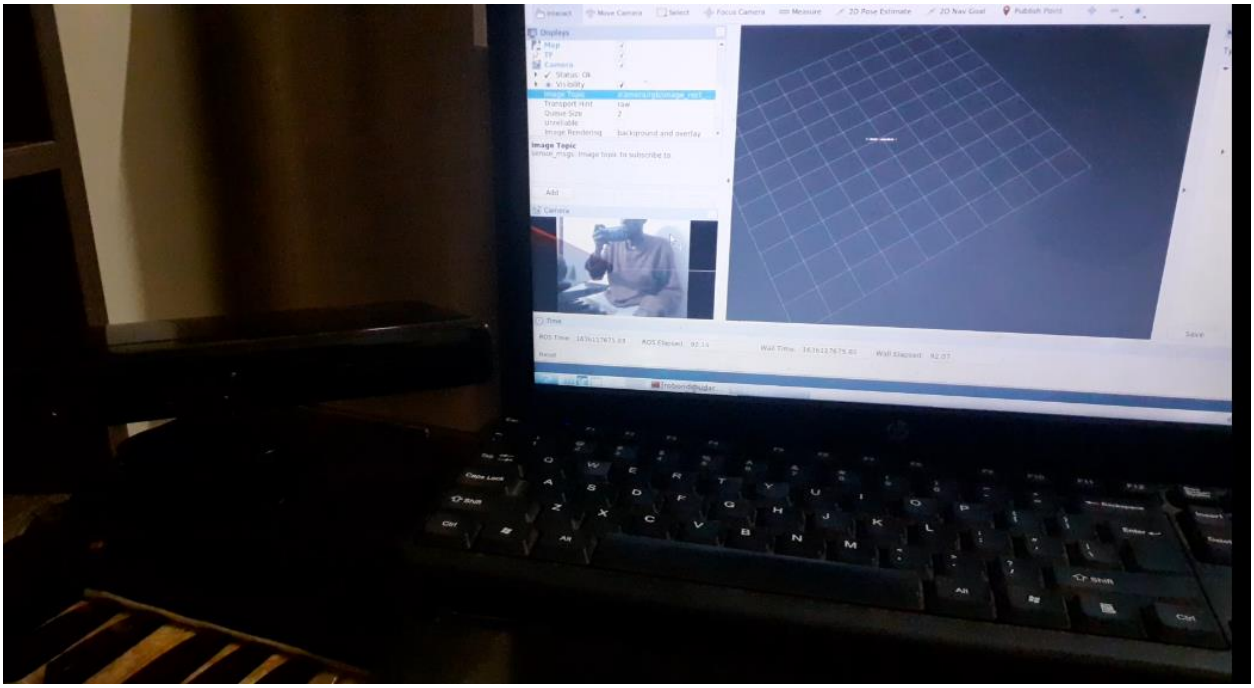


Figure 4.14 Kinect Camera

4.2.3. MAPPING:

We carried out RTAB-Mapping and generated both 3-D map and Graph view (2-D map) of our environment as shown:

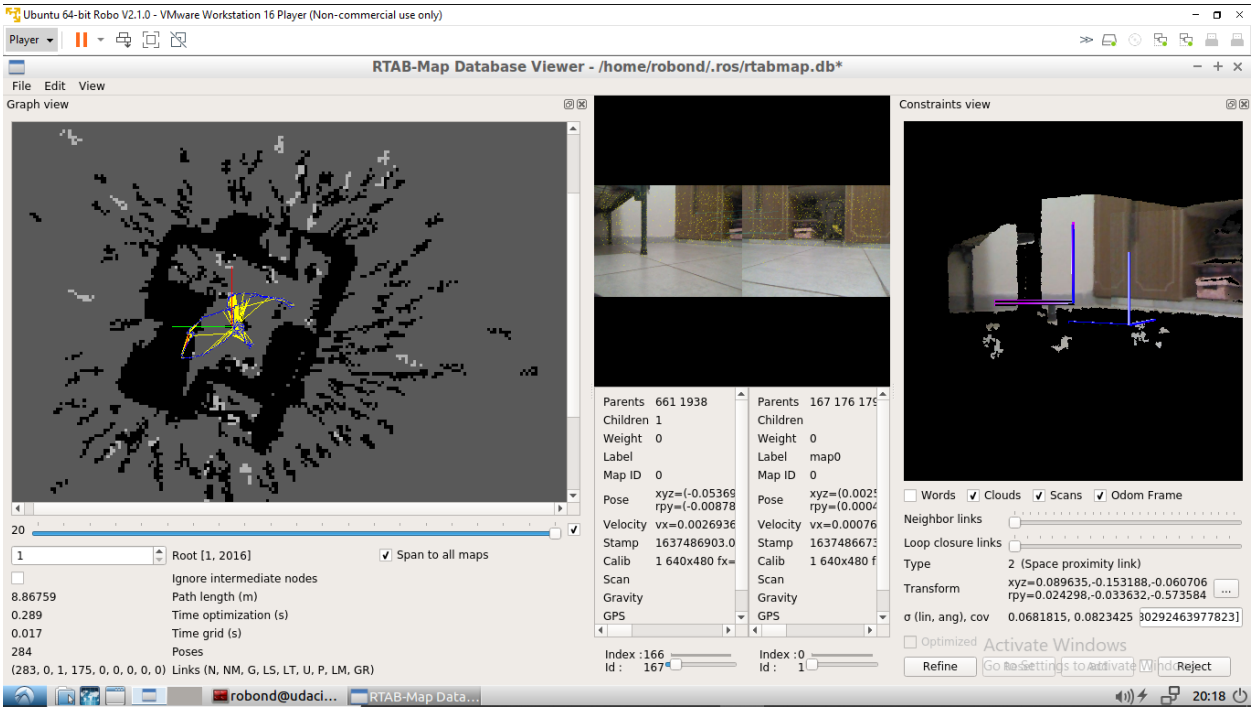


Figure 4.15 rtabmap.db

4.2.3. LOCALIZATION

We were able to localize the robot in the mapped environment using the RTAB-Map ROS package as shown below:

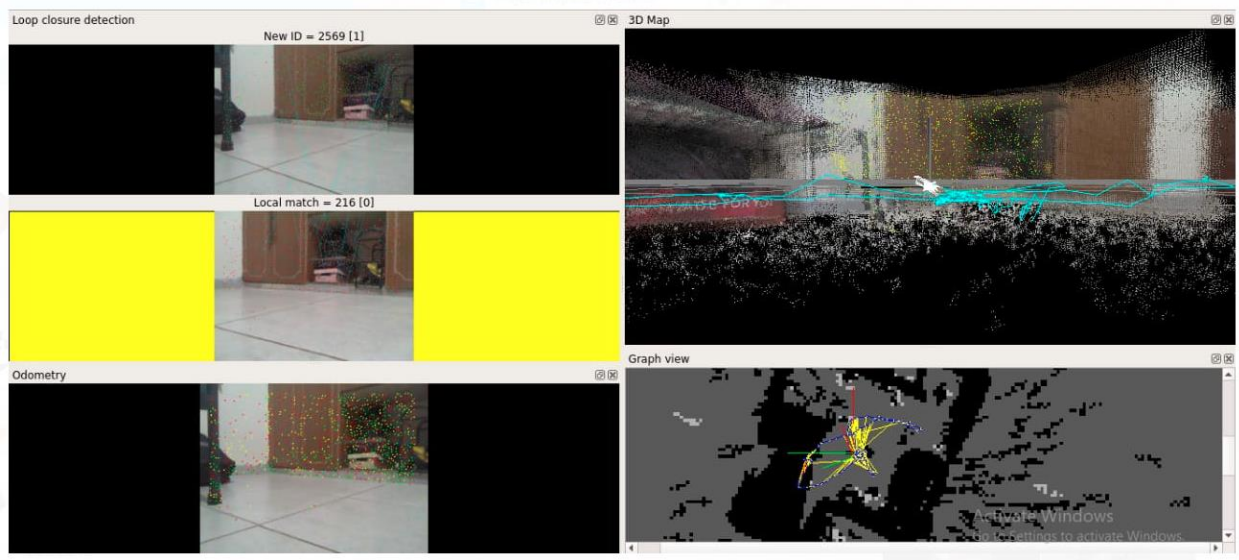


Figure 4.16 Real world localization

5. CONCLUSION AND RECOMMENDATIONS

5.1. SIMULATION CONCLUSIONS

- Robot successfully built using Unified Robot Description Format (URDF) and housed within a world built using the Gazebo Building Editor.
- Mapping achieved using Rtabmap and SLAM_gmapping packages.
- Localization achieved using AMCL and Rtabmap packages.
- Path planning and obstacle avoidance achieved using ROS Navigation Stack and Move_base packages.

5.2. REAL_WORLD ROBOT CONCLUSIONS

- Robotic grass cutter was successfully assembled consisting of a robotic chassis, Raspberry Pi 4, an Arduino Uno, an Xbox 360 Microsoft Kinect sensor, DC geared motors, 9V battery and 12V battery.
- Was able to tele-op the grass cutter.
- Was able to obtain, RGB, depth and point cloud data from the camera.
- A map was successfully built using RTAB-Map ROS package.
- The grass cutter was able to localize itself within the mapped environment.

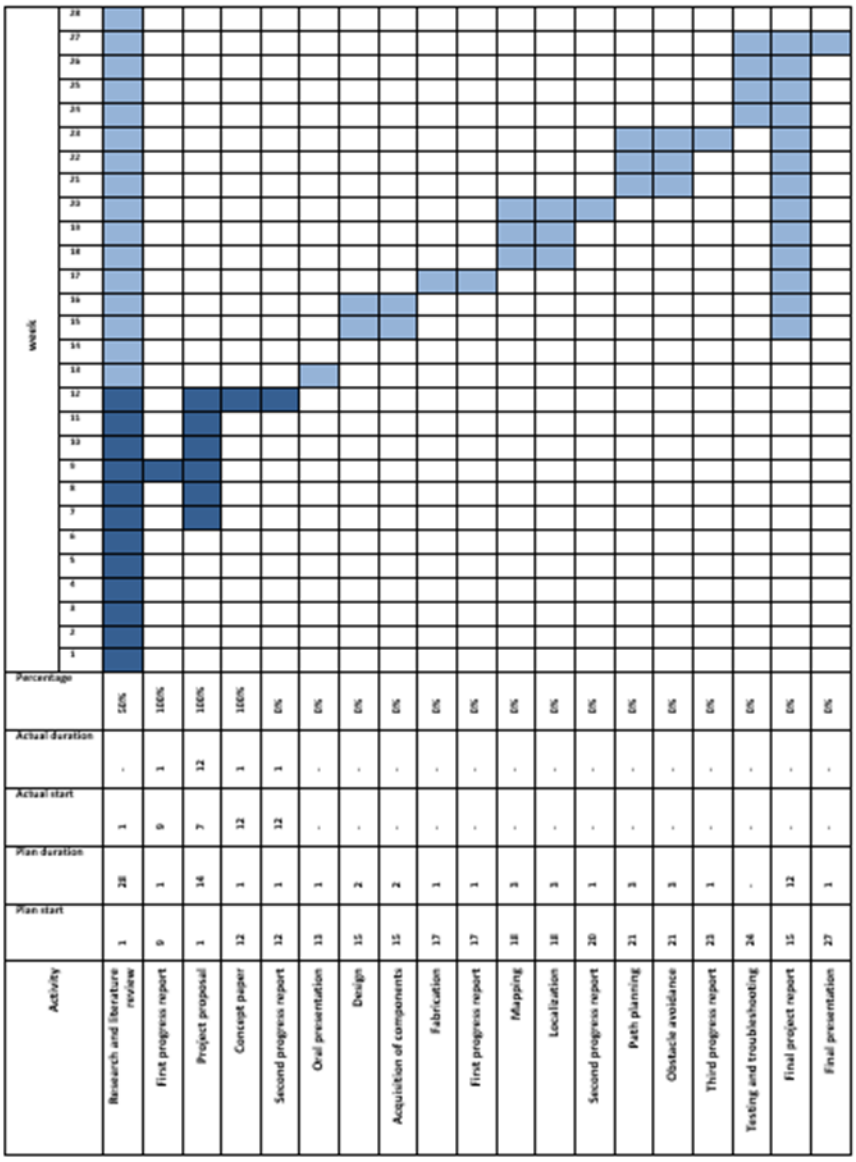
5.3. CHALLENGES FACED:

- The time allocated was not enough to fully implement the project.
- Few local component vendors, limiting variety.
- Estimated component price varied significantly from the actual sale price.
- Miscellaneous expenditure was higher than what was estimated.

5.4. RECOMMENDATIONS:

- Other final semester units should be pushed to other years and semesters.
- The school should have a repository of popular components.

6. PROJECT TIME-PLAN



Activity	Plan start	Plan duration	Actual start	Actual duration	Percentage
Research and literature review	1	28	1	-	50%
First progress report	9	1	9	1	100%
Project proposal	1	14	7	12	100%
Concept paper	12	1	12	1	100%
Second progress report	12	1	12	1	0%
Oral presentation	13	1	-	-	0%
Design	15	2	-	-	0%
Acquisition of components	15	2	-	-	0%
Fabrication	17	1	-	-	0%
First progress report	17	1	-	-	0%
Mapping	18	3	-	-	0%
Localization	18	3	-	-	0%
Second progress report	20	1	-	-	0%
Path planning	21	3	-	-	0%
Obstacle avoidance	21	3	-	-	0%
Third progress report	23	1	-	-	0%
Testing and troubleshooting	24	-	-	-	0%
Final project report	15	12	-	-	0%
Final presentation	27	1	-	-	0%

Table 6.1: Project time-plan

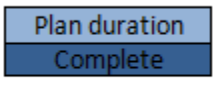


Table 6.2: Project time-plan color code

7. PROJECT BUDGET:

Quantity	Component	Cost in Ksh.
1	Robot chassis	1,500
4	Wheel	(Included in the Chassis)
2	Wheel Encoder disk	(Included in the Chassis)
4	DC Gear Motor	(Included in the Chassis)
1	Raspberry Pi 4	10,350
1	Raspberry Pi Charger	700
1	Arduino Uno	1,200
1	SD Card	1,500
1	Connecting Wires	600
1	Kinect Camera	1,000
1	9V Battery	900
1	Battery charger	400
1	12V battery	750
1	Motor Driver	500
1	Soldering Wire	250
1	Udacity Robotics Software Engineer(Nanodegree Program)	36,000
1	Hard Disk Drive	5,500
TOTAL		61,150

Table 7.1. Project budget.

References

[1] https://infographicsite.com/infographic/the-evolution-of-the-lawn-mower/.

[2] B. S. e. al, Springer Tracts in Advanced Robotics Volume 55, Berlin-Heidelberg: Springer-Verlag , 2009.

[3] F. R. e. al, "A Review of Robots: Concepts, methods, theoretical framework and applications," *International Journal of Advanced Robotic Systems*, pp. 1-22, 2019.

[4] X. Wang, "2D Mapping Solutions for Low Cost Mobile Robot," Royal Institute of Technology, School of Computer Science and Communication.

[5] B. S. e. al, Springer Handbook of Robotics, Berlin-Heidelberg: Springer-Verlag, 2008.

[6] H. M. e. al, "High Resolution Maps for Wide Angle Sonar," in *IEEE International Conference on Robotics and Automation*, The Robotics Institute, Carnegie-Mellon University, 1985.

[7] D. F. e. al, "Map-based navigation in mobile robots- A review of localization strategies," *Cognitive Systems Research*, pp. 243-282, 2003.

[8] S. Thrun, Probabilistic Robotics.

[9] S. Thrun, "Robust Monte Carlo Localization for Mobile Robots," vol. 128, pp. 1-3, 2001.

[10] S. Thrun, Probabilistic Robotics, Cambridge, Massachusetts: The MIT publisher, 2006.

[11] M. J. Pathak, "Comparative Analysis of Search Algorithms," *International Journal of Computer Applications*, vol. 179, 2018.

[12] V. A. Bhavesh, "Comparison of Various Obstacle Avoidance," *International Journal of Engineering Research & Technology (IJERT)*, vol. 4, p. 12, 2015.

[13] H. X. S. A. e. al, "A state-of-the-art analysis of Obstacle Avoidance Methods from the Perspective of an Agricultural Sprayer UAV's Operation Scenario," *Agronomy,* vol. 11, p. 6, 2021.