AIN SHAMS UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER AND SYSTEMS ENGINEERING

Simultaneous Localization And Mapping

Authors:
David NADER
George MOHSEN
Kirollos SHERIF HENRY
Kirollos SHERIF WADIE

Supervisor:
Dr. Hossam HASSAN

A thesis submitted in partial fulfillment of the requirements for the B.Sc.degree

July 2019

AIN SHAMS UNIVERSITY

Abstract

Faculty of Engineering

Computer and Systems Engineering

B.Sc.degree

Simultaneous Localization And Mapping

by David NADER
George MOHSEN
Kirollos SHERIF HENRY
Kirollos SHERIF WADIE

The SLAM is actually two targets but there is no way to accomplish one target without the other as the localization is very important to make the mapping and also the mapping is very important to accomplish localization, that's why the "Simultaneous Localization and mapping" is explained using the egg chicken conflict as it is impossible to have the localization accomplished without mapping and the opposite is always true so we have to make the both done online together at the same time .

SLAM is the process of making the robot move in a totally unknown environment and be able to detect its location with respect to the environment and detect the the surrounding and is able to draw a map for all the explored part of the environment.

Acknowledgements

We Would like to thank Dr. Hossam Hassan for his guidance and motivation during the project, and we want also to thank him for his ideas and his guide to find our way out from each and every problem.

Dr. Hossam motivated us to sky's our limit during the project and motivated us to keep searching always for what is better and find the perfect solution and output for each and every part.

Dr. Hossam was always our source of inspiration during the progress of the project and his guide was always leading us to what is right and how to reach what is right.

We would like to thank Dr. Maged Ghoneima and "Ihub" for the IGP program, as we have used the RPLidar that "Ihub" gave us to complete our project, the Lidar was the main item that we used in our project and the Lidar we took from "Ihub" was our only way to have our project done.

Contents

List of Figures

List of Tables

List of Abbreviations

SLAM	**S**imultaneous **L**ocalization **A**nd **M**apping
ROS	**R**obot **O**perating **S**ystem
EKF	**E**xtended **K**alman **F**ilter
PWM	**P**ulse **W**idth **M**odulation
RPM	**R**ounds **P**er **M**inute

Chapter 1

Introduction

The term SLAM is as stated an acronym for Simultaneous Localization And Mapping. It was originally developed by Hugh Durrant-Whyte and John J. Leonard based on earlier work by Smith, Self and Cheeseman . Durrant-Whyte and Leonard originally termed it SMAL but it was later changed to give a better impact. SLAM is concerned with the problem of building a map of an unknown environment by a mobile robot while at the same time navigating the environment using the map.

1.1 Problem

The main problem of the SLAM is that the SLAM is actually composed of two objectives and neither of them can precede the other, the two objectives are localization and mapping.

The map is needed for localization and the localization is needed to build the map and that is always called chicken-or-egg problem. [3]

FIGURE 1.1: chicken-or-egg problem

1.2 Hardware

To do SLAM there is the need for a mobile robot and a range measurement device. [1]

1.2.1 Robot

Important parameters to consider are ease of use, odometry performance and price. The odometry performance measures how well the robot can estimate its own position, just from the rotation of the wheels.[1] The robot should not have an error of more than 2 cm per meter moved and 2° per 45° degrees turned. Typical robot drivers allow the robot to report its (x,y) position in some Cartesian coordinate system and also to report the robots current bearing/heading. [1]

1.2.2 Range Finder

There are three options for the Range finder and we will discus each of them in the following sections. [1]

Laser scanner

Advantages : They are very precise, efficient and the output does not require much computation to process.

Disadvantages : Problems with laser scanners are looking at certain surfaces including glass, where they can give very bad readings (data output). Also laser scanners cannot be used underwater since the water(out of scope) disrupts the light and the range is drastically reduced. [1]

Sonar

Sonar was used intensively some years ago. They are very cheap compared to laser scanners. Their measurements are not very good compared to laser scanners and they often give bad readings. Where laser scanners have a single straight line of measurement emitted from the scanner with a width of as little as 0.25 degrees a sonar can easily have beams up to 30 degrees in width. Underwater, though, they are the best choice and resemble the way dolphins navigate.[1]

Vision

Traditionally it has been very computationally intensive to use vision and also error prone due to changes in light. Given a room without light a vision system will most certainly not work. In the recent years, though, there have been some interesting advances within this field. Often the systems use a stereo or triclops system to measure the distance. Using vision resembles the way humans look at the world and thus may be more intuitively appealing than laser or sonar. Also there is a lot more information in a picture compared to laser and sonar scans. This used to be the bottleneck, since all this data needed to be processed, but with advances in algorithms and computation power this is becoming less of a problem. [1]

We conclude that

From all the previous we conclude that the best choice and more efficient one is the Laser Scanner.
Sonar is not accurate and it is used mainly for underwater detection (out of scope).
Camera (Vision) needs high computational power which isn't available in a robot.

Laser Scanner has high accuracy, high precision, high range, needs low computation power and scans all the around environment.

1.3 SLAM Process

The SLAM process consists of a number of steps. The goal of the process is to use the environment to update the position of the robot. Since the odometry of the robot (which gives the robots position) is often erroneous we cannot rely directly on the odometry. We can use laser scans of the environment to correct the position of the robot. This is accomplished by extracting features from the environment and re-observing when the robot moves around. An Extended Kalman Filter is the heart of the SLAM process. It is responsible for updating where the robot thinks it is based on these features. These features are commonly called landmarks. The EKF keeps track of an estimate of the uncertainty in the robots position and also the uncertainty in these landmarks it has seen in the environment. An outline of the SLAM process is in the Figure 1.2 .

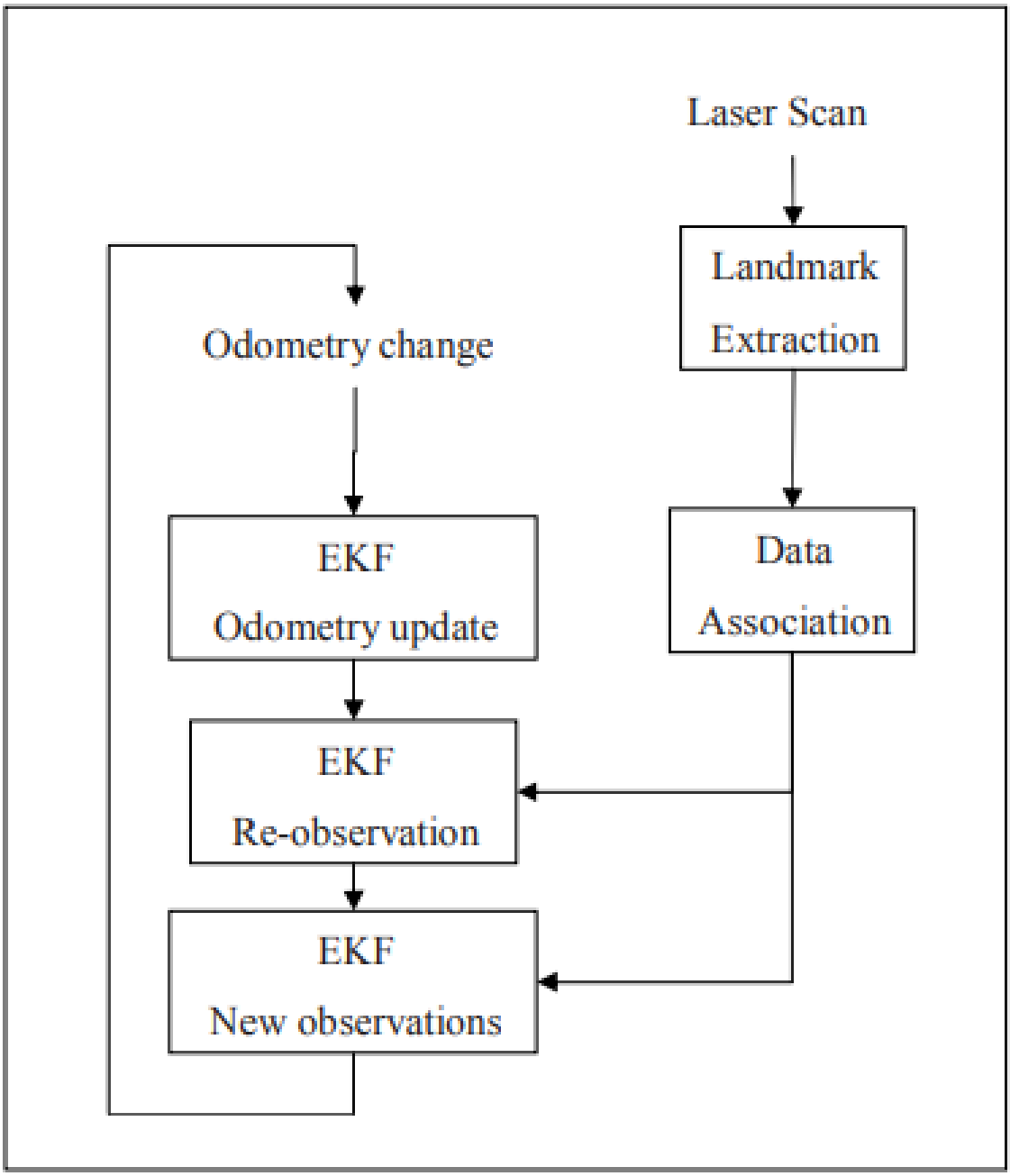


FIGURE 1.2: SLAM Process

When the odometry changes because the robot moves the uncertainty pertaining to the robots new position is updated in the EKF using Odometry update.[1] Landmarks are then extracted from the environment from the robots new position. The robot then attempts to associate these landmarks to observations of landmarks it previously has seen. Re-observed landmarks are then used to update the robots position in the EKF. Landmarks which have not previously been seen are added to the EKF as new observations so they can be re-observed later.

1.3.1 Landmark Extraction

The first step in the Slam Process is the landmark extraction the first thing that is done by the robot when exploring a new place is extracting landmarks in this place.[1]

The landmark extraction can be considered the core of the SLAM process as it is the step that all the later steps depends on, this step is usual done using the range finder sensor installed on the robot. The extracted landmarks are the reference of the robot during the exploration process as the robot used the landmarks to know its current position and its new position, Figure 1.3 shows the Landmark extraction process.

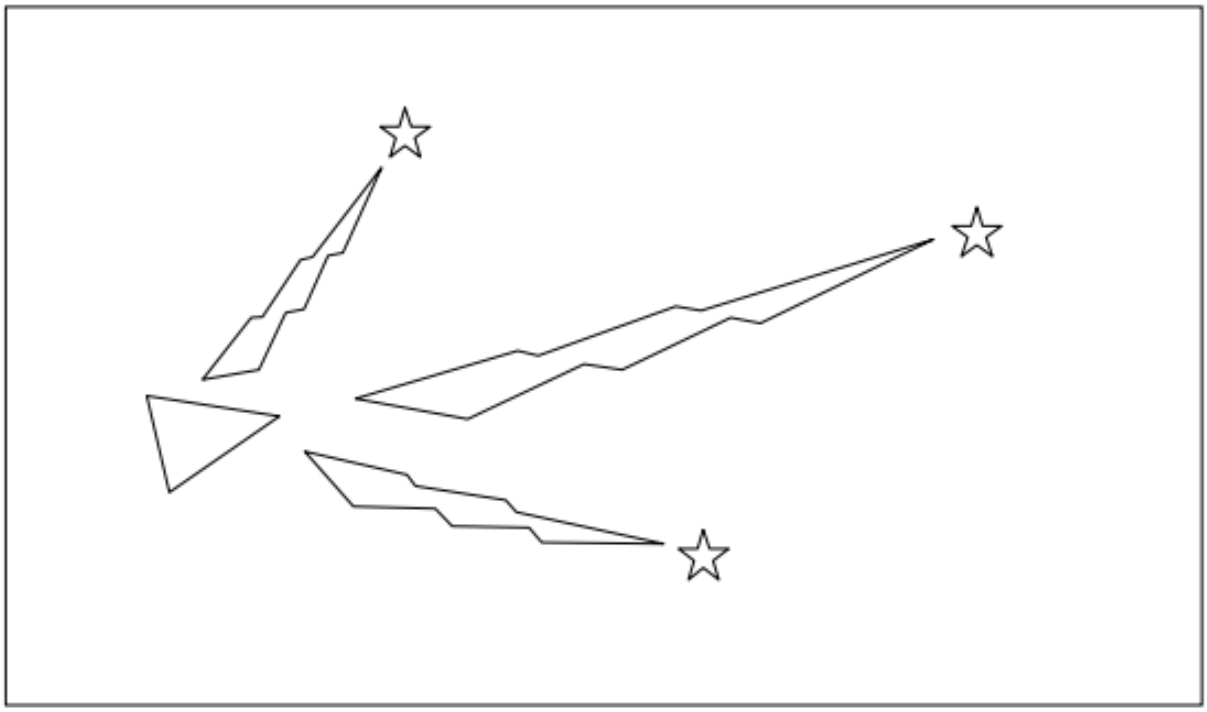


FIGURE 1.3: Landmark Extraction

1.3.2 Exploration

After the robot extract landmarks the starts moving through the environment between the landmarks and using the landmarks to know its current and detect its new position, Figure 1.4 shows the robot moving through the extracted landmarks.

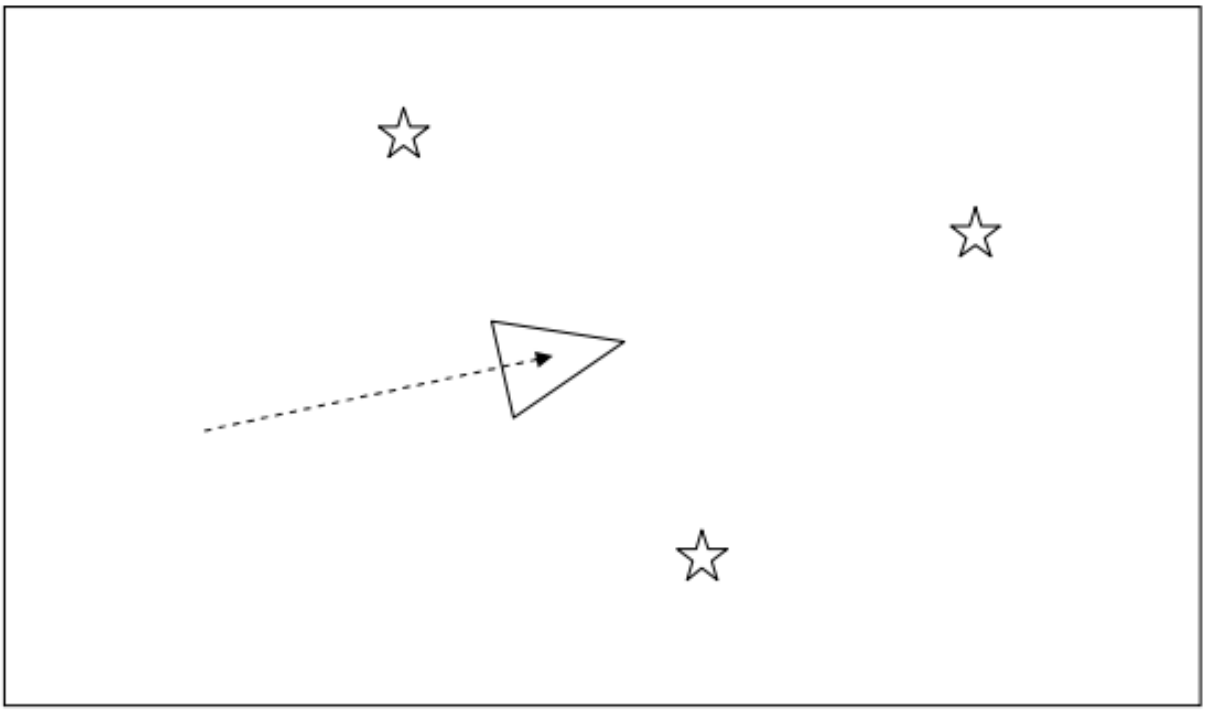


FIGURE 1.4: The robot moving through the landmarks

1.3.3 Re-observation

The robot re-observes the landmarks and compare the expected position with the current position.

The robot expect its position by calculating the distance the robot moves and its previous position relative to the landmarks.

The Figure 1.5 illustrates the step of re-observing landmarks.

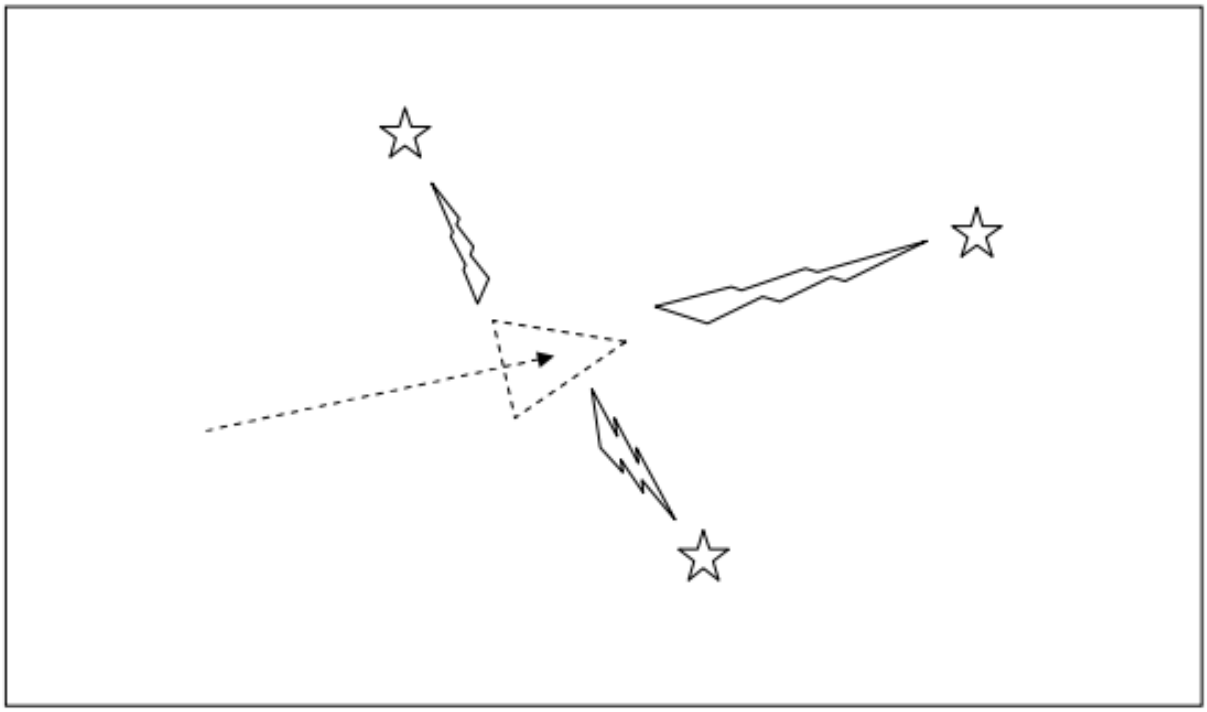


FIGURE 1.5: The robot is re-observing its position

1.3.4 Error

After the robot re-observers the landmarks it detects the error between its expected position and its true position.

Figure 1.6 Shows the error between the expected position of the robot and the actual position.

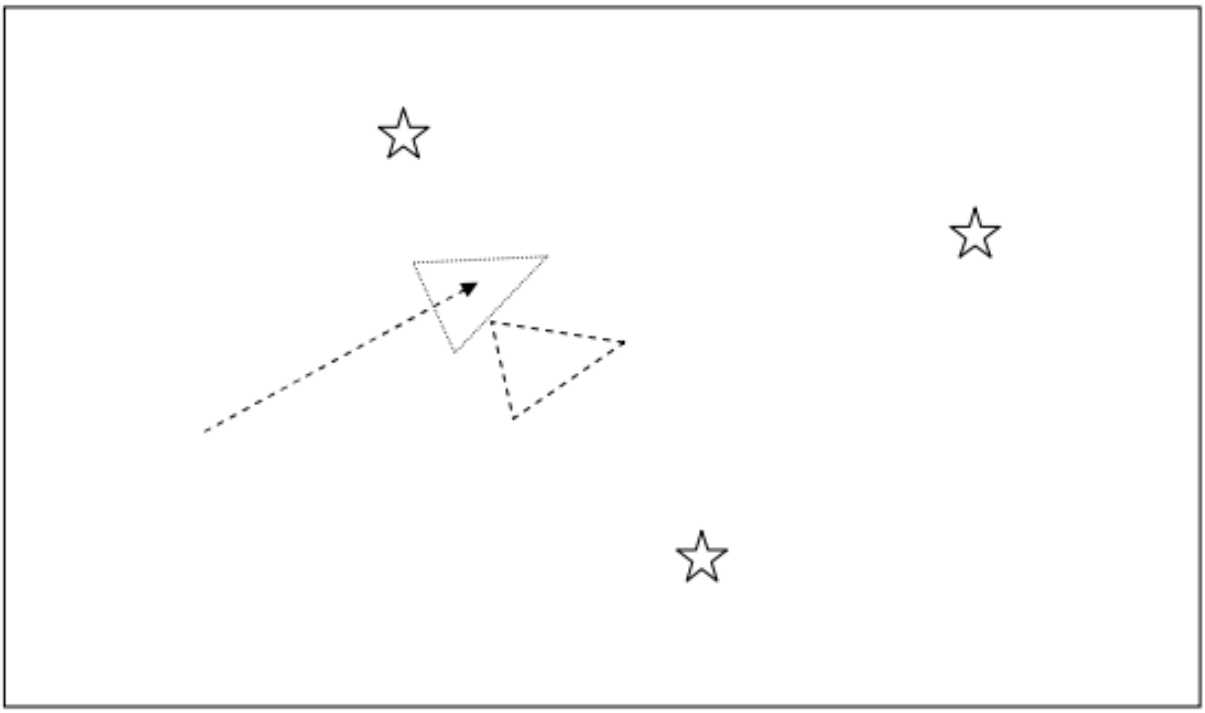


FIGURE 1.6: Error between the expected position and the actual position of the robot

1.3.5 Position Update

After The robot re-observers its position it updates its expected position to its correct position and it manged to know its correct new position using the landmarks.

In Figure 1.7 the robot is updating its expected position to the true position after re-observing the landmarks.

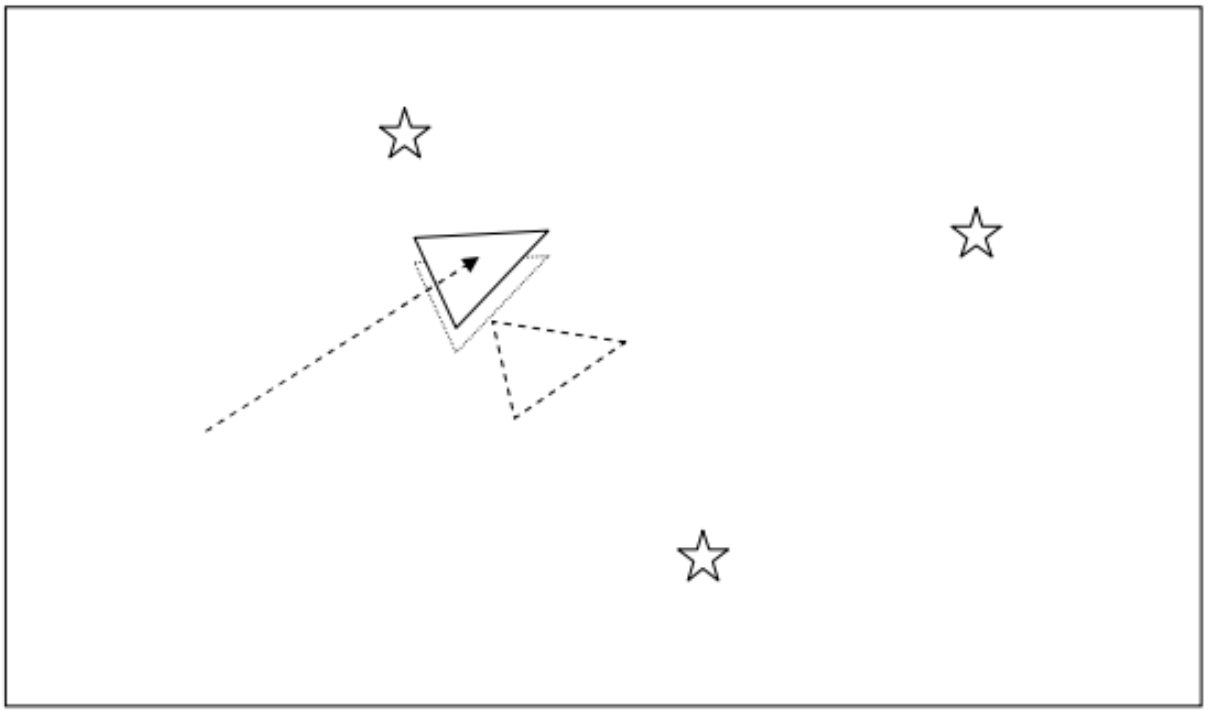


FIGURE 1.7: The robot is updating its position

Chapter 2

Related Work

Simultaneous Localization And Mapping it is not just a project for interest or proof of concept it is widely in many fields and can be helpful for different people in different fields, and it can be used in many non-engineering fields in this section we will discus briefly some applications for the SLAM and how it is very useful for many people.

2.1 Swarm Robotics

No Swarm robotics project can be done without accurate SLAM in swarm robotics all robots has to be localized, if the robots aren't localized swarm robotics can't be accomplished as the core of the Swarm robotics is the communication between the robots and robots communication with each other is useless if robots aren't localized accurately.

2.2 Under Water Exploration

SLAM was widely used in the under water exploration.

SLAM was used for mapping caves and another places under water that humans can't explore, a Sonar was used for underwater mapping.

Some hidden underwater caves and tunnels was discovered using SLAM. [12]

2.3 Historical Places

SLAM can be also be used in mapping some historical places that are very small or dangerous and humans can't explore them selves.

If a high accuracy SLAM is used you can guarantee that the map you get is the same as explored place.

There are some pharaonic tombs that can be explored and drawn easily using SLAM.

2.4 Underground exploration

The SLAM was used for underground exploration for finding mines and many other places underground that have many natural resources and can't be visited for the first time by someone as it might be dangerous.

2.5 Mine Detection

SLAM can be used with metal detector to detect mines if a place is known to have many mines. A robot with a SLAM algorithm can be used to detect mines and mark there places on the map and this can be useful to get rid of old mines hidden in many places since old wars

2.6 Caves

There are many dangerous caves and places that is very dangerous for a man to visit it can be visited and mapped using a robot with a SLAM algorithm and know what is exactly in this place and how it looks.

Chapter 3

Custom-Made Robot

A Custom-Made robot is designed as it is flexible and can be changed at any time during the project, can add something to the robot, can remove some parts of the robot and so this was our best choice as it enables us to try several things during our project and change what is needed during our work and try all possibilities and decide what is best track to keep going in.

3.1 Design

A Three-level robot is designed, 3 plastic sheets are installed on 3 different levels using spacer (Standoffs).
The robot plastic sheets have many holes for the fixation of sensors and many other items on the robot, this design makes it easy to add and remove any sensor or item from the robot and allow us to easily try different sensor to decide what we are going to use and how to reach our objective, Figure 3.1 Shows the top view of the robot.

FIGURE 3.1: Top view of the robot and the Lidar appears in it

The below figure is the side view of the robot with all the sensors, power supplies and raspberry pi mounted and most of the items appear and what item mounted in each layer in the robot, This can clearly appear in Figure 3.2.

FIGURE 3.2: Side view of the robot

3.2 Control

The robot is controlled using small computer, Raspberry Pi [10] Model 3b+, The raspberry [10] pi controls the robots by controlling the installed motors using a motor driver and also the raspberry pi use the reading of the sensors and by making some processing to sensors reading we became able to reach the project objective.

3.2.1 Communication

Communication to robot is established by communicating to the raspberry pi [10]. the communication to raspberry is established using RealVNC [9], a vnc server is installed on the raspberry pi and vnc viewer is installed on the computer that we need to communicate to the raspberry pi from.
This communication is established through two different ways either through the internet, both the server and the viewer are both connected to the internet by this was we can connect to the raspberry pi or the communication can be established using local network if both the raspberry pi and the viewer are both connected to the same network the viewer can view the raspberry pi and this way is much efficient, much faster communication and can rely on it better than the over internet communication.

3.3 Operating System

Ubuntu Mate operating System is used, it is installed on the raspberry pi and a Robot Operating System (ROS) is installed on the ubuntu mate this allows us to control the robot easily as the ROS has its working space and has packages the allows us to get the sensors reading in an easy way.

rplidar [11] package is used to get the RPLIDAR reading and hector slam algorithm is used, hector slam package subscribed to the topic rplidar package publishes. by this way we were able to make processing on the RPLIDAR readings and draw a map using the Lidar and also we managed to localize the robot using these packages using the ROS installed on the ubuntu mate on the raspberry pi 3b+.

3.4 Sensors

Some sensors are used to detect the environment and help the robots to navigate through the environment and as we have previously mentioned we managed to make processing on the sensors using the ROS installed on the raspberry pi

3.4.1 Lidar

The Main Sensor and the most important item in this project is the Lidar, using the lidar we managed to detect the whole surrounding and detect everything around the robot, using the Lidar reading we managed to draw the map of the environment and managed to localize the robot using hector slam algorithm.

3.4.2 Ultrasonic

Ultrasonic sensor was used for the obstacle avoidance algorithm as the robot keep moving until it faces an obstacles to changes its direction by this way the robot is able to navigate through the environment without any human control.

3.5 Power Supply

A Battery 7.2V 3000 mAh is used as a power supply for the motors and a power bank 1000 mAh and 2.4 A is used a power source for the raspberry pi and the Lidar.

3.5.1 Dc-to-Dc Converter

Before using power bank for the Lidar and raspberry pi we have tried to use buck converter to step down the voltage battery from 7.2V and to 5V to use the output of the converter to power the Lidar and the raspberry pi but the Electric current that came out from this connection was not enough or sufficient to power the raspberry

pi or the Lidar that was led us to use power bank to power Lidar and raspberry pi and keep the motors powered using the 7.2 V NiMH battery

3.6 Rotation Problem

The original design of this robot was using 4 motors and 4 wheels and rotating this robot by changing the speed of the wheels.

The motors in this robot are of low power so when many items are mounted on the robot it became of very large mass which made the robot unable to rotate left or right, the motors stall and became unable to move. So the original robot design wasn't good for mounting many items on it, the 4-wheels robot system isn't good for our project.

3.7 Metal Custer Wheel

The Metal Custer Wheel shown in figure 3.3 is added in the front part of the robot instead of the two front wheels.
This metal wheel made us able to make the robot rotate, the motion and rotation of the robot is now controlled using the two back wheels with its motors while the front metal wheel is just rolling.
By this way manged to make the robot rotate in an efficient way and solved the robot rotation problem

FIGURE 3.3: Metas Custer Wheel

3.8 Robot size Problem

The robot we used if of small size that which was a big problem in organizing project items and installing them on the robot.
The Lidar has to be installed in a level alone because any item beside the Lidar is a noise and misleading for the Lidar reading.

The Original design of the robot is two-level only which was not sufficient for the items we have and can't allow us to install the Lidar in level away from all other items, after many trial of organizing the items in two-levels robot only we failed to use two-levels robot.

3.8.1 Third Level

We have solved this problem by adding a third level to the robot which made us able to install the Lidar in the third level alone, Figure 3.4 show the third level of the robot installed on it the Lidar.

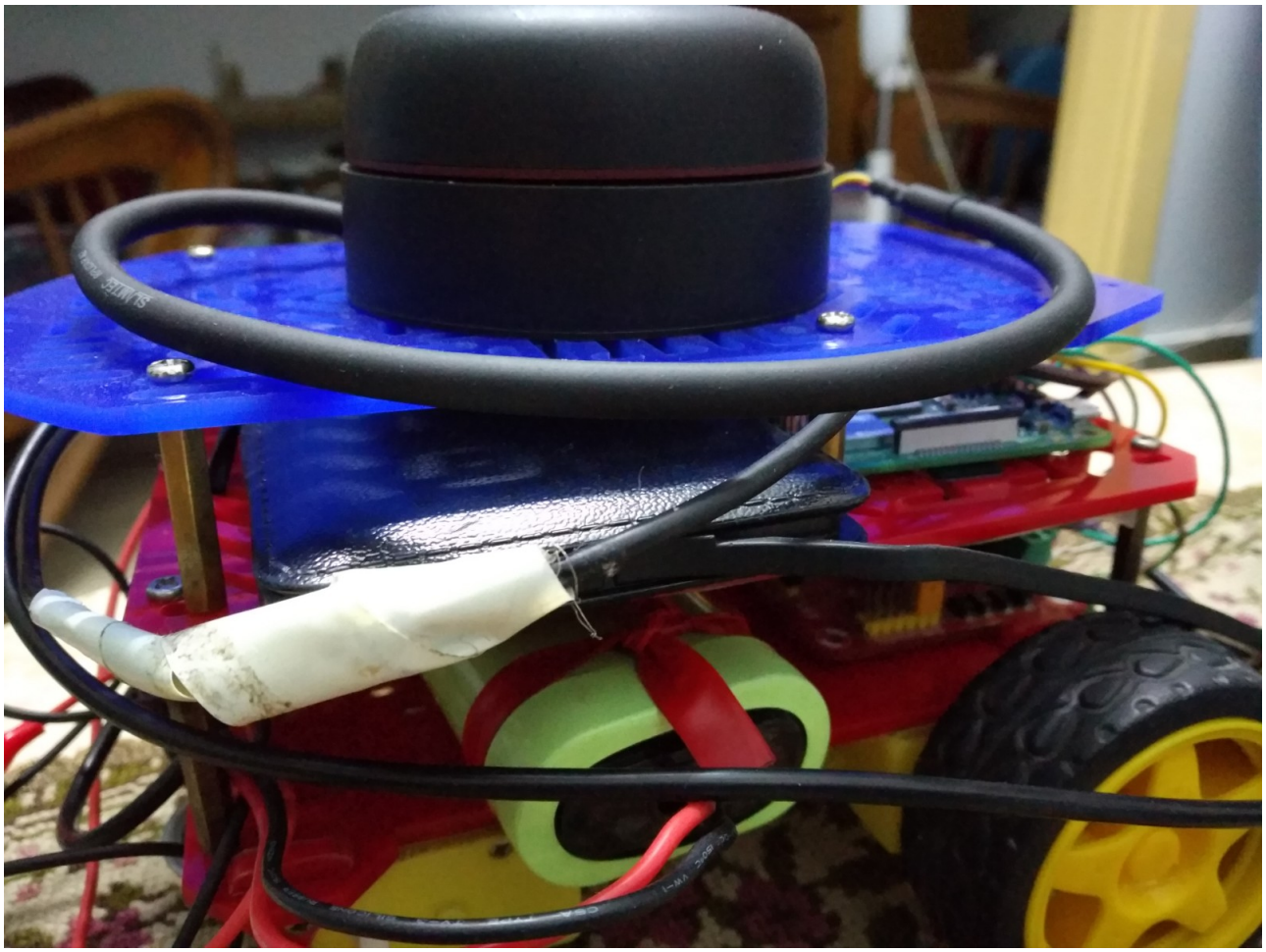


FIGURE 3.4: The Third level of the robot

3.9 Future Work

A better design robot should be done as this robot can't move in a straight line for long time it moves slight left or right during its movement.

If this project is changed to fully autonomous Robot the straight line motion of the robot is needed, so a better design robot is needed in the future.

Chapter 4

Robot Operating System (ROS)

The Robot Operating System (ROS) is a framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. Why? Because creating truly robust, general-purpose robot software is hard. From the robot's perspective, many problems that seem trivial to humans can actually encompass wild variations between instances of tasks and environments. Consider a simple "fetch an item" task, where an office-assistant robot is instructed to retrieve a stapler. First, the robot must understand the request, either verbally or through some other modality, such as a web interface, email, or even SMS.[2] Then, the robot must start some sort of planner to coordinate the search for the item, which will likely require navigating through various rooms in a building, perhaps including elevators and doors. Once arriving in a room, the robot must search desks cluttered with similarly sized objects (since all handheld objects are roughly the same size) and find a stapler.[2] The robot must then retrace its steps and deliver the stapler to the desired location. Each of those subproblems can have arbitrary numbers of complicating factors. And this was a relatively simple task! Dealing with real-world variations in complex tasks and environments is so difficult that no single individual, laboratory, or institution can hope to build a complete sys- tem from scratch. As a result, ROS was built from the ground up to encourage collab- orative robotics software development. For example, in the "fetch a stapler" problem, one organization might have experts in mapping indoor environments and could contribute a complex yet easy-to-use system for producing indoor maps. Another group might have expertise in using maps to robustly navigate indoor environments.[2]

Yet another group might have discovered a particular computer vision approach that works well for recognizing small objects in clutter. ROS includes many features specifically designed to simplify this type of large-scale collaboration.[2]

We're going to take a moment to introduce some of the key concepts that underlie the framework.[2] ROS systems are comprised of a large number of independent programs that are constantly communicating with each other. In this chapter, we'll discuss this architecture and look at the command-line tools that interact with it.[2]

4.1 The Ros Graph

A ROS system is made up of many different programs running simultaneously and communicating with one another by passing messages.[2] It is convenient to use a mathematical graph to represent this collection of programs and messages: the programs are the graph nodes, and programs that communicate with one another are connected by edges. [2]

4.2 Roscore

roscore is a service that provides connection information to nodes so that they can transmit messages to one another.[2] Every node connects to roscore at startup to regis- ter details of the message streams it publishes and the streams to which it wishes to subscribe. When a new node appears, roscore provides it with the information that it needs to form a direct peer-to-peer connection with other nodes publishing and sub- scribing to the same message topics. Every ROS system needs a running roscore, since without it, nodes cannot find other nodes. [2]

4.3 Roslaunch

roslaunch is a command-line tool designed to automate the launching of collections of ROS nodes.[2] On the surface, it looks a lot like rosrun , needing a package name and a filename: user@hostname $ roslaunch PACKAGE LAUNCH _ FILE
However, roslaunch operates on launch files, rather than nodes.[2] Launch files are XML files that describe a collection of nodes along with their topic remappings and parameters.[2] By convention, these files have a suffix of .launch. For example, here is talker _ listener.launch in the rospy _ tutorials package:

```
<launch>
        <node name="talker" pkg="rospy\_ tutorials"
        type="talker.py" output="screen" />
        <node name="listener" pkg="rospy\_ tutorials"
        type="listener.py" output="screen" />
</launch>\\
```

Each <node> tag includes attributes declaring the ROS graph name of the node, the package in which it can be found, and the type of node, which is simply the filename of the executable program. In this example, the output="screen" attributes indicate that the talker and listener nodes should dump their console outputs to the cur- rent console, instead of only to log files. This is a commonly used setting for debug- ging; once things start working, it is often convenient to remove this attribute so that the console has less noise. [2]

4.4 Catkin Workspace

catkin is the ROS build system: the set of tools that ROS uses to generate executable programs, libraries, scripts, and interfaces that other code can use.[2] If you use C++ to write your ROS code, you need to know a fair bit about catkin. [2]

4.4.1 Catkin

catkin comprises a set of CMake macros and custom Python scripts to provide extra functionality on top of the normal CMake workflow. CMake is a commonly used open source build system. If you're going to master the subtleties of catkin , it really helps if you know a bit about CMake . However, for the more casual catkin user, all you really need to know is that there are two files, CMakeLists.txt and package.xml, that you need to add some specific information to in order to have things work properly.[2]

You then call the various catkin tools to generate the directories and files you're going to need as you write code for your robots. These tools will be introduced as we need them throughout the book. Before we get to any of this, though, we need to introduce you to workspaces.[2]

4.4.2 Workspaces

Before you start writing any ROS code, you need to set up a workspace for this code to live in. A workspace is simply a set of directories in which a related set of ROS code lives. You can have multiple ROS workspaces, but you can only work in one of them at any one time. The simple way to think about this is that you can only see code that lives in your current workspace.[2]

Start by making sure that you've added the system-wide ROS setup script to your .bashrc file. Now, we're going to make a catkin workspace and initialize it:

```
user@hostname$ mkdir -p ~/catkin_ws/src
user@hostname$ cd ~/catkin_ws/src
user@hostname$ catkin_init_workspace
```

This creates a workspace directory called catkin_ ws (although you can call it anything you like), with a src directory inside it for your code.[2] The catkin_ init_ workspace command creates a CMakeLists.txt file for you in the src directory, where you invoked it. 1 Next, we're going to create some other workspace files:

```
user@hostname$ cd ~/catkin_ws
user@hostname$ catkin_make
```

Running catkin_ make will generate a lot of output as it does its work.[2] When it's done, you'll end up with two new directories: build and devel. build is where

catkin is going to store the results of some of its work, like libraries and executable programs if you use C++.[2] devel contains a number of files and directories, the most interesting of which are the setup files. [2]

4.4.3 Ros Packages

ROS software is organized into packages, each of which contains some combination of code, data, and documentation. 2 The ROS ecosystem includes thousands of publicly available packages in open repositories, and many thousands more packages are certainly lurking behind organizational firewalls. Packages sit inside workspaces, in the src directory. Each package directory must include a CMakeLists.txt file and a package.xml file that describes the contents of the package and how catkin should interact with it. Creating a new package is easy:

```
user@hostname$ cd ~/catkin_ws/src
user@hostname$ catkin_create_pkg my_awesome_code rospy
```

This changes the directory to src (where packages live) and invokes catkin_ cre ate_ pkg to make the new package called my_ awesome_ code , which depends on the (already existing) rospy package. [2]

4.5 rosrun

Since ROS has a large, distributed community, its software is organized into packages that are independently developed by community members. The concept of a ROS package will be described in greater detail in subsequent chapters, but a package can be thought of as a collection of resources that are built and distributed together. Pack- ages are just locations in the filesystem, and because ROS nodes are typically exe- cutable programs, one could manually cd around the filesystem to start all the ROS nodes of interest.[2]

For example, the talker program lives in a package named rospy_ tutorials , and its executable programs are found in /opt/ros/indigo/share/rospy_ tutorials. However, chasing down these long paths would become tiresome in large filesystems, since nodes can be deeply buried in large directory hierarchies. To automate this task, ROS provides a command-line utility called rosrun that will search a package for the requested program and pass it any parameters supplied on the command line. The syntax is as follows:

```
user@hostname$ rosrun PACKAGE EXECUTABLE [ARGS]
```

[2]

4.6 topics

As we saw in the previous chapter, ROS systems consist of a number of independent nodes that comprise a graph. These nodes by themselves are typically not very useful. Things only get interesting when nodes communicate with each other, exchanging information and data.[2] The most common way to do that is through topics. A topic is a name for a stream of messages with a defined type. For example, the data from a laser range-finder might be sent on a topic called scan , with a message type of LaserScan , while the data from a camera might be sent over a topic called image , with a message type of Image. Topics implement a publish/subscribe communication mechanism, one of the more common ways to exchange data in a distributed system. Before nodes start to transmit data over topics, they must first announce, or advertise, both the topic name and the types of messages that are going to be sent. Then they can start to send, or publish, the actual data on the topic.[2] Nodes that want to receive messages on a topic can sub- scribe to that topic by making a request to roscore . After subscribing, all messages on the topic are delivered to the node that made the request. One of the main advantages to using ROS is that all the messy details of setting up the necessary connections when nodes advertise or subscribe to topics is handled for you by the underlying communication mechanism so that you don't have to worry about it yourself.

4.6.1 Publishing to a topic

Here is a basic code for advertising a topic and publishing messages on it.

```
import rospy
from std_msgs.msg import Int32
rospy.init_node('topic_publisher')
pub = rospy.Publisher('counter', Int32)
rate = rospy.Rate(2)
count = 0
while not rospy.is_shutdown():
pub.publish(count)
count += 1
rate.sleep()
```

[2]

Code Explanation:

```
import rospy
```

appears in every ROS Python node and imports all of the basic functionality that we'll need. The next line imports the definition of the message that we're going to send over the topic:

```
from std_msgs.msg import Int32
```

In this case, we're going to use a 32-bit integer, defined in the ROS standard message package, std_ msgs . For the import to work as expected, we need to import from <package name>.msg , since this is where the package definitions are stored (more on this later). Since we're using a message from another package, we have to tell the ROS build system about this by adding a dependency to our package.xml file:

```
<depend package="std_msgs" />
```

Without this dependency, ROS will not know where to find the message definition, and the node will not be able to run. After initializing the node, we advertise it with a Publisher :

```
pub = rospy.Publisher('counter', Int32)
```

This gives the topic a name (counter) and specifies the type of message that will be sent over it (Int32). Behind the scenes, the publisher also sets up a connection to roscore and sends some information to it.[2] When another node tries to subscribe to the counter topic, roscore will share its list of publishers and subscribers, which the nodes will then use to create direct connections between all publishers and of all sub- scribers to each topic.[2] At this point, the topic is advertised and is available for other nodes to subscribe to. Now we can go about actually publishing messages over the topic:

```
rate = rospy.Rate(2)
count = 0
while not rospy.is_shutdown():
pub.publish(count)
count += 1
rate.sleep()
```

[2]

First, we set the rate, in hertz, at which we want to publish. For this example, we're going to publish twice a second. The is_ shutdown() function will return True if the node is ready to be shut down and False otherwise, so we can use this to determine if it is time to exit the while loop. Inside the while loop, we publish the current value of the counter, increment its value by 1, and then sleep for a while. The call to rate.sleep() will sleep for long enough to make sure that we run the body of the while loop at approximately 2 Hz.[2] And that's it. We now have a minimalist ROS node that advertises the counter topic and publishes integers on it. Checking that everything works as expected: Open a new terminal, and start up roscore . Once it's running, you can see what topics are available by running rostopic list in another terminal:

```
user@hostname$ rostopic list
/rosout
/rosout_agg
```

Now, run the node we've just looked at in yet another terminal. Make sure that the basics package is in a workspace, and you've sourced the setup file for that Workspace:

```
user@hostname$ rosrun basics topic_publisher.py
```

once the node is running, you can verify that the counter topic is advertised by running rostopic list again:

```
user@hostname$ rostopic list
/counter
/rosout
/rosout_agg
```

Even better, you can see the messages being published to the topic by running

```
user@hostname$ rostopic echo counter –n 5
```

The -n 5 flag tells rostopic to only print out five messages. Without it, it will happily go on printing messages forever, until you stop it with a Ctrl-C. [2]

4.6.2 subscribing to a topic

shows a minimalist node that subscribes to the counter topic and prints out the values in the messages as they arrive.

```
import rospy
from std_msgs.msg import Int32
def callback(msg):
print msg.data
rospy.init_node('topic_subscriber')
sub = rospy.Subscriber('counter', Int32, callback)
rospy.spin()
```

[2]

The first interesting part of this code is the callback that handles the messages as they come in:

```
def callback(msg):
print msg.data
```

ROS is an event-driven system, and it uses callback functions heavily.[2] Once a node has subscribed to a topic, every time a message arrives on it the associated callback

function is called, with the message as its parameter. In this case, the function simply prints out the data contained in the message.[2] After initializing the node, as before, we subscribe to the counter topic:

```
sub = rospy.Subscriber('counter', Int32, callback)
```

We give the name of the topic, the message type of the topic, and the name of the callback function. Behind the scenes, the subscriber passes this information on to roscore and tries to make a direct connection with the publishers of this topic. If the topic does not exist, or if the type is wrong, there are no error messages: the node will simply wait until messages start being published on the topic. Once the subscription is made, we give control over to ROS by calling rospy.spin() . This function will only return when the node is ready to shut down. This is just a useful shortcut to avoid having to define a top-level while loop like we did in the publisher.py.

Checking that everything works as expected First, make sure that the publisher node is still running and that it is still publishing messages on the counter topic.[2] Then, in another terminal, start up the subscriber node:

```
user@hostname$ rosrun basics topic_subscriber.py
```

It should start to print out integers published to the counter topic by the publisher Node. [2]

4.7 Robot and Simulators

The previous sections discussed many fundamental concepts of ROS. They may have seemed rather vague and abstract, but those concepts were necessary to describe how data moves around in ROS and how its software systems are organized. In this section, we will introduce common robot subsystems and describe how the ROS architec- ture handles them. Then, we will introduce the robot that we will use throughout the remainder of the book and describe the simulators in which we can most easily experiment with them.[2] there are a growing number of standard products that can be purchased and used "out of the box" for research, development, and operations in many domains of robotics. This section will describe one of these platforms, which will be used for examples throughout the rest of the book. [2]

4.7.1 turtleBot

The TurtleBot was designed in 2011 as a minimalist platform for ROS-based mobile robotics education and prototyping. It has a small differential-drive mobile base with an internal battery, power regulators, and charging contacts. Atop this base is a stack of laser-cut "shelves" that provide space to hold a netbook computer and depth cam- era, and lots of open space for prototyping. To control cost, the TurtleBot

relies on a depth camera for range sensing; it does not have a laser scanner. Despite this, map- ping and navigation can work quite well for indoor spaces. [2]

4.7.2 Stage

For many years, the two-dimensional simultaneous localization and mapping (SLAM) problem was one of the most heavily researched topics in the robotics community.[2] A number of 2D simulators were developed in response to the need for repeatable experiments, as well as the many practical annoyances of gathering long datasets of robots driving down endless office corridors. Canonical laser range-finders and differential-drive robots were modeled, often using simple kinematic models that enforce that, for example, the robot stays plastered to a 2D surface and its range sensors only interact with vertical walls, creating worlds that vaguely resemble that of Pac-Man. Although limited in scope, these 2D simulators are very fast computationally, and they are generally quite simple to interact with.[2] Stage is an excellent example of this type of 2D simulator. It has a relatively simple modeling language that allows the creation of planar worlds with simple types of objects. Stage was designed from the outset to support multiple robots simultane- ously interacting with the same world. It has been wrapped with a ROS integration package that accepts velocity commands from ROS and outputs an odometric transformation as well as the simulated laser range-finders from the robot(s) in the Simulation. [2]

4.7.3 Gazebo

Although Stage and other 2D simulators are computationally efficient and excel at simulating planar navigation in office-like environments, it is important to note that planar navigation is only one aspect of robotics. Even when only considering robot navigation, a vast array of environments require nonplanar motion, ranging from outdoor ground vehicles to aerial, underwater, and space robotics. Three- dimensional simulation is necessary for software development in these environments. Simulators often use rigid-body dynamics, in which all objects are assumed to be incompressible, as if the world were a giant pinball machine. This assumption drasti- cally improves the computational performance of the simulator, but often requires clever tricks to remain stable and realistic, since many rigid-body interactions become point contacts that do not accurately model the true physical phenomena. The art and science of managing the tension between computational performance and physical realism are highly nontrivial. There are many approaches to this trade-off, with many well suited to some domains but ill suited to others.[2] Like all simulators, Gazebo is the product of a variety of trade-offs in its design and implementation. Historically, Gazebo has used the Open Dynamics Engine for rigid-body physics, but recently it has gained the ability to choose between physics engines at startup. For the purposes of this book, we will be using Gazebo with either the Open Dynamics Engine or with the Bullet Physics library, both of which are capable of real-time simulation

with relatively simple worlds and robots and, with some care, can produce physically plausible behavior. There are many other simulators that can be used with ROS, such as MORSE and V- REP. Each simulator, whether it be Gazebo, Stage, MORSE, V-REP, turtlesim, or any other, has a different set of trade-offs. These include trade-offs in speed, accuracy, graphics quality, dimensionality (2D versus 3D), types of sensors supported, usability, platform support, and so on. No simulator of which we are aware is capable of maxi- mizing all of those attributes simultaneously, so the choice of the "right" simulator for a particular task will be dependent on many factors. [2]

4.8 Building Maps of the Environment

Now that you know how ROS works and have moved your robot around a bit, it's time to start looking at how to get it to navigate around the world on its own. In order to do this, the robot needs to know where it is, and where you want it to go to. Typically, this means that it needs to have a map of the world and to know where it is in this map. In this chapter, we're going to see how to build a high-quality map of the world, using data from your robot's sensors. If your robot had perfect sensors and knew exactly how it was moving, then building a map would be simple: you could take the objects detected by the sensors, transform them into some global coordinate frame (using the robot's position and some geome- try), and then record them in a map (in this global coordinate frame). Unfortunately, in the real world, it's not quite that easy. The robot doesn't know exactly how it's mov- ing, since it's interacting with an uncertain world. No sensor is perfect, and you'll have to deal with noisy measurements. How can you combine all this error-laden information together to produce a usable map? Luckily, ROS has a set of tools that will do this for you.[2] The tools are based on some quite advanced mathematics, but, luckily, you don't have to understand everything that's going on under the hood in order to use them. We'll describe these tools in this chapter, but first let's talk a bit about exactly what we mean by "map.".

4.8.1 Maps in ROS

Navigation maps in ROS are represented by a 2D grid, where each grid cell contains a value that corresponds to how likely it is to be occupied. Figure 4.1 shows an example of a map learned directly from the sensor data on a robot.[2] White is open space, black is occupied, and the grayish color is unknown.

Map files are stored as images, with a variety of common formats being supported (such as PNG, JPG, and PGM). Although color images can be used, they are converted to grayscale images before being interpreted by ROS. This means that maps can be displayed with any image display program. Associated with each map is a YAML file that holds additional information, such as the resolution (the length of

FIGURE 4.1: Map Example

each grid cell in meters), where the origin of the map is, and thresholds for deciding if a cell is occupied or unoccupied. Example map.yaml

```
image: map.pgm
resolution: 0.1
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.65
free_thresh: 0.196
negate: 1
```

[2]

This map is stored in the file map.png, has cells that represent 10 cm squares of the world, and has an origin at (0, 0, 0). A cell is considered to be occupied if the value in it is more than 65% of the total range allowed by the image format. A cell is unoccu- pied if it has a value less than 19.6% of the allowable range. This means that occupied cells will have a large value and will appear lighter in color in the image. Unoccupied cells will have a lower value and would appear darker. Since it is more intuitive for open space to be represented by white and occupied space by black, the negate flag allows for the values in the cells to be inverted before being used by ROS. So, for Example 5-1, if we assume that each cell holds a single unsigned byte (an integer from 0 to 255), each of the values will first be inverted by subtracting the original value from 255. Then, all cells with a value less that 49 (255 * 0.196 = 49.98) will be consid- ered free, and all those with a value greater than 165 (255 *

0.65 = 165.75) will be con- sidered to be occupied. All other cells will be classified as "unknown."[2] These classifications will be used by ROS when we try to plan a path through this map for the robot to follow. [2]

4.8.2 Recording Data with rosbag

rosbag is a tool that lets us record messages and replay them later. This is really useful when debugging new algorithms, since it lets you present the same data to the algo- rithm over and over, which will help you isolate and fix bugs.[2] It also allows you to develop algorithms without having to use a robot all the time. You can record some sensor data from the robot with rosbag , then use this recorded data to work on your code. rosbag can do more than record and play back data, but that's what we're going to focus on for now.[2] To record messages, we use the record functionality and a list of topic names. For example, to record all the messages sent over the scan and tf topics, you would run:

```
user@hostname$ rosbag record scan tf
```

This will save all of the messages in a file with a name in the format YYYY-MM-DD- HH-mm-ss.bag, corresponding to the time that rosbag was run. This should give each bag file a unique name, assuming you don't run rosbag more than once a second. You can change the name of the output file using the -O or –output-name flags, and add a prefix with the -o and –output-prefix flags. For example, these commands:

```
user@hostname$ rosbag record –O foo.bag scan tf
user@hostname$ rosbag record –o foo scan tf
```

would create bags named foo.bag and foo_ 2015-10-05-14-29-30.bag, respectively (obviously, with appropriate values for the current date and time).[2] We can also record all of the topics that are currently publishing with the -a flag:

```
user@hostname$ rosbag record –a
```

You can play back a pre-recorded bag file with the play functionality. There are a number of command-line parameters that allow you to manipulate how fast you play back the bag, where you start in the file, and other things (all of which are documented on the wiki), but the basic usage is straightforward:

```
user@hostname$ rosbag play ––clock foo.bag
```

This will replay the messages recorded in the bag file foo.bag, as if they were being generated live from ROS nodes.[2] Giving more than one bag file name will result in the bag files being played sequentially. The –clock flag causes rosbag to publish the clock time, which will be important when we come to build our maps.[2] You can find out information about a bag file with the info functionality:

```
user@hostname$ rosbag info laser.bag
```

[2]

4.8.3 Building Maps

Now we're going to look at how you can build a map like the one shown in Figure 5-1 using the tools in ROS. One thing to note about that map in Figure 5-1 is that it's quite "messy." Since it was created from sensor data taken from a robot, it includes some things you might not expect. Along the bottom edge of the map, the wall seems to have holes in it.[2] These are caused by bad sensor readings, possibly the result of clutter under the desks in those rooms. The strange blob in the largish room toward the top in the middle is a pool table. The gray spots in the larger room going diago- nally down and right are chair legs (this was a conference room). The walls are not always perfectly straight, and there are sometimes "unknown" areas in the middle of rooms if the sensors never made measurements there. When you start to make maps of your own with your robot, you should be prepared for them to look like this.[2] Gen- erally speaking, using more data to create the map will result in a better map. How- ever, no map will be perfect. Even though the maps might not look all that great to you, they're still perfectly useful to the robot, as we will see. You can build maps with the slam_ gmapping node from the gmapping package. The slam_ gmapping node uses an implementation of the GMapping algorithm, written by Giorgio Grisstti, Cyrill Stachniss, and Wolfram Burgard. GMapping uses a Rao-Blackwellized particle filter to keep track of the likely positions of the robot, based on its sensor data and the parts of the map that have already been built. If you're interes- ted in the details of the algorithm, they're described in these two papers:

Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard, "Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters," IEEE Transactions on Robotics 23 (2007): 34–46. Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard, "Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selec- tive Resampling," Proceedings of the IEEE International Conference on Robotics and Automation (2005): 2432–2437.

First, we're going to generate some data to build the map from. Although you can build a map using live sensor data, as the robot moves about the world, we're going to take another approach. We're going to drive the robot around and save the sensor data to a file using rosbag .[2] We're then going to replay this sensor data and use slam_ gmapping to build a map for us. Collecting data in a bag file is often a good idea when building a map, since it lets you play around with the parameters of the slam_ gmapping node to get a good map, without having to go and run the robot through the world again. This can be a real time-saver, especially if you need to tweak the mapping node parameters a lot.[2] First, let's record some data.Start up a simulator with a Turtlebot in it:

```
user@hostname$ roslaunch turtlebot_stage turtlebot_in_stage.launch
This launch file starts up the Stage robot simulator and an instance of rviz.
```

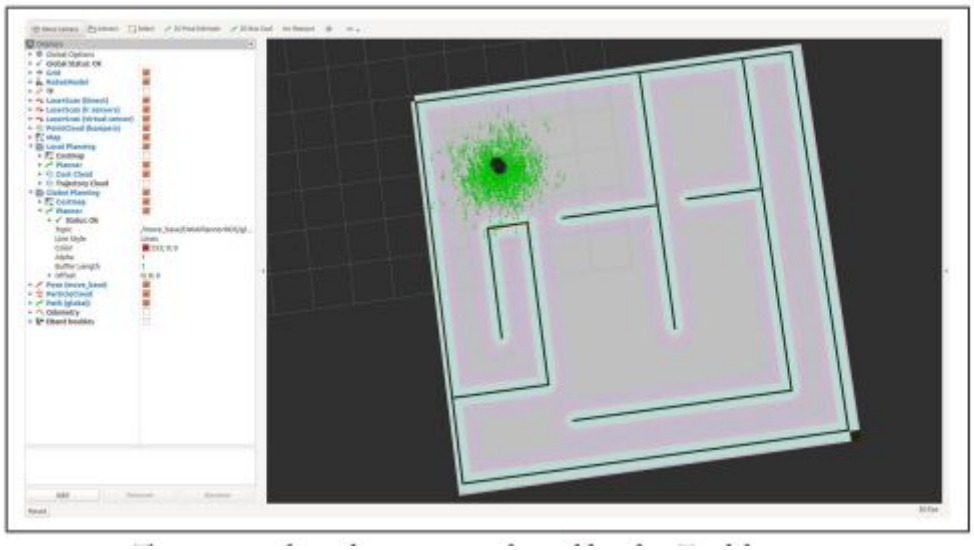


FIGURE 4.2: TurtleBot Example

Now, start up the the keyboard_ teleop node from the turtlebot_ teleop package:

```
user@hostname$ roslaunch turtlebot_teleop keyboard_teleop.launch
```

Practice driving the robot around for a bit. Once you've got the hang of it, we can get started collecting some data. slam_ gmapping builds maps from data from the laser range-finder and the odometry system, as reported by tf . Although the Turtlebot doesn't actually have a laser range-finder, it creates LaserScan messages from its Kin- ect data and sends them over the scan topic. With the simulator still running, in a new terminal window, start recording some data:

```
user@hostname$ rosbag record –O data.bag /scan /tf
```

Now, drive the robot around the world for a while. Try to cover as much of the map as possible, and make sure you visit the same locations a couple of times. Doing this will result in a better final map. If you get to the end of this section and your map doesn't look very good, try recording some new data and drive the robot around the simulated world for longer, or a bit more slowly. Once you've driven around for a while, use Ctrl-C to stop rosbag .[2] Verify that you have a data bag called data.bag. You can find out what's in this bag by using the ros bag info command:

```
user@hostname$ rosbag info data.bag
```

Once you have a bag that seems to have enough data in it, stop the simulator with a Ctrl-C in the terminal you ran roslaunch in. It's important to stop the simulator before starting the mapping process, because it will be publishing LaserScan mes-

sages that will conflict with those that are being replayed by rosbag . Now it's time to build a map. Start roscore in one of the terminals. In another terminal, we're going to tell ROS to use the timestamps recorded in the bag file, and start the slam_ gmapping node:

```
user@hostname$ rosparam set use_sim_time true
user@hostname$ rosrun gmapping slam_gmapping
```

We're going to use rosbag play to replay the data that we recorded from the simulated robot:

```
user@hostname$ rosbag play --clock data.bag
```

When it starts receiving data, slam_ gmapping should start printing out diagnostic information. Sit back and wait until rosbag finishes replaying the data and slam_ gmap ping has stopped printing diagnostics. At this point, your map has been built, but it hasn't been saved to disk.[2] Tell slam_ gmapping to do this by using the map_ saver node from the map_ server package. Without stopping slam_ gmapping , run the map_ saver node in another terminal:

```
user@hostname$ rosrun map_server map_saver
```

This will save two files to disk: map.pgm, which contains the map, and map.yaml, which contains the map metadata. [2]

4.9 Operating System Problem

We first Tried installing ROS on a raspbian operating System on the raspberry pi, After we have installed it correctly and started working on ROS we faced some problems, some packages was missing and not everything worked as expected.
We have tried fixing this problems, installing uninstalled packages, but we wasn't able to install all needed packages for our project and there was problem in some of them.

4.9.1 Problem Solution

This problem was solved by changing the operating system on the raspberry pi to ubuntu mate, after installing ROS on ubuntu mate everything worked fine and we became able to run all packages needed in our project.

Chapter 5

rplidar ROS Package

This package provides basic device handling for 2D Laser Scanner RPLIDAR A1/A2 and A3.

RPLIDAR is a low cost LIDAR sensor suitable for indoor robotic SLAM application. It provides 360 degree scan field, 5.5hz/10hz rotating frequency with guaranteed 8 meter ranger distance, current more than 16m for A2 and 25m for A3 . By means of the high speed image processing engine designed by RoboPeak, the whole cost are reduced greatly, RPLIDAR is the ideal sensor in cost sensitive areas like robots consumer and hardware hobbyist.

RPLIDAR A3 performs high speed distance measurement with more than 16K samples per second,RPLIDAR A2 performs high speed distance measurement with more than 4K/8K samples per second,RPLIDAR A1 supports 2K/4K samples per second. For a scan requires 360 samples per rotation, the 10hz scanning frequency can be achieved. Users can customized the scanning frequency from 2hz to 10hz freely by control the speed of the scanning motor. RPLIDAR will self-adapt the current scanning speed.

The driver publishes device-dependent sensor_ msgs/LaserScan data.[11]

5.1 rplidar Node

rplidarNode is a driver for RPLIDAR. It reads RPLIDAR raw scan result using RPLIDAR's SDK and convert to ROS LaserScan message.[11]

5.2 Published Topics

scan (sensor_ msgs/LaserScan) it publishes scan topic from the laser.[11]

5.3 Services

- stop_ motor (std_ srvs/Empty)
  Call the serive to stop the motor of rplidar.

- start_ motor (std_ srvs/Empty)
  Call the service to start the motor of rplidar.

5.4 Parameters

- serial_ port (string, default: /dev/ttyUSB0)
  serial port name used in your system.
- serial_ baudrate (int, default: 115200)
  serial port baud rate.
- frame_ id (string, default: laser)
  frame ID for the device.
- inverted (bool, default: false)
  indicated whether the LIDAR is mounted inverted.
- angle_ compensate (bool, default: true)
  indicated whether the driver needs do angle compensation.
- scan_ mode (string, default: std::string())
  the scan mode of lidar.

5.5 Practical Output

Figure 5.1 Shows the practical output of running rplidar package using RPLIDAR A2M4 and displaying output using rviz [13]

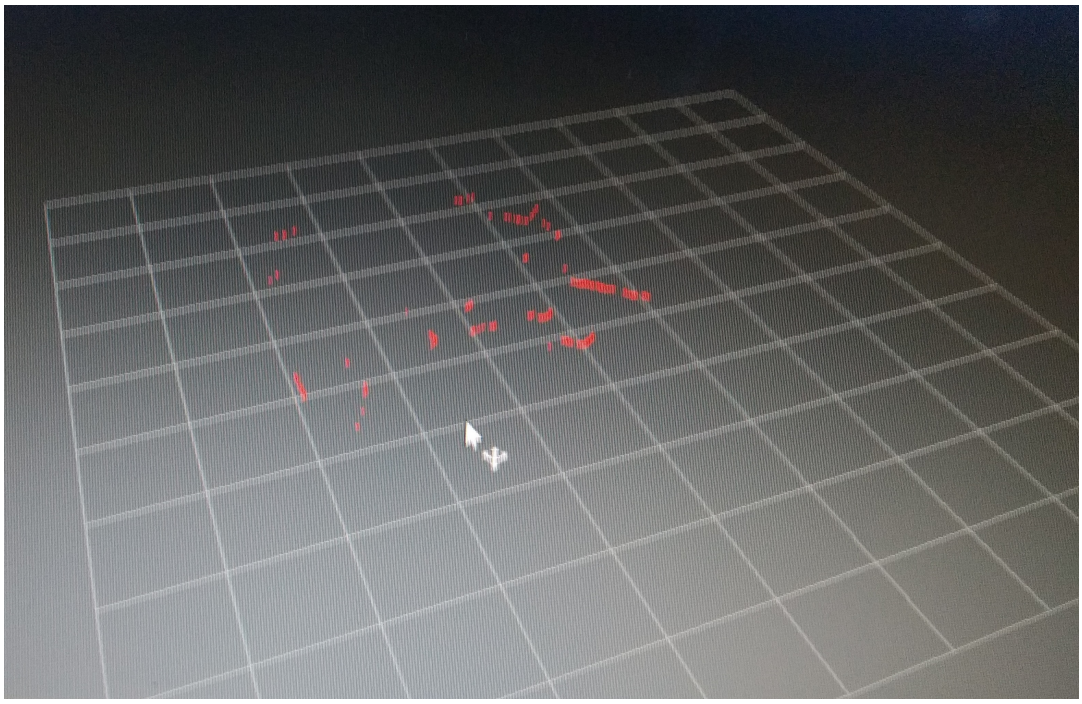


FIGURE 5.1: rplidar package output

Chapter 6

Hector SLAM

6.1 Mapping

Types of maps:

- Metric Map
- Topological Map
- Semantic Map

6.1.1 Metric Map

Metric map is map that A location is represented as a coordinate.
Figure 6.1 is an example of the metric map.

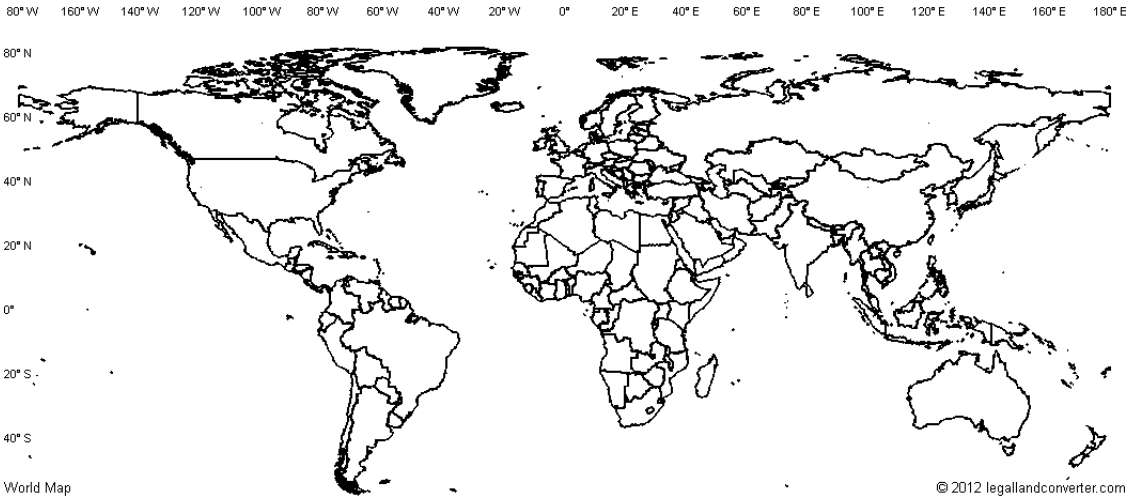


FIGURE 6.1: Example of the metric map

6.1.2 Topological Map

Topological Map is map that the Locations are represented as nodes and their connectivity as arcs.
Figure 6.2 shows an example of the topological map.

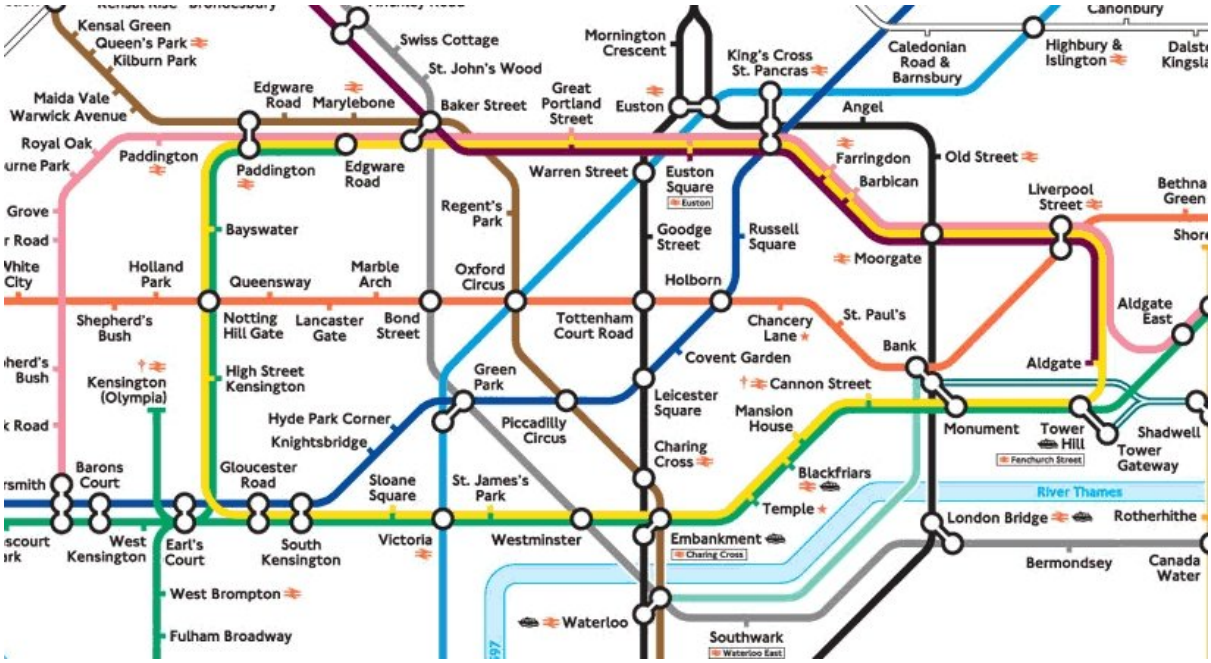


FIGURE 6.2: Example of the Topological Map

6.1.3 Semantic Map

Semantic Map is a map with labels. Figure 6.3 is an example of the semantic map.

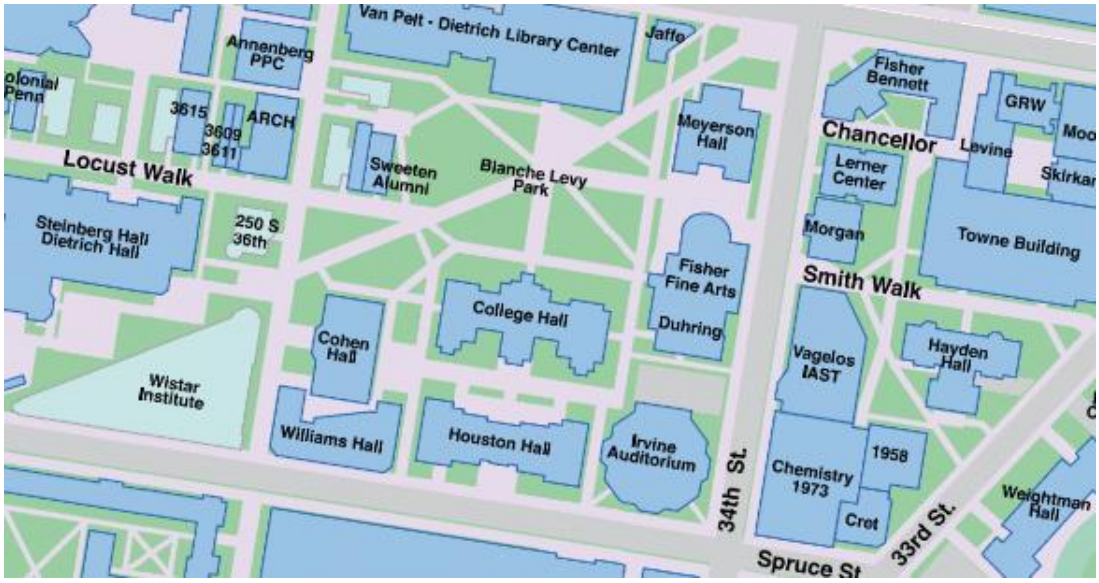


FIGURE 6.3: Example of the Semantic Map

6.2 Occupancy Grid Mapping

Occupancy Grid Mapping refers to a family of computer algorithms in probabilistic robotics for mobile robots which address the problem of generating maps from noisy and uncertain sensor measurement data, with the assumption that the robot pose is known. The basic idea of the occupancy grid is to represent a map of the environment as an evenly spaced field of binary random variables each representing the presence of an obstacle at that location in the environment. Occupancy grid algorithms compute approximate posterior estimates for these random variables.

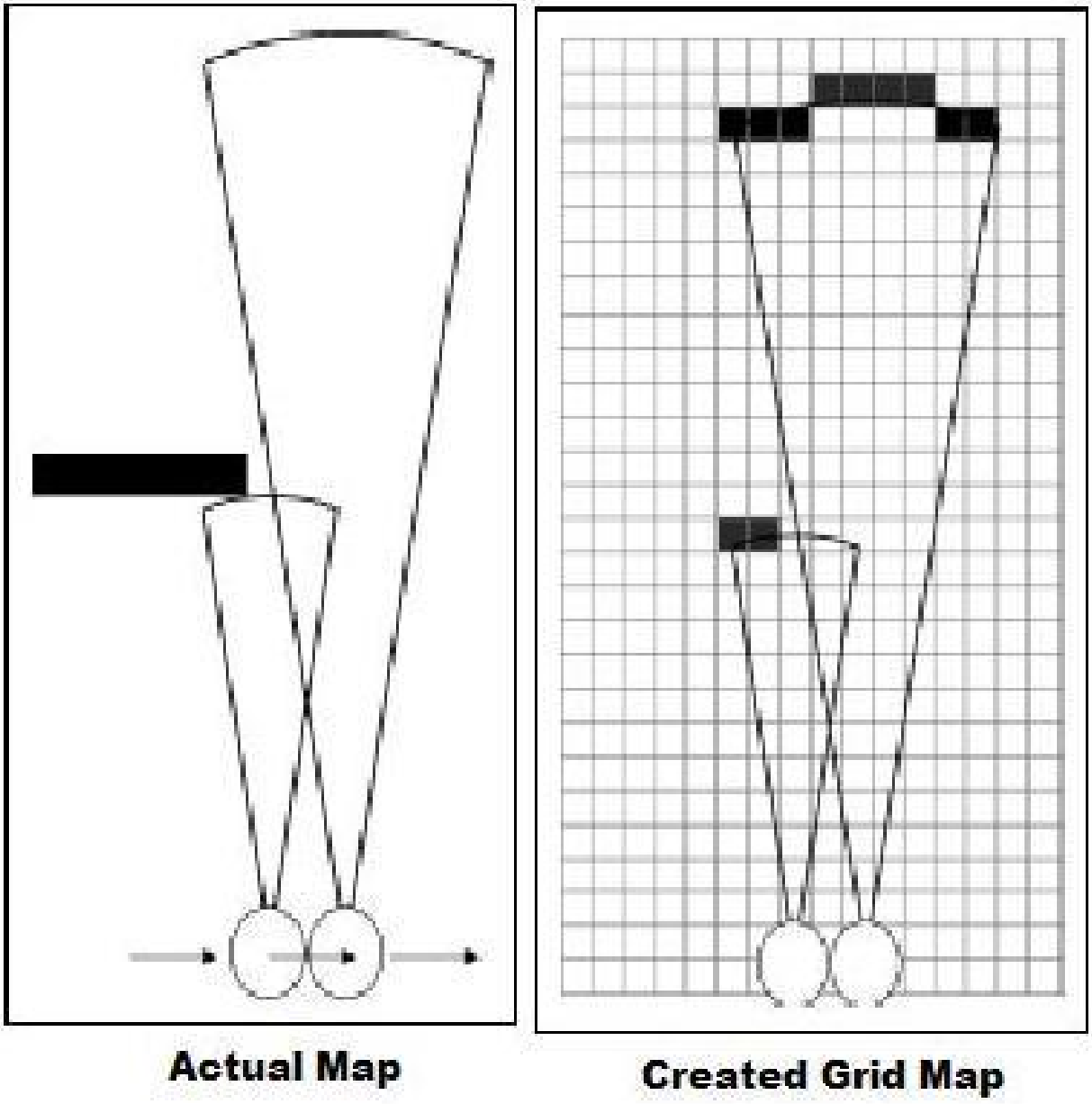


FIGURE 6.4: Occupancy Grid Map

The map we built in Figure 6.4 is an occupancy grid as you can see in the image above. The occupancy grid is nothing but a standard grid, the value of each cell

indicates if it is occupied, free or unexplored cell.

- Occupancy : binary R.V.
  M : { free , occupied } → { 0 , 1 }
- Measurement
  Z ∼ { 0 , 1 }
- Measurement Model
  p(z | m)

p(z = 1 | m = 1): True occupied measurement.
p(z = 0 | m = 1): False free measurement.
p(z = 1 | m = 0): False occupied measurement.
p(z = 0 | m = 0): True free measurement.

6.2.1 Log Odds

Log odds are an alternate way of expressing probabilities, which simplifies the process of updating them with new evidence. Unfortunately, it is difficult to convert between probability and log odds. The log odds is the log of the odds ratio. Thus, the log odds of A are:

$$log(p(A)/p(\ A))$$

And the log odds of A are:

$$log(p(\ A)/p(A))$$

Log_ occ = Log probability of occupied cells:

$$log(p(z=1|m=1)p(z=1|m=0))$$

Log_ free = Log probability of free cells:

$$log(p(z=0|m=0)p(z=0|m=1))$$

[4]

6.2.2 Log-odd update

Initial Map:

$$log\_odd = 0, \forall(x,y)$$

$$p(m=1) = p(m=0) = 0.5$$

.[4]

- Cells with $z = 1$
  $Log\_odd = log\_odd + log\_acc$
- Cells with z = 0
  $Log\_odd = log\_odd - log\_free$ [4]

6.3 Hector SLAM Algorithm

Hector SLAM is an open source implementation of the 2D SLAM technique. The technique is based on using a laser scan to build a grid map of the surroundings. In comparison to the majority of grid-map SLAM techniques, Hector SLAM does not require wheel odometry information. Thus, the 2D robot pose is estimated based on the scan matching process alone. The high update rate and accuracy of the modern LIDAR has been leveraged for the scan matching process thus allowing fast and accurate pose estimates. To be able to represent arbitrary environments an occupancy grid map is used, which is a proven approach for mobile robot localization using LIDARs in real-world environments.[6] As the LIDAR platform might exhibit 6DOF motion, the scan has to be transformed into a local stabilized coordinate frame using the estimated attitude of the LIDAR system. Using the estimated platform orientation and joint values, the scan isconverted into a point cloud of scan endpoints. Depending on the scenario, this point cloud can be preprocessed, for example by downsampling the number of points or removal of outliers. For the presented approach, only filtering based on the endpoint z coordinate is used, so that only endpoints within a threshold of the intended scan plane are used in the scan matching process. [6]

6.3.1 Map Access

The discrete nature of occupancy grid maps limits the precision that can be achieved and also does not allow the direct computation of interpolated values or derivatives.[5] For this reason an interpolation scheme allowing sub-grid cell accuracy through bilinear filtering is employed for both estimating occupancy probabilities and derivatives. Intuitively, thegrid map cell values can be viewed as samples of an underlying continuous probability distribution. Given a continuous map coordinate P_m , the occupancy value $M(Pm)$ as well as the gradient $\nabla M(P_m)$ can be approximated by using the four closest integer coordinates $P_{00\cdots11}$.

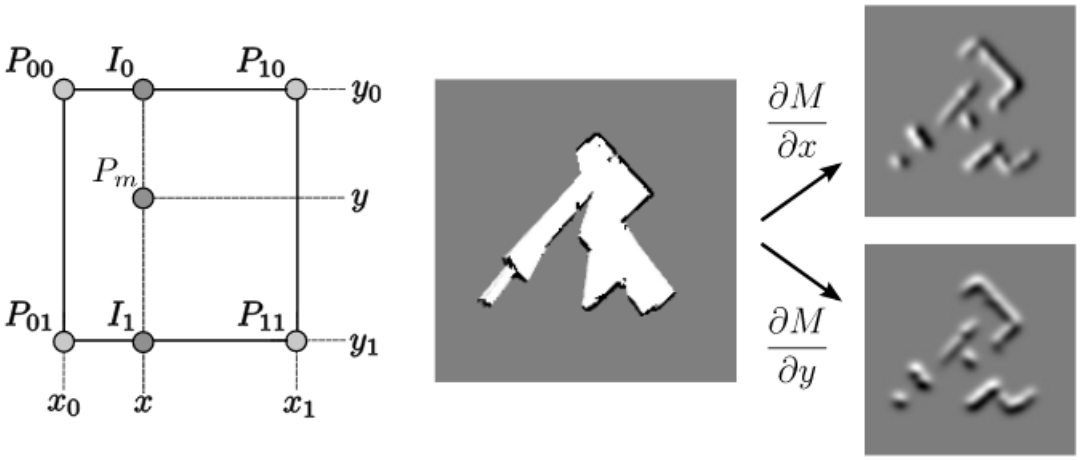
Linear interpolation along the x- and y-axis then yields

$$M(P_m) \approx \frac{y - y_0}{y_1 - y_0} \left(\frac{x - x_0}{x_1 - x_0} M(P_{11}) + \frac{x_1 - x}{x_1 - x_0} M(P_{01}) \right) + \frac{y_1 - y}{y_1 - y_0} \left(\frac{x - x_0}{x_1 - x_0} M(P_{10}) + \frac{x_1 - x}{x_1 - x_0} M(P_{00}) \right)$$

The derivatives can be approximated by:

$$\begin{aligned} \frac{\partial M}{\partial x}(P_m) \approx & \frac{y - y_0}{y_1 - y_0}(M(P_{11}) - M(P_{01})) \\ & + \frac{y_1 - y}{y_1 - y_0}(M(P_{10}) - M(P_{00})) \\ \frac{\partial M}{\partial y}(P_m) \approx & \frac{x - x_0}{x_1 - x_0}(M(P_{11}) - M(P_{10})) \\ & + \frac{x_1 - x}{x_1 - x_0}(M(P_{01}) - M(P_{00})) \end{aligned}$$

[5]



[5]

It should be noted that the sample points/grid cells of the map are situated on a regular grid with distance 1 (in map coordinates) from each other, which simplifies the presented equations for the gradient approximation. [5]

6.3.2 Scan Matching

Scan matching is the process of aligning laser scans with each other or with an existing map. Modern laser scanners have low distance measurement noise and high scan rates. A method for registering scans might yield very accurate results for this reason. For many robot systems the accuracy and precision of the laser scanner is much higher than that of odometry data, if available at all. Our approach is based on optimization of the alignment of beam endpoints with the map learnt so far. The basic idea using a Gauss-Newton approach is inspired by work in computer vision. Using this approach, there is no need for a data association search between beam endpoints or an exhaustive pose search. As scans get aligned with the existing map, the matching is implicitly performed with all preceding scans. We seek to find the rigid transformation $\zeta = (\rho_x \rho_y \Phi)^T$ that minimizes

$$\boldsymbol{\xi}^* = \underset{\boldsymbol{\xi}}{\operatorname{argmin}} \sum_{i=1}^{n} \left[1 - M(\mathbf{S}_i(\boldsymbol{\xi}))\right]^2$$

that is, we want to find the transformation that gives the best alignment of the laser scan with the map. Here,$S_i(\zeta)$ are the world coordinates of scan endpoint. The function $M(S_i(\zeta))$ returns the map value at the coor-dinates given by $S_i(\zeta)$. Given some starting estimate of ζ, we want to estimate $\nabla\zeta$ which optimizes the error measure according to

$$\sum_{i=1}^{n} \left[1 - M(\mathbf{S}_i(\boldsymbol{\xi} + \Delta\boldsymbol{\xi}))\right]^2 \rightarrow 0\,.$$

By first order Taylor expansion of $M(S_i(\zeta + \nabla\zeta))$ we get:

$$\sum_{i=1}^{n} \left[1 - M(\mathbf{S}_i(\boldsymbol{\xi})) - \nabla M(\mathbf{S}_i(\boldsymbol{\xi})) \frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}} \Delta\boldsymbol{\xi})\right]^2 \rightarrow 0\,.$$

This equation is minimized by setting the partial derivative with respect to $\nabla\zeta$ to zero:

$$2\sum_{i=1}^{n}\left[\nabla M(\mathbf{S}_i(\boldsymbol{\xi}))\frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}}\right]^T \left[1 - M(\mathbf{S}_i(\boldsymbol{\xi})) - \nabla M(\mathbf{S}_i(\boldsymbol{\xi}))\frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}}\Delta\boldsymbol{\xi})\right] = 0$$

[5]

Solving for $\nabla\zeta$ yields the Gauss-Newton equation for the minimization problem: [5]

$$\Delta\boldsymbol{\xi} = \mathbf{H}^{-1}\sum_{i=1}^{n}\left[\nabla M(\mathbf{S}_i(\boldsymbol{\xi}))\frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}}\right]^T [1 - M(\mathbf{S}_i(\boldsymbol{\xi}))]$$

$$\mathbf{H} = \left[\nabla M(\mathbf{S}_i(\boldsymbol{\xi}))\frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}}\right]^T \left[\nabla M(\mathbf{S}_i(\boldsymbol{\xi}))\frac{\partial \mathbf{S}_i(\boldsymbol{\xi})}{\partial \boldsymbol{\xi}}\right]$$

[5]

6.3.3 Hector One Iteration

- Receive scan from LIDAR
- Transform scan endpoints into "/base_stabilized" frame
- Throw out endpoints outside cut-offs:
  - Laser_ z_ min_ value
  - Laser_ z_ max_ value
  - Laser_ min_ dist
  - Laser_ max_ dist
- Perform 2D pose estimation
  - Set pose estimate, either:
    - Pose from preceding iteration
    - Pose using a tf start estimate
  - Iterate:
    - Project endpoints onto map based on current pose estimate

    - Estimate map occupancy probability gradients at scan endpoints
    - Perform Gauss-Newton iteration to refine pose estimate
- Update map if robot is estimated to have travelled more than thresholds indicated by
    - Map_ update_ distance_ thresh
    - SMap_ update_ angle_ thresh

6.3.4 Multi-Resolution Map Representation

- Gradient-based Optimization can get stuck in local minima
- Solution: Use multi-level map representation
- Every level updated separately at map update step using scan data

6.4 Hector Mapping Package

hector_ mapping is a SLAM approach that can be used without odometry as well as on platforms that exhibit roll/pitch motion (of the sensor, the platform or both). It leverages the high update rate of modern LIDAR systems like the Hokuyo UTM-30LX and provides 2D pose estimates at scan rate of the sensors (40Hz for the UTM-30LX). While the system does not provide explicit loop closing ability, it is sufficiently accurate for many real world scenarios. The system has successfully been used on Unmanned Ground Robots, Unmanned Surface Vehicles, Handheld Mapping Devices and logged data from quadrotor UAVs. [7]

6.4.1 Hardware Requirements

To use hector_ mapping, you need a source of sensor_ msgs/LaserScan data (for example a Hokuyo UTM-30LX LIDAR or bagfiles or rplidar). The node uses tf for transformation of scan data, so the LIDAR does not have to be fixed related to the specified base frame. Odometry data is not needed. [7]

6.4.2 Parameters

- Map_ resolution → This is the length of a grid cell edge.
- Map_ size → The size [number of cells per axis] of the map. The map is square and has (map_ size * map_ size) grid cells.
- Map_ update_ distance_ thresh → minimum distance to be travelled before having a map update.
- Map_ update_ angle_ thresh → minimum angle to be travelled before a map update.

- Update_ factor_ free → The map update modifier for updates of free cells in the range [0.0, 1.0]. A value of 0.5 means no change.
- Update_ factor_ occupied → The map update modifier for updates of occupied cells in the range [0.0, 1.0]. A value of 0.5 means no change.
- Laser_ min_ dist → The minimum distance [m] for laser scan endpoints to be used by the system. Scan endpoints closer than this value are ignored.
- Laser_ max_ dist → The maximum distance [m] for laser scan endpoints to be used by the system. Scan endpoints farther away than this value are ignored.
- Laser_ z_ min_ value → The minimum height [m] relative to the laser scanner frame for laser scan endpoints to be used by the system. Scan endpoints lower than this value are ignored.
- Laser_ z_ max_ value → The maximum height [m] relative to the laser scanner frame for laser scan endpoints to be used by the system. Scan endpoints higher than this value are ignored.

6.5 Differences between Fast SLAM and Hector SLAM

The map_ update_ distance_ threshold and map_ update_ angle_ thresh parameters indicate the required distance or angle that the platform has to travel before a map update process is executed. The default distance and angle are 0.4 m and 0.9 rad, respectively.

Figure 6.5 shows Corridor maps obtained using Hector SLAM technique with different combinations of Distance and Angle Threshold values.

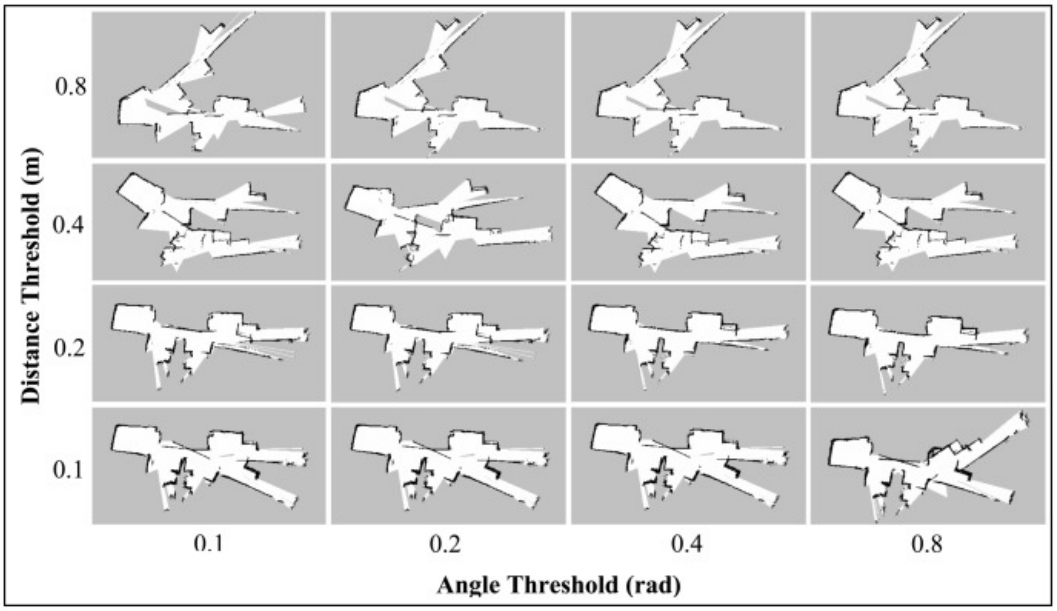


FIGURE 6.5: Corridor maps using Hector SLAM technique with different Threshold values

Figure 6.6 shows room maps obtained using Hector SLAM technique with different combinations of Distance and Angle Threshold values.

in both Corridor and Room maps in Figure 6.5 and in Figure 6.6, the effect of the angle threshold parameter was seen to be minimal.[8] Only slight differences could be observed when the angular parameter is changed while keeping the distance threshold constant. On the other hand, the maps for both corridor and room are seen to be more accurate at lower distance threshold values (i.e., 0.1 m and 0.2 m). [8]

6.6 Gmapping Linear and Angular Update

The linearUpdate and angularUpdate are the parameters that correspond to the required translation and rotation of robot before a scan is processed. The default values are 1.0 m for linear update and 0.5 rad for angular update.

Figure 6.7 Shows corridor maps obtained using Gmapping technique with different combinations of Linear and Angular Update values.

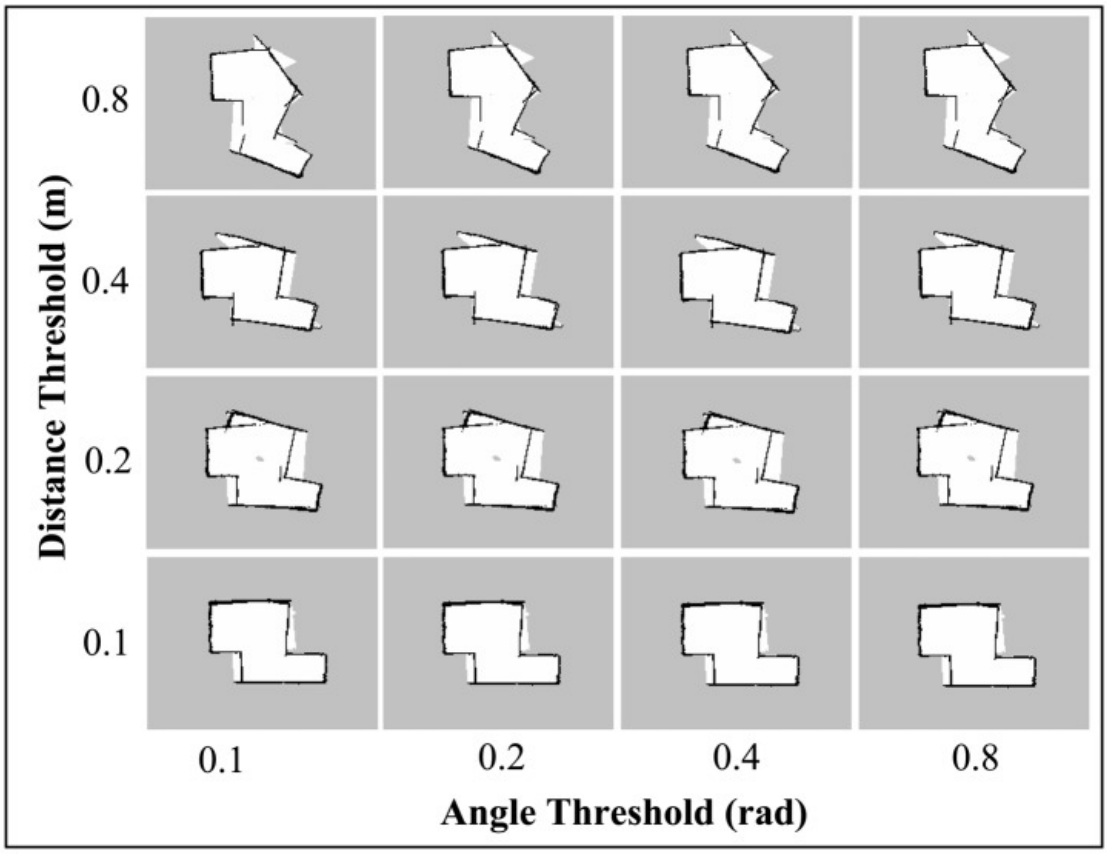


FIGURE 6.6: Room maps using Hector SLAM technique with different Threshold values

Figure 6.8 Shows room maps obtained using Gmapping technique with different combinations of Linear and Angular Update values.

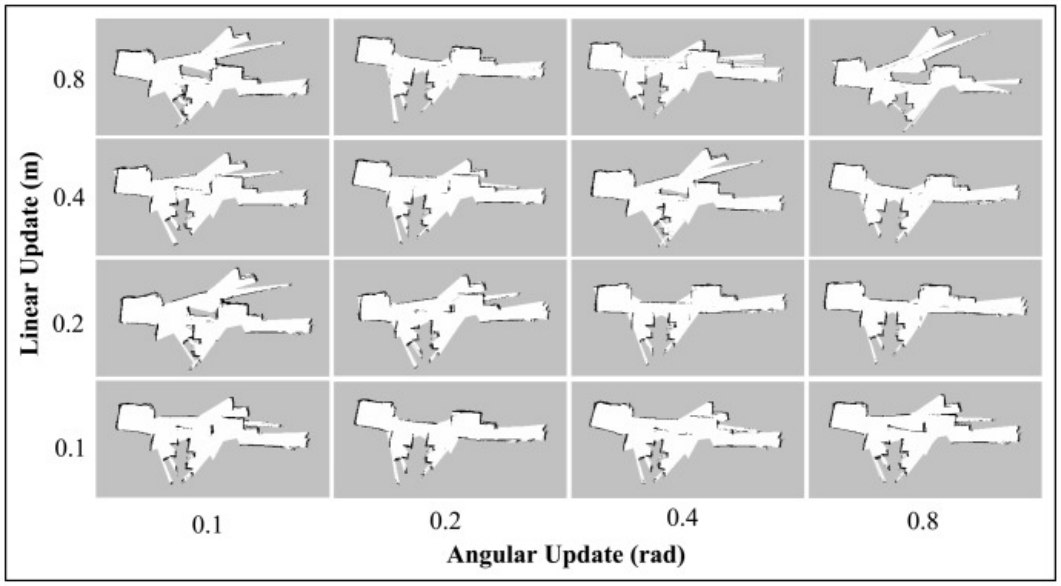


FIGURE 6.7: Corridor maps using Gmapping technique with different Threshold values

As can be seen from the corridor maps, no trend could be inferred for the different linear and angular update values. However, for Room maps, it was found that the higher linear update values improved the convergence to the real map.

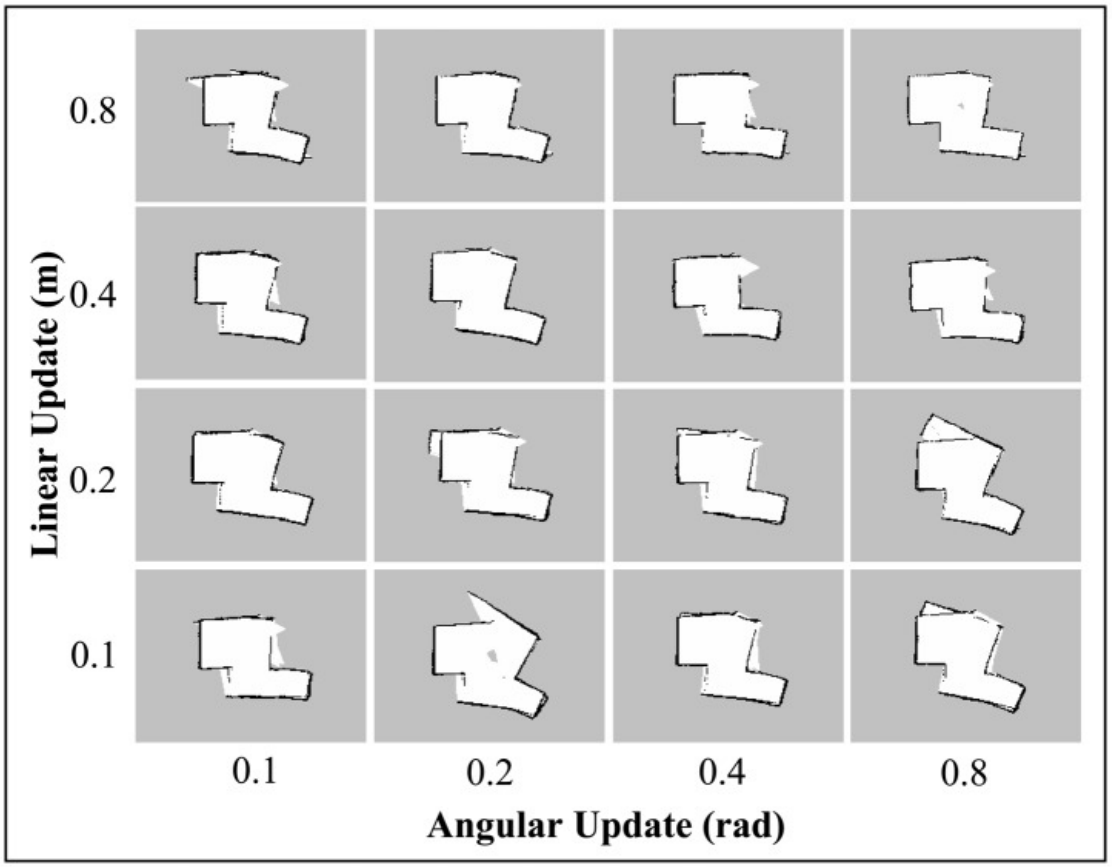


FIGURE 6.8: Room maps using Gmapping technique with different Threshold values

6.7 Practical Output

In this section we will show a sample of the maps we had by using the hector SLAM algorithm.

Our robot has two modes, it can be remotely controlled using a laptop connected the raspberry pi or it explore autonomously by using the obstacle avoidance algorithm, using this algorithm with the hector SLAM algorithm allow the robot to explore the place without human interaction as the robot moves forward and avoid obstacles by turning left which allows the robot to explore unexplored parts of the place.

In this section we will show a sample of the output using both methods.

6.7.1 Moving from room to another

In This section will show the map produced from moving the robot from to another room.

Figure 6.9 shows the Robot Moving in the first Room.

FIGURE 6.9: The robot moving in the first room

Figure 6.10 shows the robot entering the second room

FIGURE 6.10: The robot entered the second room

Figure 6.11 Shows the generated map after the robot directly visited the second room.

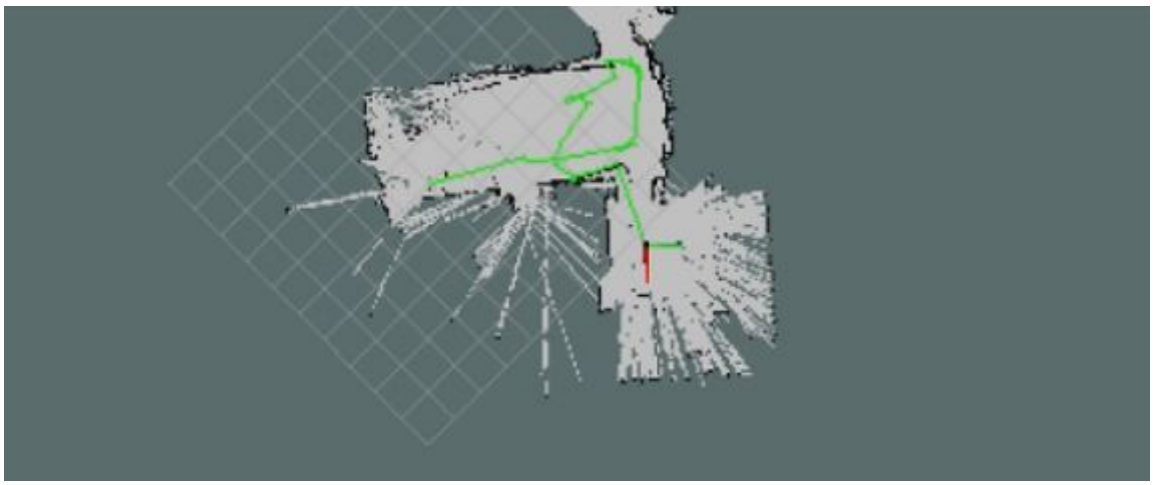


FIGURE 6.11: The generated map after visiting the next room

6.7.2 Moving Forward and Returning

Figure 6.12 Shows the robot moving in corridor and return back to its position, and the robot trajectory returned back to its starting position which means that robot localization is accurate.

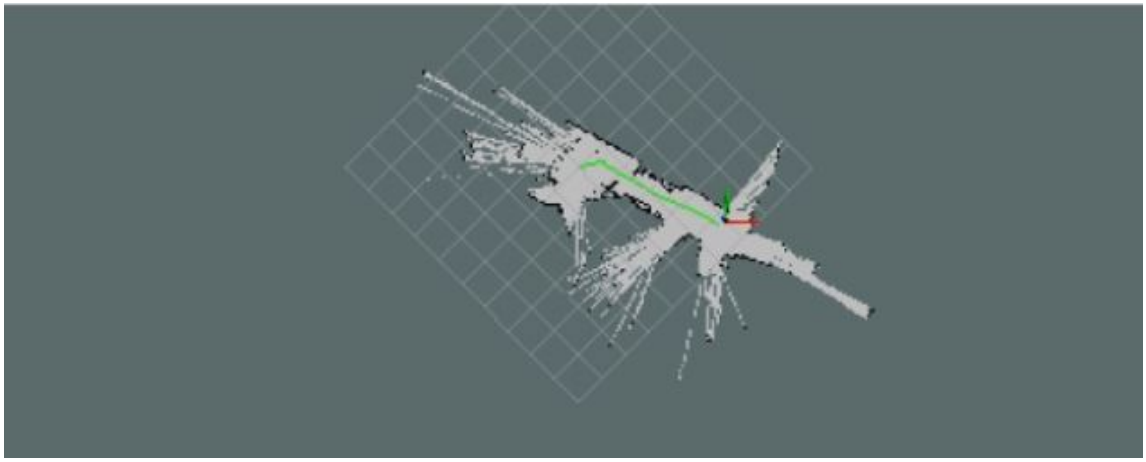


FIGURE 6.12: Moving in Corridor

6.7.3 Loop Exploration

In Figure 6.13 we managed to make the robot move in a loop in order to explored previously explored place from another position and the robot trajectory in Figure 6.13 shows that the robot was able to recognize that it was the previously explored place and determine that it returned to its original position.

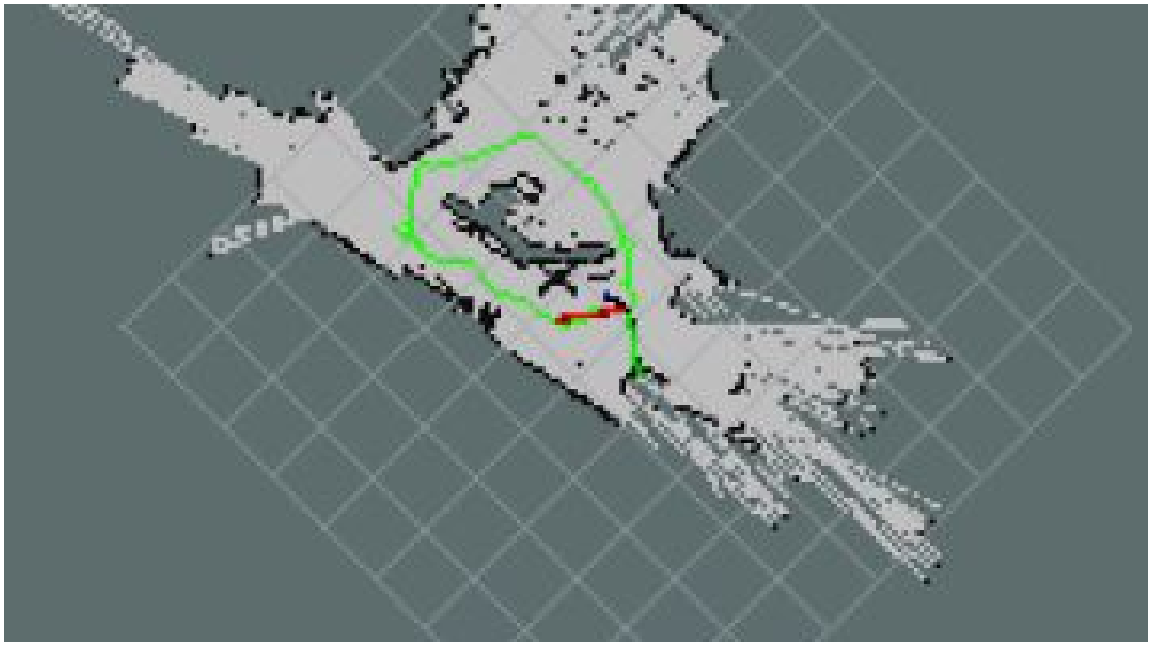


FIGURE 6.13: Loop exploration

6.7.4 Lectures Hall

Figure 6.14 shows the map of the front part of room 344 in the old building in the Faculty of Engineering Ain Shams university.

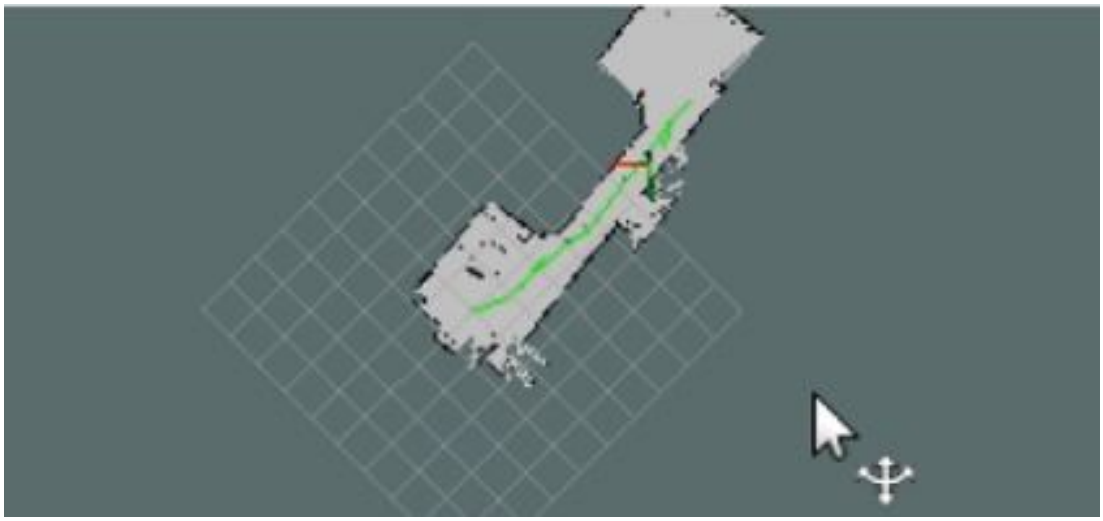


FIGURE 6.14: Room 344

6.7.5 Two Corridors

In This Section we will show using obstacle avoidance algorithm to make the robot move forward in a corridor and rotates left when it reaches the end of this corridor to enter another corridor and reach the room at the end of this corridor.

Figure 6.15 Shows the robot moving through the first corridor.

FIGURE 6.15: Robot moving through a corridor

Figure 6.16 Shows the map of the two corridors the robot explored using obstacle avoidance algorithm without human interaction.

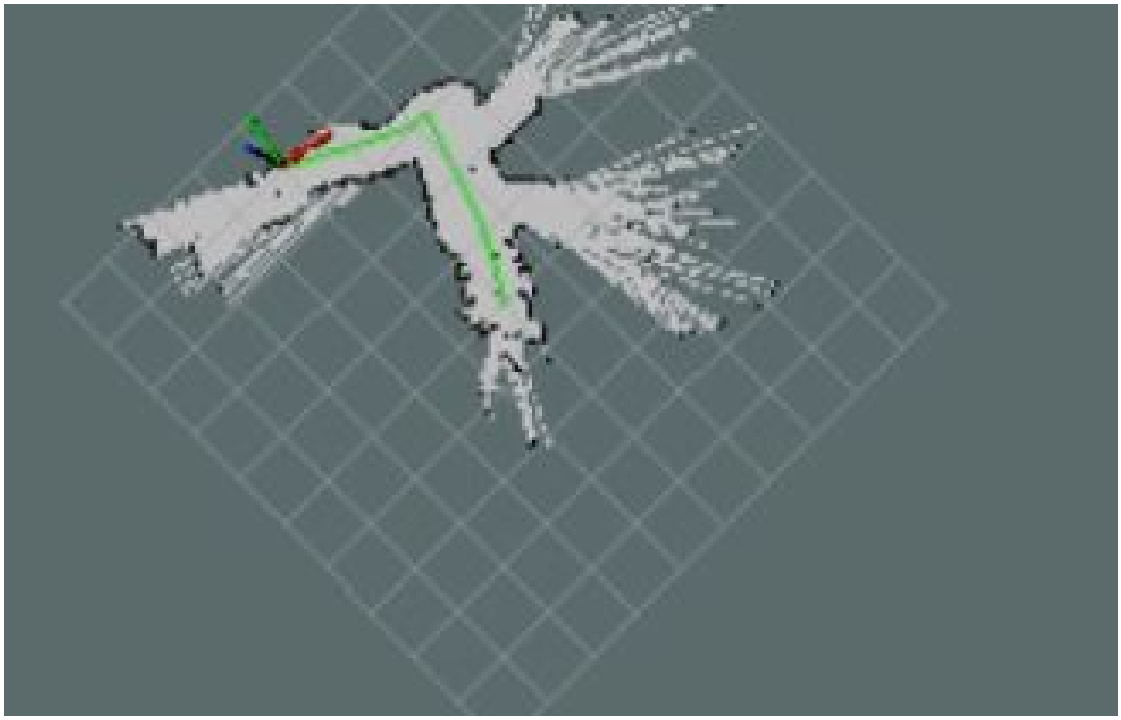


FIGURE 6.16: Two Corridors map

6.7.6 Part of a Floor in the university

Figure 6.17 shows a part of the first floor of the new programs building at the Faculty of Engineering Ain Shams university, this part was explored using obstacle avoidance algorithm, the robot moved without human to interaction to explore this part.

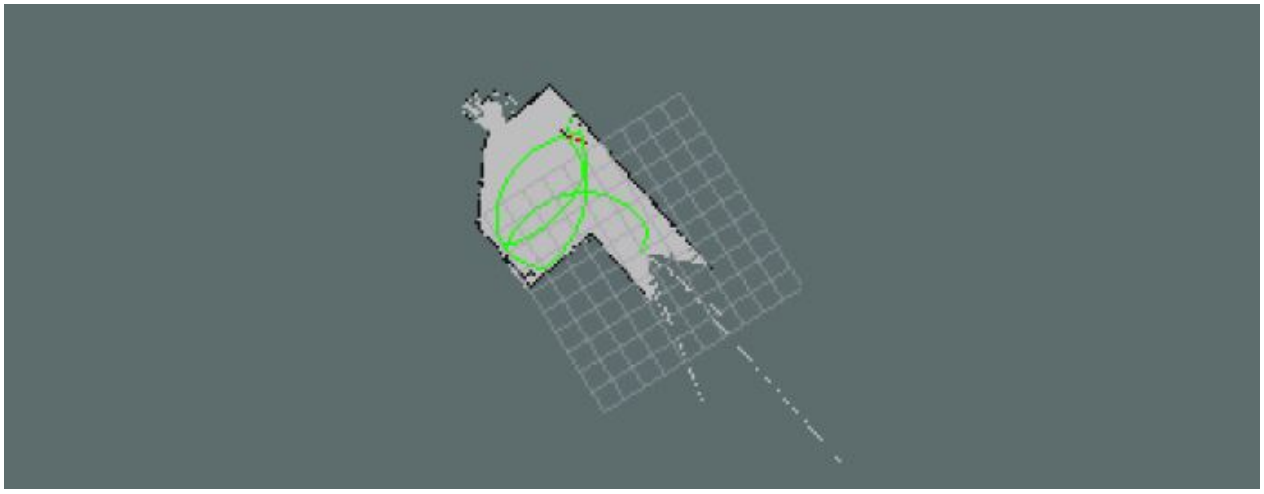


FIGURE 6.17: New Programs building First Floor

6.7.7 Corridor of a Floor in the university

Figure 6.18 shows part of the corridor of the first floor of the new programs building in the Faculty of Engineering Ain Shams University, obstacle avoidance algorithm is used to make the robot explore this part without human interaction.

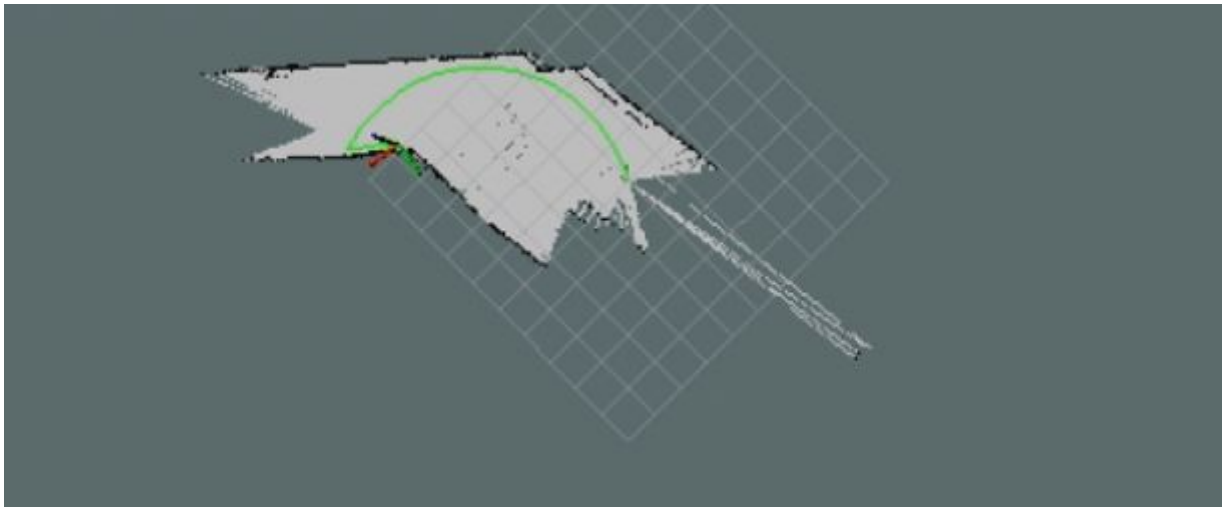


FIGURE 6.18: New Programs building First Floor Corridor

Chapter 7

Obstacle Avoidance

During the early stage of the project our main target was the accomplishment of the main objective of the project, reaching satisfying output and make sure everything is working correctly and as expected.
After long time of searching, trying, testing and after several trial we have found that the SLAM algorithm is working fine and as expected, we have used the hector slam algorithm that allow us to building a full map of place we are exploring and having an accurately localized robot that knows its location with respect to the environment and any change in its position is accurately detect and can easily detect its new position with drawing a full trajectory on the map of his motion from place to another with a localization similar to its position in the real world.

After making sure that everything in the main algorithm is working fine and as expected we started trying to add more features to the robot, we started to think of adding some intelligence to the robot and what is more important is to add feature useful for SLAM and to make the robot performing its main objective more elegant and in an intelligent way.
After searching and trying the alternatives we can implement we finally decided that the best choice is implementing obstacle avoidance algorithm that can allow the robot to navigate through the environment without human interaction.

7.1 Main Idea

The main idea behind implementing this algorithm is that this algorithm allow the robot to move autonomously without hitting something and this may allow it explore the environment without human interaction as it allows it keep moving forward and rotate in a certain position, left or right when it face an obstacle.
By this was it may allow the robot to explore the whole environment after certain amount of time because it keeps moving forward and turn in certain position on facing an obstacle which allow it explore unexplored as he has come to an end for the way he is moving in.

7.2 Using Lidar

The best option to implement obstacle avoidance is to implement it using the Lidar as the Lidar has high accuracy, high precision, long range and rotates.

Using Lidar as a feed for the obstacle avoidance algorithm guarantees high accuracy, also guarantees that everything will work as expected and allows the robot to explore any new space without human interaction.

7.2.1 Lidar Limited Time

After we have made a decision to use the Lidar for the implementation of the obstacle avoidance and after we found that the Lidar is the range finder that will make everything work correctly we found that using it is not applicable.

Using Lidar in implementing obstacle avoidance algorithm is not applicable because we use the RPlidar we took from Ihub which we don't have all the time and we can use it for a limited time only because another team also use it.
We were able to use the RPlidar for a few days only and these days are the days that we managed to make the hector slam algorithm work correctly and making a demo video for the slam algorithm working using Lidar.

We conclude that although the best thing to use for obstacle avoidance algorithm is the Lidar, the Lidar is not option as it is not available and so we have to search for another option to implement the obstacle avoidance algorithm.

7.3 Using Ultrasonic Sensor

Form the previous section we have came to a conclusion that using the Lidar for implementing obstacle avoidance algorithm is not an option and we have to search for another alternative and after some search for the range-finders we have decided to use ultrasonic.

Ultrasonic is not accurate, has a limited range, doesn't rotate and it is not the best option by any means but it is our only option under the constraint that we use the Lidar only for a limited time at that point we found that we have to use ultrasonic sensor to implement obstacle avoidance algorithm to allow the robot to navigate without human interaction.

7.4 Pseudocode

Here is simple pseudocode for the implemented obstacles avoidance algorithm and it can clearly explains how it works

```
Start

        while True

                robot_move_forward()
                distance = ultrasonic()

                if distance<offset

                        robot_turn_left()
```

End

This pseudocode simply explains the implemented obstacle avoidance algorithm.

In this implementation the robot simply keeps moves forward until the ultrasonic sensor detects an obstacle the robot turns left and keeps moving and this way allows the robot to keep exploring the environment without human interaction.

7.5 Source Code

In this section we will illustrate a part of the source code of the implemented obstacle avoidance algorithm.

7.5.1 Ultrasonic Function

This show the implemented ultrasonic function that is used in the obstacle avoidance implementation.

```
#Ultrasonic Function
def ultrasonic_echo(Trig, Echo):

    GPIO.output(Trig, True)
    time.sleep(0.00001)
    GPIO.output(Trig, False)
```

```
    while GPIO.input (Echo) == 0:
        StartTime = time.time()

    while GPIO.input (Echo) == 1:
        StopTime = time.time()


    TimeElapsed = StopTime -StartTime

    distance = (TimeElapsed * 34300)/2

    return distance
#End Ultrasonic Function
```

7.5.2 Implementation

In this section we will use the previously implemented ultrasonic function to implement the obstacle avoidance algorithm.

```
        dist = ultrasonic_echo (Trig_M, Echo_M)

        if dist>40:
            Motor_A_Forward(100)
            Motor_B_Forward(100)


        else:
            Motor_A_Stop()
            Motor_B_Stop()

            time.sleep(0.5)
            Motor_A_Forward(80)
            Motor_B_Backward(80)

            time.sleep(0.9)
```

Appendix A

Sensors

A.1 Ultrasonic Sensor

FIGURE A.1: HC-SR04 Ultrasonic Module

A.1.1 Specifications and Description

TABLE A.1: HC-SR04 Specifications

Parameters	**Min**	**Typ.**	**Max**	**Unit**
Operating Voltage	4.50	5.0	5.5	V
Quiescent Current	1.5	2	2.5	mA
Working Current	10	15	20	mA
Ultrasonic Frequency	-	40	-	KHz
Measuring Angle	-	15	-	Degree

HC-SR04 Ultrasonic module provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm. The modules includes ultrasonic transmitters, receiver and control circuit.

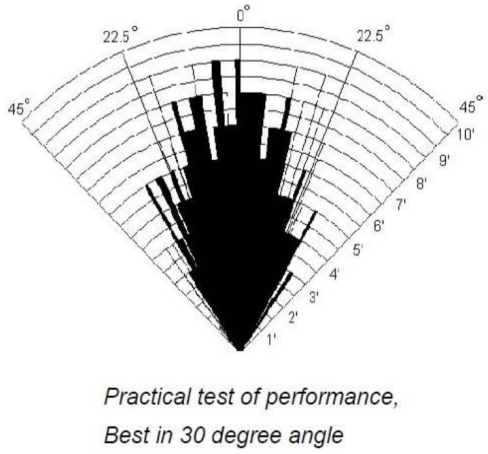


FIGURE A.2: HC-SR04 Test of performance

A.1.2 Theory of Operation

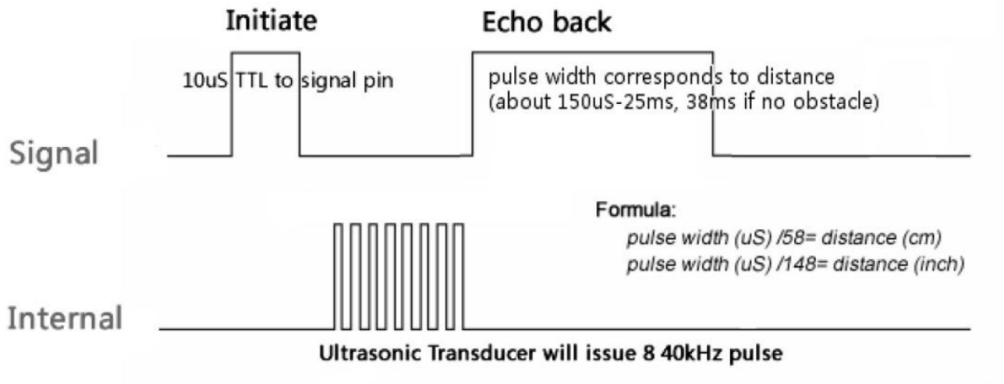


FIGURE A.3: HC-SR04 Timing Diagram

A short ultrasonic pulse is transmitted at the time 0, reflected by an object. The senor receives this signal and converts it to an electric signal. The next pulse can be transmitted when the echo is faded away. This time period is called cycle period. The recommend cycle period should be no less than 50ms. If a 10μs width trigger pulse is sent to the signal pin, the Ultrasonic module will output eight 40kHz ultrasonic signal and detect the echo back. The measured distance is proportional to the echo pulse width and can be calculated by the formula above. If no obstacle is detected, the output pin will give a 38ms high level signal.

To obtain the distance, measure the width (Ton) of Echo pin.
Let Time = width of echo pulse in μs

- Distance in centimeters = Time / 58
- Distance in inches = Time / 148

A.1.3 Pins Configuration

TABLE A.2: HC-SR04 pins configuration

Pin	**Function**
VCC	5V DC supply voltage
GND	Ground connection
Trigger	The trigger signal for starting the transmission is given to this pin. The trigger signal must be a pulse with 10μs high time. When the module receives a valid trigger signal it issues 8 pulses of 40KHz ultrasonic sound from the transmitter. The echo of this sound is picked by the receiver.
Echo	the module outputs a waveform with a high time (positive duty cycle) proportional to the distance.

A.2 RPLIDAR A2M4

FIGURE A.4: RPLIDAR A2M4

The RPLIDAR A2 is the next generation low cost 360 degree 2D laser scanner (LIDAR) solution developed by SLAMTEC. It can take up to 4000 samples of laser ranging per second with high rotation speed. And equipped with SLAMTEC patented OPTMAG technology, it breakouts the life limitation of traditional LIDAR system so as to work stably for a long time. The system can perform 2D 360-degree scan within a 6-meter range. The generated 2D point cloud data can be used in mapping, localization and object/environment modeling. The typical scanning frequency of the RPLIDAR A2 is 10hz(600rpm). Under this condition, the resolution will be 0.9

. And the actual scanning frequency can be freely adjusted within the 5-15hz range according to the requirements of users.

A.2.1 System Connection

The RPLIDAR A2 consists of a range scanner core and the mechanical powering part which makes the core rotate at a high speed. When it functions normally, the scanner will rotate and scan clockwise.And user scan get the range scan data via the communication interface of the RPLIDARand control the start, stop and rotating speed of the rotate motor via PWM. The RPLIDAR A2 comes with a rotation speed detection and adaptive system. The system will adjust the angular resolution automatically according to the actual rotating speed. And there is no need to provide complicated power system for RPLIDAR.

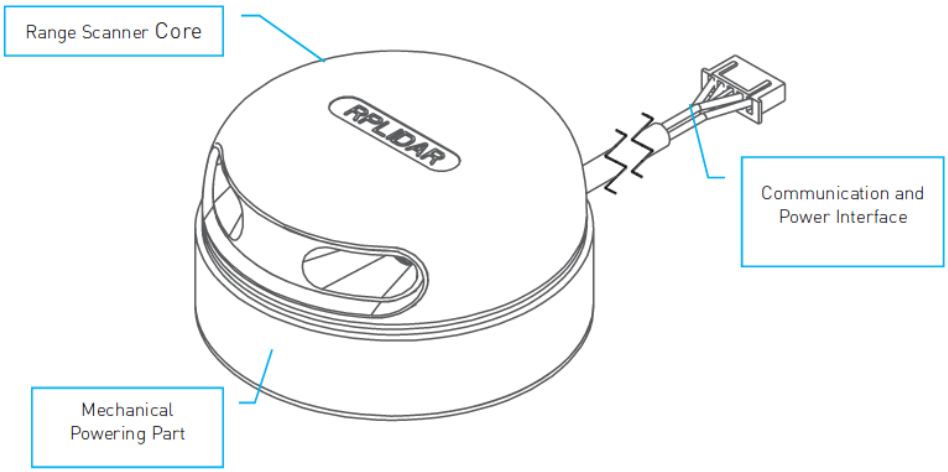


FIGURE A.5: RPLIDAR A2M4 Structure

A.2.2 Mechanism

The RPLIDARA2 is based on laser triangulation ranging principle and adopts the high-speed vision acquisition and processing hardware developed by SLAMTEC. The system ranges more than 4000 times per second.

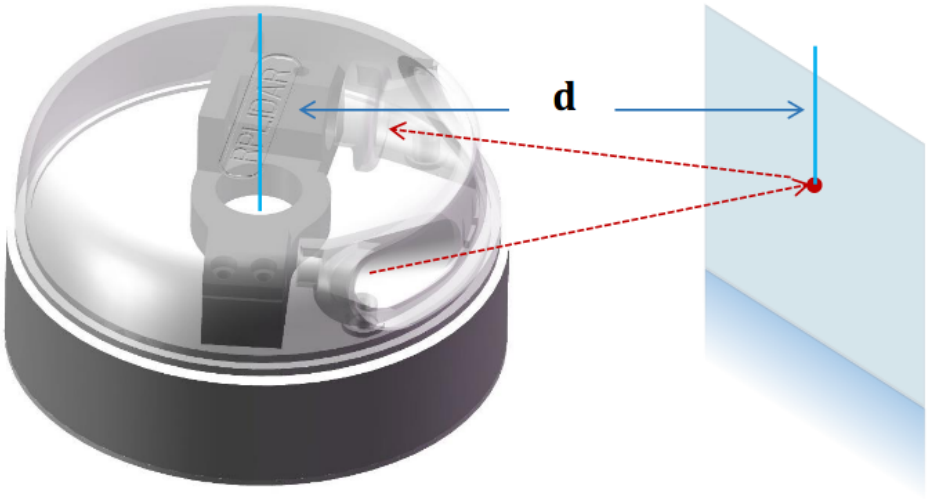


FIGURE A.6: RPLIDAR A2M4 Mechanism

During every ranging process, the RPLIDAR emits modulated infrared laser signal and the laser signal is then reflected by the object to be detected. The returning signal is then sampled by vision acquisition system in RPLIDAR and the DSP embedded in RPLIDAR starts processing the sample data and outputs distance value and angle value between object and RPLIDAR via communication interface.

A.2.3 Measurement Performance

- For Model A2M3/A2M4 Only

Item	Unit	Min	Typical	Max	Comments
Distance Range	Meter(m)	TBD	0.15 - 6	TBD	White objects
Angular Range	Degree	n/a	0-360	n/a	
Distance Resolution	mm	n/a	<0.5 <1% of the distance	n/a	<1.5 meters All distance range*
Angular Resolution	Degree	0.45	0.9	1.35	10Hz scan rate
Sample Duration	Millisecond(ms)	n/a	0.25	n/a	
Sample Frequency	Hz	2000	≥4000	4100	
Scan Rate	Hz	5	10	15	Typical value is measured when RPLIDAR takes 400 samples per scan

A.2.4 Coordinate System Definition of Scanning Data

The RPLIDAR A2 adopts coordinate system of the left hand. The dead ahead of the sensors is the x axis of the coordinate system; the origin is the rotating center of the

range scanner core. The rotation angle increases as rotating clockwise. The detailed definition is shown in the following figure:

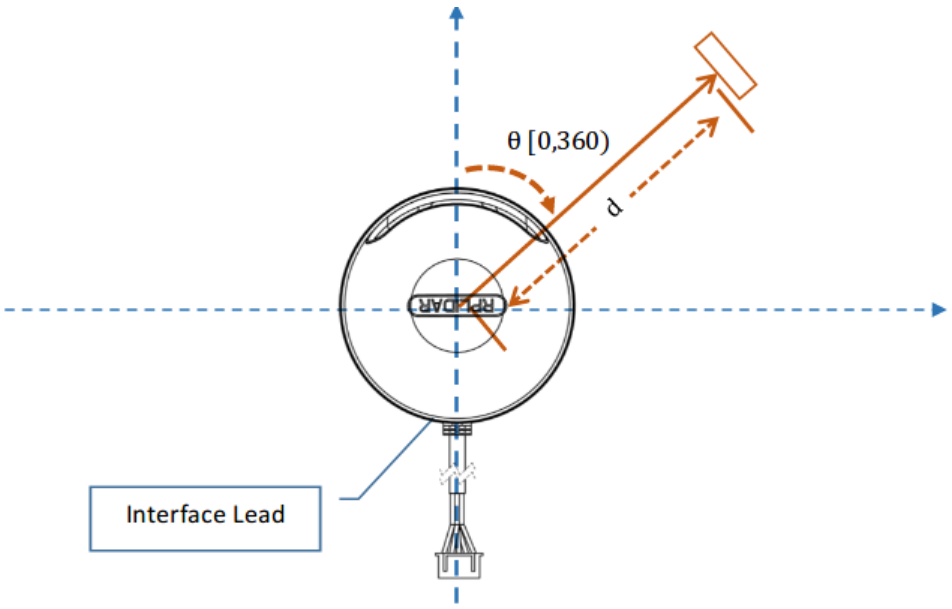


FIGURE A.7: RPLIDAR A2 Rotation

A.2.5 Laser Power Specification

- For Model A2M3/A2M4 Only

Item	Unit	Min	Typical	Max	Comments
Laser wavelength	Nanometer(nm)	775	785	795	Infrared Light Band
Laser power	Milliwatt (mW)	TBD	3	5	Peak power
Pulse length	Microsecond(us)	60	87	90	
Laser Safety Class			FDA Class I		

Note: the laser power listed above is the peak power and the actual power is much lower than the value.

A.2.6 Communication Interface

The RPLIDAR A2 uses separate 5V DC power for powering the range scanner core and the motor system. And the standard RPLIDAR A2 uses XH2.54-5P male socket. Detailed interface definition is shown in the following figure:

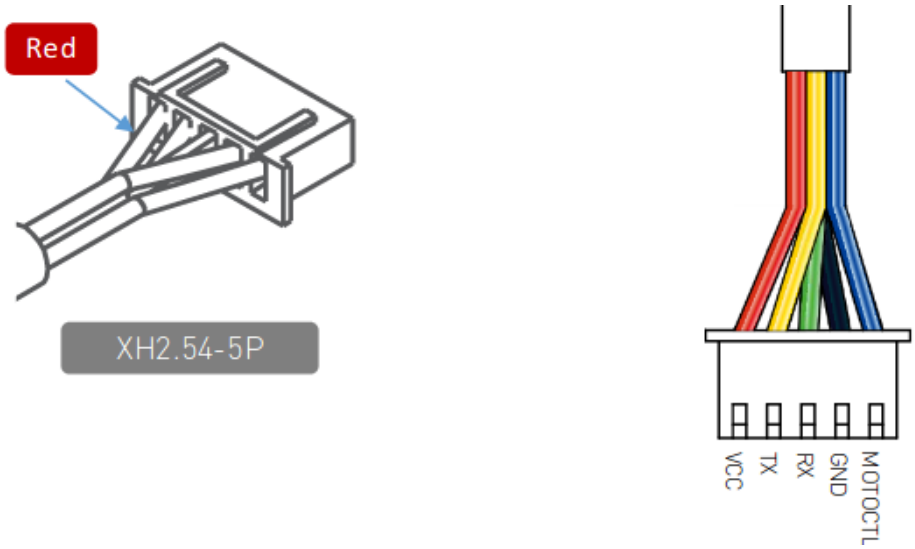


FIGURE A.8: RPLIDAR A2 Communication Interface

Color	Signal Name	Type	Description	Min	Typical	Max
Red	VCC	Power	Total Power	4.9V	5V	5.5V
Yellow	TX	Output	Serial port output of the scanner core	0V	3.3V	3.5V
Green	RX	Input	Serial port input of the scanner core	0V	3.3V	3.5V
Black	GND	Power	GND	0V	0V	0V
Blue	MOTOCTL	Input	Scan motor /PWM Control Signal(active high, internal pull down)	0V	3.3V	5V

Bibliography

[1] "SLAM for Dummies " ,Søren Riisgaard and Morten Rufus Blas .

[2] "Programming Robots with ROS", Morgan Quigley, Brian Gerkey & William D. Smart, 2015.

[3] Introduction to Mobile Robotics, "SLAM: Simultaneous Localization and Mapping", Wolfram Burgard, Cyrill Stachniss, Kai Arras, Maren Bennewitz.

[4] F1/10th Autonomous Racing, "Simultaneous Localization & Mapping", Paril Jain, Penn Engineering.

[5] Hector SLAM for robust mapping in USAR environments, "ROS RoboCup Rescue Summer School", Stefan Kohlbrecher, Johannes Meyer, Karen Petersen, Thorsten Graber, 2012.

[6] Performance Analysis of the Microsoft Kinect Sensor for 2D, "Simultaneous Localization and Mapping (SLAM) Techniques" Kamarulzaman Kamarudin , Syed Muhammad Mamduh , Ali Yeon Md Shakaff, Ammar Zakaria, 2014.

[7] wiki ros: Hector SLAM, http://wiki.ros.org/hector_ slam

[8] REST Journal on Emerging trends in Modelling and Manufacturing, "A critical comparison between Fast and Hector SLAM algorithms", Mustafa Eliwa, Ahmed Adham, Islam Sami and Mahmoud Eldeeb, VOL:3, 2017

[9] RealVNC, https://www.realvnc.com/en/

[10] raspberry pi, https://www.raspberrypi.org/

[11] wiki ros: rplidar, http://wiki.ros.org/rplidar

[12] Real-Time SLAM with Octree Evidence Grids for Exploration in Underwater Tunnels, Nathaniel Fairfield, George Kantor, David Wettergreen, The Robotics Institute Carnegie Mellon University Pittsburgh, 2006.

[13] wiki ros: rviz, http://wiki.ros.org/rviz