Assistive Localization and Autonomous Navigation for a Multi-Robot System

Control and Automation

AALBORG UNIVERSITY
DENMARK

Master Thesis

Amalia Lelia Cretu-Sircu, Ion Sircu

June, 2020

AALBORG UNIVERSITY
DENMARK

Title:
Multirobot Autonomous Navigation through Assistive Localization

Theme:
Control of Complex Systems

Project Period:
CA9, January 2020
Date: 01.09.2019 - 03.06.2020

Group number:
934

Participants:
Amalia Lelia Cretu-Sircu
Ion Sircu

Supervisors:
Henrik Schiøler

Total Pages: 127

The Technical Scientific Faculty
School of Information and Communication Technologies (SICT)
Fredrik Bajers Vej 7B
9220 Aalborg East

Synopsis:

This project develops a ROS framework for assistive localization in a multi-robot setup. It is well known that autonomous robots need to know their location in an environment and to improve their localization estimate a map of the environment is needed. A robot that localizes itself and maps the environment needs encoders, an inertial measuring unit (IMU) and an environment perception sensor such as a LiDar. When thinking about industrial environments where several mobile robots need to perform various tasks, the cost of this solution can be very high. In this thesis, the proposed strategy in the context of Industry 4.0 is to have a lead robot equipped with necessary sensors for localization and mapping and a swarm of client robots that have essential sensors for motion: encoders and IMU. The solution proposes the use of a low-cost ultrasonic global localisation system (GoT). Although precise in localization, the system suffers from systematic errors. These systematic errors are handled by fingerprinting problem-locations on the map and patching the GoT signal with error corrections. The lead robot creates and updates the map made available to the client robots. The client robots receive the patched GoT signal to localize and navigate the map. In order for all robots to share the same map and receive the corrected signal a ROS framework is developed to support a swarm of mobile robots.

Names and Signatures:

Amalia Lelia Cretu-Sircu

_____

Ion Sircu

_____

Preface

The authors of this report are tenth semester Master's students of the Control and Automation programme in Aalborg University (AAU). This report constitutes a long Master's Thesis in part fulfillment of the degree. The research started September 2019 (ninth semester) and continued until May 2020 (tenth semester).

The thesis is written at an appropriate technical language level in accordance with the programme's curriculum to satisfy the learning objectives of the two semesters. The target audience for this report is students in the ninth and tenth semester of Control and Automation as well as first semester students in Robotics from Aalborg University.

The main scope of this project is to develop and implement a multi-robot configuration for localization and mapping using specific sensors. The work performed in this project is aimed at Aalborg University research projects on Industry 4.0. The solution developed in this work can be generalized to any multi-robot setup, but the facilities and infrastructure mentioned in the project had AAU MP Laboratory in focus. In this project, specific research systems have been used such as the GamesOnTrack ultrasonic positioning system, and pursued integration with the path planning platform from AAU Uppaal.

During the national lock-down in Denmark from March to May 2020, the implementation of the solution was done only through simulations and once access was gained to the laboratory mid-May, tests were performed on the robot. Components for the second robot as part of the multi-robot setup did not arrive in due time. Despite constraints, the tests have shown promising results and a knowledgeable research endeavour.

With this occasion, we would like to thank from a safe distance Henrik Schiøler for supervising our project since 2019 through unique times such as the CoVid-19 pandemic.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Context . . . 3

2 Problem Formulation and Methodology **7**
2.1 Problem Statement . . . 8
2.2 Methodology . . . 9
2.3 Project Structure . . . 11
2.4 Requirements Specifications . . . 13
2.5 Added Value - Related Work . . . 14
2.5.1 SLAM . . . 14
2.5.2 Indoor Positioning and Fingerprinting . . . 16

3 Analysis **18**
3.1 Hardware . . . 18
3.1.1 GamesOnTrack (GoT) Receiver Subscriber Rev.3 . . . 19
3.1.2 Range-finder scanner - RPLidar . . . 20
3.1.3 GY-88 IMU - HMC5883L . . . 20
3.1.4 Motors and Encoders XH-500-37D . . . 20
3.1.5 Teensy 3.6 . . . 21
3.1.6 RaspberryPi 3 . . . 21
3.2 Software . . . 23
3.2.1 Computation Graph Level . . . 23
3.2.2 Filesystem Level . . . 27

4 SLAM and Robot Navigation **29**
4.1 Robot Localization . . . 29
4.1.1 Hidden Markov Models and Recursive Bayesian Filter . . . 29
4.1.2 State Vector and Environment . . . 31
4.1.3 Robot Perception . . . 33
4.2 Robot Mapping . . . 34
4.3 Simultaneous Localization and Mapping - SLAM . . . 36
4.4 Sensor Fusion . . . 39
4.5 Robot Navigation . . . 41

5 Low-Level Design and Implementation **42**
5.1 Local and Global Robot Positioning . . . 42
5.1.1 Odometry . . . 42
5.1.2 GoT Positioning . . . 47
5.1.3 Sanity Check . . . 49
5.2 Robot Orientation . . . 51

5.2.1 Directional Vector Heading 51

6 High-Level Design **53**
6.1 Gazebo Multi-Robot Simulation Environment 53
6.2 ROS SLAM 55
6.3 Sensor Fusion 56

7 Implementation **58**
7.1 GoT ROS node and Fingerprinting 58
7.1.1 Steps 1-5 Simulating the GoT Beacons 60
7.1.2 Step 6 - Patching the GoT Position Error 62
7.1.3 Step 7 - Store all errors and the respective ground truth estimate. . 62
7.1.4 Step 8 - Distribute the GoT Error Correction to All Robots in the Multi-Robot Setup 64
7.1.5 GoT Error Correction Parameters 66
7.2 Final Notes 68

8 Simulation and Implementation Testing **69**
8.1 ROS and Gazebo Simulation Results 69
8.2 Laboratory Results 72

9 Conclusions **78**
9.1 Evaluation 78
9.1.1 Simulation Results Evaluation 79
9.1.2 Laboratory Results Evaluation 80
9.1.3 Requirements Specifications Evaluation 82
9.2 Perspective 85
9.3 Future Work 86

Bibliography **87**

A Components Wiring to Teensy3.6 Pins **91**

B The KF and EKF **92**

C Robot Perception in SLAM **96**

D Generalized System Model **98**

E Mathematical Model of EKF SLAM **99**

F Magnetometer Calibration **101**

G RQT Graph of the Simulated Robot SLAM: ROS Nodes and Topics **103**

H RQT Graph of the Simulated Robot SLAM: ROS Frames **104**

I Simulated Robot in ROS using URDF Files **105**

J Robot Navigation Methods **106**

K Tutorial to Configuring the ROS Framework for the 2D Robotic Solution **110**
K.1 Configuring RPi3 and Remote Workstation 110
K.2 Configuring the ROS Joystick Package 110
K.3 Configuring the Simulation Environment for the Robot 113
K.4 Spawning Simulated Robots into Gazebo Worlds 117
K.5 Configuring Gazebo with the GoT Node 118

1 Introduction

1.1 Background

This section creates the background and describes the context of the project to ensure the problem formulation refers to real-life scenarios and challenges of actuality.

The most sought after feature of today's mobile robots is autonomy. This demand has evolved from very early technologies started by the first industrial revolution, Industry 1.0. It is because of these beginnings that robots are rather designed for manufacturing than other fields.

Industry 1.0 is referred to as the 'Golden Age' of productivity started by the industry's mechanisation using non-electricity powered technologies such as the steam engine, motorised machines, etc. These new levels of automation lowered production costs and brought economic and social development [1].

Industry 2.0 is marked by the need of mass-production. Electricity-based technologies were introduced in the industry such as assembly lines, automated machines, etc. Automation meant that products' quality could be guaranteed. As such manufacturing processes started to involve practices of standardization. Needless to say, these technological evolution spurred development in other areas such as computing and communication technologies. [1]

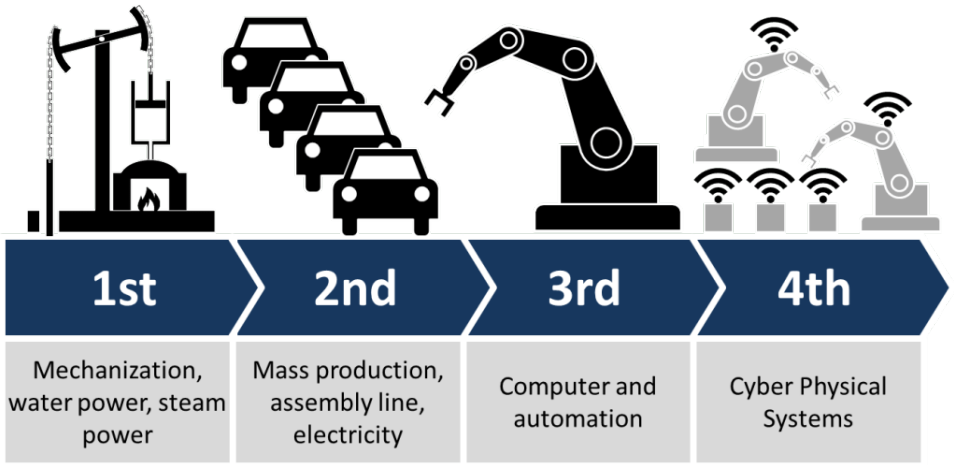


Figure 1.1: The delimitation of the four major industrial revolutions. The classification of technologies under each revolution is generally not the same throughout the community. Source: By Christoph Roser at AllAboutLean.com under the free CC-BY-SA 4.0 license. [2].

Cross-disciplinary technologies such as mechatronics, robotics and autonomous systems mark the start of Industry 3.0 - the digital manufacturing. Combining digital innovations and connectivity, manufacturing processes become partially or fully digitized for better resource management and responsiveness to customer's demands. Robots advance from doing highly-repetitive tasks to being involved in series of tasks with a certain underlying level of complexity. Robots become smarter, faster and cheaper. [1]

As the technological development brought paradigm shifts leading to industrial revolutions [3], the fourth industrial revolution brings advanced digitalisation technologies from outside manufacturing that Industry 4.0 refers strictly to concepts such as: smart factory, dark factory, smart manufacturing, industrial internet-of-things (IOT), embedded systems, harnessing the potential of connected devices, communicating with each other to make reliable decisions. [1] This is also depicted in Figure 1.1 as cyber-physical systems.

Cyber-physical systems refer to the merge between the real, physical world and the digital [3]. The reliability of these machines has led to minimal human intervention if not at all. As the robots became smarter, the tasks solved are increasingly more complex. Examples of such task are collaborative robots (human-robot close interaction), robots finishing manufacturing processes with no human intervention, real-time deliveries of materials by autonomous mobile robots, etc.

The interconnection of all devices (sensors, machines, robots, tools) allows for data gathering, exploration and fusion for better decisions. Based on big data a plethora of technologies can be employed in a smart factory. Some of the most important technologies defining Industry 4.0 can be visualized in Figure 1.2 with autonomous robots as the driver for technologies that can further improve their capabilities.

Autonomous robots are an important example for countless industries, especially for industrial manufacturing context. Autonomous robots needed in manufacturing are of many types (manipulators, machines, mobile) but of relevance for this project remain the mobile robots.

Their role in the smart factory can be diverse but of importance remains the ability to autonomously navigate from point-to-point the factory's floors avoiding obstacles or other mobile robots, and eventually, carry materials needed in other parts of the plant. Such robots can be integrated in a manufacturing execution systems and receive signals as to where and when to move. As exemplified in [4], modular conveyor belts can be placed on mobile robots and depending on the production needs, these robots can redesign the manufacturing flow near designated programmable logic controllers (PLCs) [1]. This example brings about the context of this project.

[1] PLC is an industrial computer designed to control manufacturing processes such as assembly lines, robotic tools, etc.

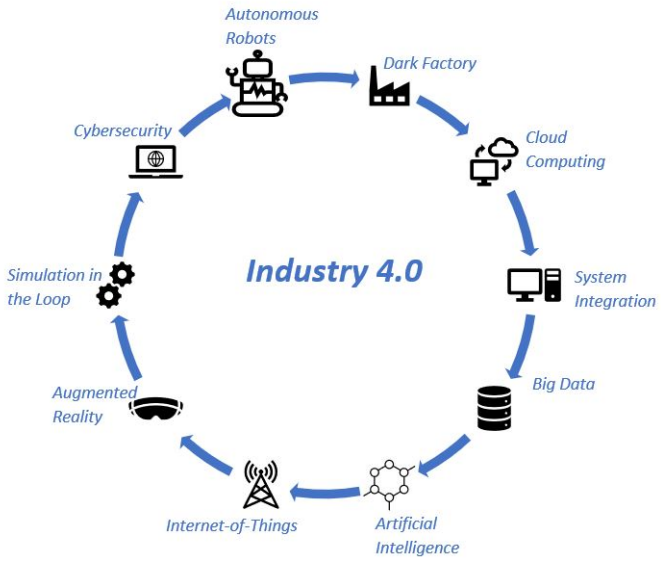


Figure 1.2: The key technologies of the Industry 4.0 that are used in a smart factory [5]. Inspired from source: [4]

.

Mobile robots for industrial manufacturing have not always been completely autonomous, but to varying degrees. An autonomous mobile robot can make decisions on how to get from point A to point B based on a navigation algorithm that knows the map of the factory and the location of the robot. Accurate and precise localization of the robot to provide autonomy is of interest to this project.

1.2 Context

This section presents the objective of the project and the connection to the real-world problem it tries to solve.

A prevailing type of robots in industrial manufacturing are the automated guided vehicles (AGV) defined by a mobile robot following a magnetic or painted path [6]. The robot only needs to know the distance along the pathway to the target not its location in the environment. Obstacles on the pathway are avoided by the use of LiDar. Being limited by their wired pathway, the AGV are inflexible and cannot leave the physical path. Being able to self-localize is important for leaving the physical path.

It is possible, however, if there exists a virtual map of the working environment the robot may be able to follow a virtual pathway using the map and knowing where it is on the map with the help of special markers, i.e. reflective, QR, tags, etc. The robot uses a perception sensor for the markers, i.e LiDar, camera, RFID, etc. and using triangulation can calculate its position. This is shown in the figure below (Figure 1.3).

Figure 1.3: Industrial mobile robot using a map of the environment known a priori to localize and navigate. The positioning is done with the help of markers (i.e. reflective, QR codes, etc.) strategically placed in the environment [6].

Limited movement flexibility and marker placement are some disadvantages of this type of autonomy. Markers have to be detected by the sensor, hence they have to be placed at the same height of the sensor. Moreover, problems arise from the obstruction of markers by high shelves, moving objects or people standing in front of one.

Autonomous mobile robots (AMR) are defined as the robots free of any enhanced-infrastructure for navigation [6]. These robots are flexible as can travel anywhere in the industrial environment while avoiding obstacles and humans in their path. The map of the environment is not known a priori. A LiDar can still be used. In order to operate in such busy environments fully autonomously the robots need to create a map of the environment while finding their position on the map and taking decisions on reaching point B from point A. Path planning and tracking needs to consider obstacle avoidance as an important dimension of AMRs especially when working near humans.

It can be observed that robots start to become equipped with more sensors compared to the earlier version where the industrial infrastructure was supposed to change to accommodate the robots. Making robots smarter comes at a cost. In industrial setting where there is a need for several robots to perform various tasks, the cost of purchasing and maintaining mobile robots can become very high.

A multi-robot setup does not necessarily mean that robots collaborate although the efforts are growing in this direction. It comes naturally to have a swarm of the same type of robots each doing their own job at localisation, mapping and avoiding obstacles. Making robots collaborate in mapping an environment, recognizing each other, optimizing the route to a work station, accept appropriate tasks are in the early developments.

When discussing multi-robot setups the concept of swarm, formations [7] and multi-

agent systems comes into light as research is mature to handle information systems where robots interact with each other in order to overcome environment changes such as: obstacle avoidance, component fault, repair, replacement, etc. An overarching architecture for distributed control and information sharing between agents is an essential component for multi-robot setups. An example of multi-robot control architecture is CoMutaR as presented by [8] as the coalition formation based on multi-tasking robots. This architecture tackles both task distribution among robots and coordination amongst the teams formed.

Architectures for distributed control of robotic networks, information sharing and behaviour programming are a vast research topic but at a higher layer of abstraction than necessary in this project. The multi-robot configuration in this project is meant to demonstrate assistive localization accuracy through fingerprinting[2] created by a lead robot and shared to other 'blind'/client robots (equipped with less sensors).

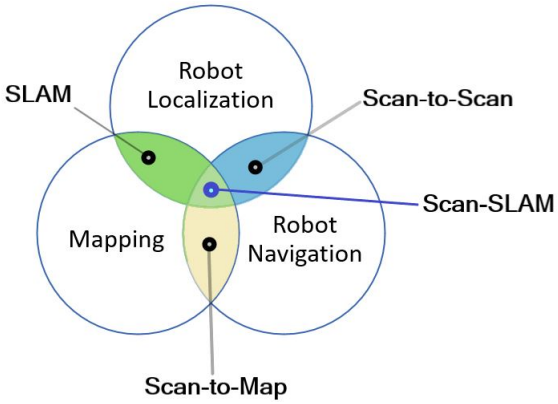


Figure 1.4: The four types of approaches to localization, mapping and navigation: SLAM(localization and mapping), scan-to-scan (localization and navigation), scan-to-mapping (navigation and mapping), scan-SLAM (SLAM and scan-to-scan or scan-to-map). Source: Inspired from [6]

Four approaches to robot autonomy implementation can be seen in Figure 1.4 where depending on the resources available one or two approaches can be implemented.

Scan-to-Scan

This method does not require a map as can be seen from Figure 1.4 but it can estimate the robot's position and pose from sequential LiDar data. The position estimate is updated continuously and is subject to drift in long-term use. It is used for process startup when a map is not yet created or when the environment has changed [6].

[2]Fingerprinting is an indoor positioning method using non-navigational radio frequency signals: Bluetooth, WiFi, ZigBee, magnetic field, light, etc.[9]. It is detailed in Chapter 6.

Scan-to-Map

The algorithm requires a stored map of the environment while the robot estimates its position by matching actual readings to the map. The matching can give erroneous results if the environment is symmetric or similar in different regions and if the stored map does not match the readings.

Integrated Approach - scan-SLAM

This algorithm solves all three aspects of the mobile robot: localization, mapping and navigation. It can be done by adding more sensors such as a camera to help extract environment features and match object in a map better, but at the expense of computational power.

While each algorithm presented has both advantages and disadvantages, any combination can be used to reach the requirement specifications for the mobile robot however this project focuses solely on SLAM.

An AMR needs to create a map of an unknown environment while finding its own position on the map. In order to solve this problem it uses an algorithm. Simultaneous Mapping and Localization (SLAM) is a solution to the navigation problem of a robot in an unknown environment. It tries to answer two questions: where am I? and how does the environment look like? It is simultaneous because it solves what metaphorically is a chicken-and-egg problem: the robot needs a map to know where it is however to create a map it needs to know where it is located.

SLAM is a key component in autonomous mobile robots. Using relative observations of the surroundings the platform can navigate in an unknown environment while creating a map. Multiple sensors can be used with SLAM with the most prevalent: encoders and LiDar however a global positioning system can also be used. SLAM is also a methodology for sensor fusion that solves the navigation problem in an unknown environment having constraints on the resources available, i.e. sensors, storage.

As an algorithm, SLAM is comprised of two parts: robot localization and building of the map. Robot state is what builds the localization part being comprised of simple instances: position, orientation, velocity, etc. while mapping is a representation of landmarks position, obstacles, features, etc. A map is needed to reduce the position error of the robot and to allow for a visualized path planning and tracking by an operator [10]. If the robot depends on dead-reckoning [3] it leads to an accumulation of error and the robot drifts from the set position. SLAM is a solution for reducing localization errors.

[3]Dead-reckoning is a process of finding current position based on the previous position plus measured distance elapsed on the course. It accumulates directional errors making the robot drift from the set course. The accuracy of this method can be improved by adding positional sensors such as GPS/absolute positional system.

2 Problem Formulation and Methodology

It is envisioned that the mobile robot to be an integrated part of a smart production line receiving jobs from a delegating server to transport for example Lego bricks. In the specific context of the Smart Production Lab AT AAU, each production unit involved - PLCs, manipulators, collaborative robots, etc. has an unique ID and task assigned by the production planner (on the server).

The mobile robot is also part of the production line and receives an ID and a task from the server. As a job is received, the trajectory planner (also on the server) calculates the route between the current position of the robot and the desired location. The trajectory planner communicates where the robot should go and the robot communicates back the current location on the map. There are designated stations where the robot has to arrive, receive the packet, receive the location of the new delivery station and arrive at the specified delivery station. An illustrative example of the laboratory environment can be seen in Figure 2.1.

The map of the environment is unknown (landmarks/obstacles are not known) as it is expected not all operators to have or be able to produce a map of the production environment. The robot is equipped with the standard sensors for performing SLAM - encoders and LiDar. There is also a low-cost ultrasonic solution for global positioning of the robot to be used with the algorithm to improve robot's pose in the map. Navigation or planning of the robot trajectories is done at the higher level than the job delegating server.

The SLAM algorithm is the solution used in this project to estimate the robot's current position based on low-cost, erroneous and noisy sensors, while not having a map of the environment or a high-precision localization sensor to be used as ground-truth. In a typical implementation of the SLAM algorithm, Kalman filters can be used. Other filters can also be used, i.e. particle filter, however this project focuses on the extended Kalman filter (EKF). The EKF for SLAM is described in Chapter 4.

The goal of this project is to develop a low-cost mobile robotic solution to work in a multi-robot configuration for a smart production laboratory. Localization accuracy obtained by the lead robot is the focus of this project through the algorithm of simultaneous localization and mapping (SLAM). The research seeks to improve the accuracy in localization of SLAM through sensor fusion and fingerprinting.

Next section formulates the problem statement of the project and the methodology of how it intends to solve it.

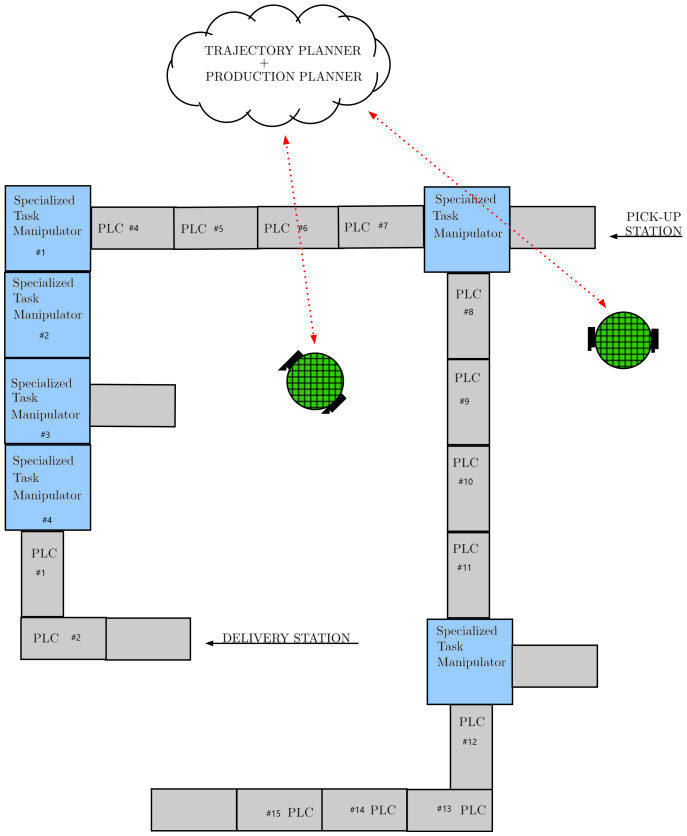


Figure 2.1: Illustrative example of the Smart Production Lab at Aalborg University. It can be seen different production units - PLCs in gray and robot manipulators for specialized tasks in blue with designated pick-up and delivery stations for the mobile robot.

2.1 Problem Statement

This section discusses the problem formulation and the methods that are going to be used in order to develop the solution of the problem. Finally, it exposes the project structure meant to help the reader understand the building steps of the solution and its implementation.

Having exposed the context of this project, it is clear that the main goal of the project is to improve robot localization accuracy and precision through SLAM and fingerprinting on a low-cost hardware robotic solution. Sensor fusion is expected to further compensate for sensor noisiness and loss of accuracy. Four sensors: magnetometer, wheel encoders, LiDar and ultrasonic global positioning systems GoT are used on the lead robot to navigate in an unknown environment while creating a map and self-localizing. The client robots in the multi-robot configuration are envisioned to use encoders, a magnetometer

and corrected GoT through fingerprinting.

Commonly, SLAM uses encoders and a type of perception sensors i.e. LiDar or camera to map and find out where the robot is on the map. Through sensor fusion between all available sensor data including SLAM position estimation, accuracy improves so that it can be used as ground-truth against GoT. Using fused SLAM (SLAM position estimate + fused sensor data) as ground-truth allows fingerprinting of the systematic errors GoT system exhibits in non-line-of-sight situations. Fingerprinting allows for correcting or patching the GoT signal where the error between ground-truth GoT is higher than a required threshold. By equipping the client robot with a GoT receiver it is possible to obtain its global position on the map without the errors caused by interference. Finally the goal of the project is formulated below.

Develop a low-cost multi-robot solution to improve accuracy of localization estimation through SLAM, sensor fusion and fingerprinting. The multi-robot setup is comprised of a lead robot and a client robot. The lead robot uses SLAM and sensor fusion with sensors as encoders, magnetometer, LiDar and GoT to fingerprint GoT interference-affected locations. The client robot uses encoders, magnetometer and the interference-corrected GoT signal. The increased accuracy of localization estimate improves robot autonomy in map navigation.

In order to reach the goal of the project, a methodology to guide the development of the solution is described below. The methodology is meant to bring clarity in the methods and approaches used in the project for reaching the solution.

2.2 Methodology

Through the methodology used in this project a clear demarcation is sought between low-level and high-level architecture .

Low-level architecture denotes both hardware and software design of components that have to work seamlessly. This part is meant to be an autonomous plug-and-play type of architecture where no input or modification is required from the user. The hardware components of the low-level architecture includes:

- Motor gear (i.e. wheel motors, motor drivers)
- Sensors (i.e. magnetometer, GoT receiver, wheel encoders, LiDar)
- Micro-controller (i.e. Teensy 3.6)
- Batteries

The software algorithms in the low-level architecture:

- handles incoming sensor data;
- calibration and correction of sensors (i.e magnetomer);
- calculation of position, orientation of the robot and obstacles (i.e encoders, LiDar, GoT);
- data transmission protocol between sensor data and high-level architecture (i.e ROS message frame)

The high-level architecture includes both hardware and software components that rely on data transmitted correctly and at the right frequency from the low-level components. High-level components are:

- Computer (i.e. RaspberryPi 3)
- Terminals (i.e. joystick, keyboard)
- Remote workstation

The components of this architecture can change and depend on user input. This design is made for trial-and-error scenarios where resources can be added or removed based on the computational needs or requirement specifications (described in the next section). The software component includes:

1. host ROS framework of nodes, topics, messages, services, etc.
2. calculation of robot pose estimates, landmarks, map (i.e. SLAM)
3. calculation of trajectories (i.e. navigation algorithm)
4. data transmission protocol to low-level components (i.e ROS message frame)

It is envisioned that these 2 levels of architecture are defining the lead mobile robot as a stand-alone solution and further integration with a production planner or a trajectory planner can be done through the high-level layer. The architecture for the lead robot can be seen in Figure 2.2. The client robot design follows the same architecture adjusting for the fewer sensors.

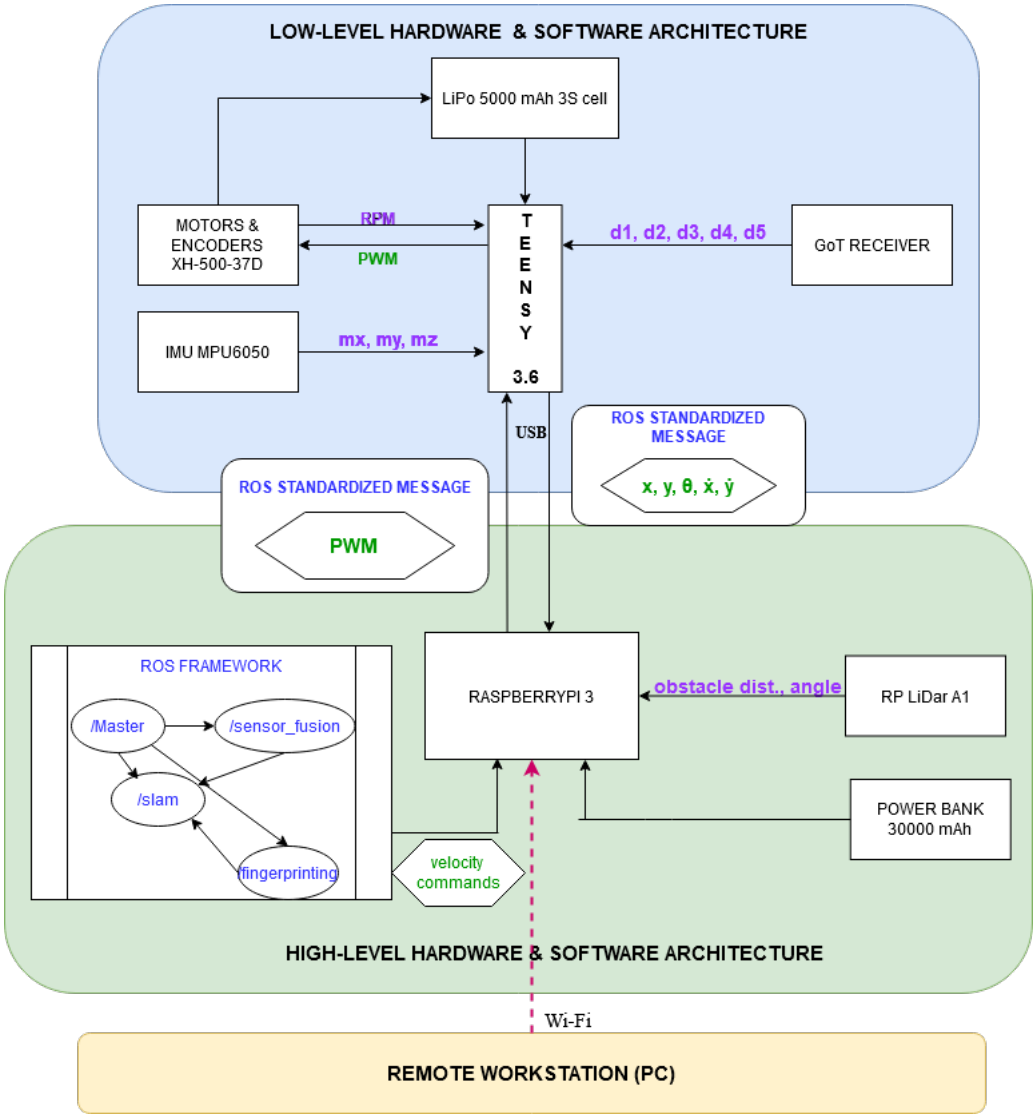


Figure 2.2: The low- and high-level hardware and software design for the lead 2D mobile robot. The sensors are transmitting raw data: magnetic measurements m_x, m_y, m_z, distances between beacons and receiver d_1, d_2, d_3, d_4, d_5 and revolutions per minute (RPM) to Teensy which after processing sends a ROS message with processed data for pose and velocities. The ROS framework sends velocity commands to RPi3 which transforms them into PWM going to Teensy as a ROS message.

2.3 Project Structure

This section is meant to show the red thread of the report in order to understand the development of the design and implementation of the solution. It is meant to bring

clarity of the results obtained from each chapter and the assumptions used to proceed to the next. In Figure 2.3 the red thread is symbolically indicated by the red arrows which also show the flow direction.

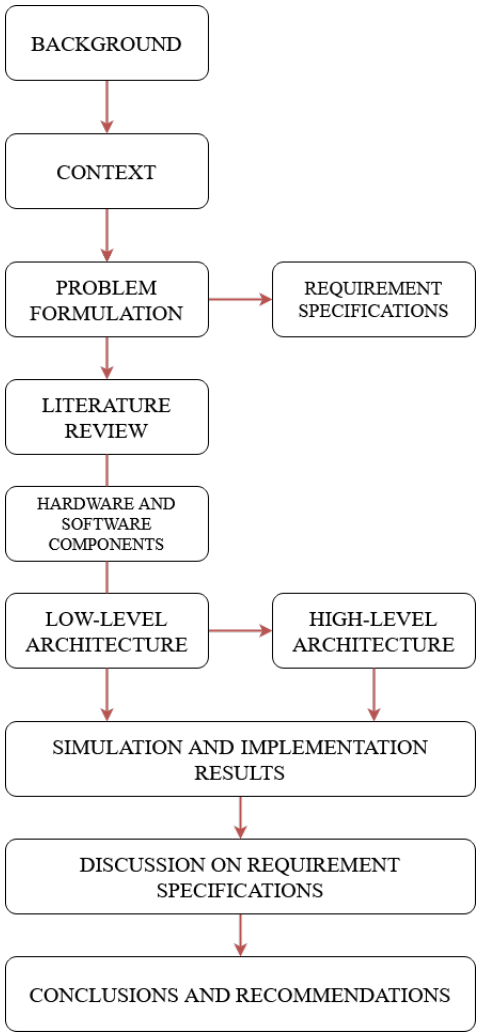


Figure 2.3: The main sections and chapters of the report providing results towards the development of the solution. The red arrows are meant to represent the red thread of the flow of the report.

This section exposed the problem statement of the project, the methodology to guide the solution and the project structure to visualize the building steps of the solution. It has also mentioned that the project should also fulfil the requirement specifications imposed by the authors in order to evaluate the efficiency and completeness of the solution. The requirement specifications are explained in the next section.

2.4 Requirements Specifications

Requirement specifications are set by stakeholders of a product. Stakeholders hold an interest in the project - engineers interested in Industry4.0, students studying control, automation or robotics. These requirements represent the goal that guides the development of the solution beyond its construction. The solution has to reach or stay within the limits imposed by these requirements.

It is important how these requirement specifications are formulated as there should be no room for second interpretations. Their formulation has to be SMART - specific (targeted area), measurable, attainable, relevant and time-bound. It means that the formulation regarding a requirement has to be about a targeted area or interface, the requirement is measurable and can be obtained, is relevant for the solution and reaches the goal in a timely manner.

When the implementation results of the solution are available, these results are verified against the requirement specifications for validation of the solution.

The requirements are formulated for each important component interface.

- **Requirement specifications on the robot design**
  1. Finalize a unified hardware design of the robot.
  2. Establish a reproducible wiring of all components for a 2nd robot.
  3. Build a 2^{nd} robot.
- **Requirement specifications on the interface between low-level and high-level architecture**
  1. Wrap all sensor data in appropriate ROS message type.
  2. ROS Master receives all sensor data.
- **Requirement specifications on ROS Framework**
  1. Wrap sensor data into message types accepted by *robot_localization* package.
  2. Wrap sensor data into message types accepted by *hector_slam* package.
  3. Simulate a GoT system in ROS.
  4. Simulate the real robot in Gazebo.
  5. Simulate a navigation algorithm.
  6. Simulate 2 robots in a map with different sensor configurations.
- **Requirement specifications on sensor fusion and GoT**
  1. Fingerprint all locations in Gazebo and laboratory where GoT transmits erroneous measurements higher then threshold of 0.2 m.

2. Patch the GoT signal at the fingerprinted locations with a scaling parameter.
3. Observe an improvement in fused odometry with patched GoT of over 0.1 m in Gazebo for the client robot when passing in a fingerprinted location on the map.
4. Observe an improvement in fused odometry of over 0.1m in laboratory for the client robot when passing in a fingerprinted location on the map.
5. Observe an improvement in fused localization with GoT measurements of over 0.1 m in Gazebo for the lead robot.
6. Observe an improvement in fused localization with GoT measurements of over 0.1 m in laboratory for the lead robot.
7. Finalize a plug-and-play ROS infrastructure for the 2D robotic solution.

2.5 Added Value - Related Work

This section presents work and research done in relation to the main focus of the project: localization estimation through SLAM and fingerprinting. By understanding how much has been achieved so far, the added value of the research done in this project can be shown.

When discussing separately about the two topics SLAM or fingerprinting the research is abundant, although SLAM is rather an early research topic. Fingerprinting is a widely used position estimation method for radio frequency (RF) signals and it is not a novel method. It is particularly used on position estimation of mobile devices, i.e. phones. Uses cases for this type of implementation are still developing with a leading use case for indoor mapping using personal mobile devices. Research or work overlapping both SLAM and fingerprinting is new and scarce especially when involving mobile robots in an industrial setup.

2.5.1 SLAM

A review of the literature on how SLAM has been designed for mobile robots in an industrial setting given resource constraints is presented. As mentioned previously, the SLAM algorithm consists of two main goals: estimating the state of a robot and building a model or map of the environment. The state of a robot is described by its on-board sensors which could describe position and orientation. A map of the environment is usually built using information from these sensors at the same time as localizing the robot. If the map is known *a priori* or if the robot knows its global location by means of a fixed referential system, i.e. GPS then the robot does not need to perform SLAM to know its location. However, without a map, dead-reckoning from odometry would quickly drift over time when estimating the pose of the robot whereas in the presence of a map the robot can eliminate its localization error by going to areas it visited previously, a process known as loop-closure (or place recognition).

According to [11] loop-closure is a critical part of SLAM such that sacrificing loop-closure reduces SLAM to odometry. If odometry is obtained by integrating wheel encoders, the obtained robot pose drifts quickly, rendering it unusable [12, Chapter 6]. Including however observations about the environment will correct trajectory drifts. According to [13] more recent odometry algorithms based on visual and inertial measurements have a very small drift raising the question if SLAM is necessary.

To answers this question [11] provides a three-part answer:

- SLAM research has produced the visual and inertial navigation (VIN) which constitute the state-of-the-art e.g. [14], [15]. This algorithm ignores the loop closure part of SLAM. The research on sensor fusion has spurred for designs with low-cost sensors, no GPS, etc. [11]
- If navigation disregards loop-closures, the robot interprets the world as an "infinite corridor". The advantage of loop-closures is that it reveals the true topology of the map allowing the robot to find shortcuts between points on the map [11].
- Some applications do require a complete map of the environment, e.g. for cases where the robot needs to explore the structural integrity of a building or ground coverage for military operations, etc. [11].

Despite an increasing on-going SLAM research, it cannot be said that there is a one-size-fits-all SLAM algorithm to all robot types, environments and resources. In fact, there are many types of SLAM algorithms, each answering to a specific robot type/environment/resource. The 'classic' SLAM algorithm involves a mobile robot equipped with wheel encoders and a 2D range-finder. Other mature SLAM algorithm are visual-based SLAM with slowly-moving robots (e.g. Mars rovers [16], domestic robots [17]) and visual-inertial odometry [14]. However [11] states that all current SLAM algorithms could be induced to fail if the requirements on the robot motion or the structure of the environment are too challenging. E.g a visual SLAM algorithm could fail if it requires the robot to move quickly or an algorithm relying on a range-finder could fail in a fast-changing environment. Article [11] gives 4 points on the direction of development of SLAM algorithms:

1. **Robust Performance**: the SLAM algorithm exhibits high chances of success in dynamic environments for long periods of time, fail-safe and detection mechanisms and auto-tuning capabilities [11].
2. **High-level understanding** of the environment such as semantics describing the environment, high-level geometry, etc. [11].
3. **Resource Awareness**: the algorithm is capable of matching the sensing and computational capability dependent on the resources available. [11]
4. **Task-Driven Perception**: SLAM is able to ignore sensor data not providing estimate improvements for a particular task given. [11].

SLAM is the algorithm allowing the robot to localize in an environment and it needs a trajectory planner in order to navigate it. In the book [18] is mentioned that it is now important to compute robot's current and future uncertainty to determine the right choice of control in path planning. Navigation algorithms vary from simple $go-to-point$, bug algorithms for obstacle avoidance to more complex such as optimization on cost maps. The insight brought is that not all paths induce the same level of uncertainty and the robot should be able to choose the path that has less uncertainty as that will ensure the robot reaching the target goal.

The book [18] exemplifies the *coastal navigation* algorithm where the robot follows a path that has distinct features along its way, i.e. corners, walls, furniture to help the robot stay localized as opposed to taking a shorter path through an 'empty' space where the uncertainty raises and the robot may not reach its target goal. These type of considerations go beyond the classic SLAM and bring a more robust dimension in the face of sensor and model limitations as well as dynamic or very large environments. They allow for a less strict approach when modelling the robot or the sensors, but it comes with some disadvantages: computational complexity and need to approximate [18].

2.5.2 Indoor Positioning and Fingerprinting

There is an increasing interest in real-time indoor positioning and the main technologies used are RF, infrared (IR) and ultrasonic (US). Ultrasonic is known as a low-cost fine grain position estimation system. An indoor localization system (ILS) contains receivers and transmitters. If the configuration follows NAVSAT GPS then the satellites are beacons transmitting the distance to the receiver. There are different methods to calculate this distance: time of arrival (TOA), time difference of arrival (TDOA), angle of arrival (AOA), received signal strength (RSS). Techniques using these measurements are trilateration, triangulation, fingerprinting, etc.[19]. The literature so far does not agree on a difference between the concepts of positioning and localization, thus these are used interchangeably in this project.

US ILS has proven to be low cost, reliable, scalable, energy efficient, fine grained centimeter accuracy and no interference between building rooms [19]. It can also track multiple nodes at the same time. The localization system are compared by accuracy and precision. Accuracy is the error in physical distance; precision is the repeatability of accuracy [19].

Fingerprinting is rather known as a method for position estimation based on RSS. There are two phases: a training phase and an online phase [9]. In the training phase a map (or database) is created with the RSSI value registered between the reference points, here the beacons and the targeted device. In the online phase the target measures RSSI observations and the algorithm uses the map to calculate the position estimate [9]. Methods for improving the fingerprint technique are abundant in the literature and involve using a mix of two or more technologies, using Kalman filtering for pre-processing or two techniques for improving location estimate with fewer reference points as in [20].

[9] mentions that a challenge is to create and maintain such a database as RF and US signals are vulnerable to changes in the environment such as temperature, humidity, building structure attenuation, multipath fading, etc. The positioning measurements drift and cannot be update in time.[9] In the study [9] the robot is equipped with a LiDar, Wi-Fi sensor, magnetic sensor and a light sensor. Using SLAM to estimate location on the map, SLAM estimates act as reference point on the map. Orientation errors accumulate as the moves through a feature-less environment adding more localization errors. In the meantime, for each position records the values of the sensors: RSSI, light strength, magnetic strength, etc. SLAM is used to maintain the accuracy of the position estimate during fingerprinting. It provides an error envelope for the fingerprinting position estimation. This method is envisioned for tracking mobile phones employing a pedestrian model for SLAM. It uses SLAM to update the fingerprint map due to finer sampling points and accuracy in positioning.

In [21] uses a multitude of sensors: ultrasonic distance sensors, GNSSGPS, IMU, encoders, barometric and magnetometer to perform sensor fusion. For sensor fusion it uses an extended Kalman filter (EKF) and a particle filter. The goal of the paper is to produce a seamless indoor-outdoor multi-story buildings localization estimation.

From the research done in recent works it is recently that SLAM, sensor fusion are used in connection to indoor positioning systems. The observed thread in literature is to use either of the two methodologies of Kalman filtering, SLAM or sensor fusion to improve the accuracy and precision of the ILS. ILS can be based on RF, IR or ultrasonic or any two multiple combinations. **The research thread is followed in this project by using SLAM to improve fingerprinting. Fingerprinting is not done in relation to RSS or any signal but in relation to the systematic error recorded between SLAM position estimates and GoT position estimates. Moreover, sensor fusion using the sensors available is done prior to SLAM having a more precise control input to SLAM. Results of this project are presented as a comparison between position estimates accuracy in relation to fusing different sensors.**

3 Analysis

In the analysis chapter the available resources are presented and their potential uses for the development of a solution. It starts by describing the construction of the physical robot in Section 3.2 and continues into the software resouces such as Robot Operating System (ROS) and Gazebo.

3.1 Hardware

This section presents the hardware used in the project and follows the methodological order low- and high-level.

The main hardware components available for the lead robot are:

- GoT Receiver Subscriber Rev.3
- RPLidar A1
- IMU MPU6050
- 2x motor and encoder XH-500-37D
- 2x motor drivers VNH5019A
- Teensy 3.6
- RaspberryPi 3
- LiPo 5000 mAh 3S cells
- Power Bank 30000 mAh

The hardware design has sustained few modifications until settling on a final prototype. The inital design had both an Arduino Uno, a Teensy 3.6 and no RaspberryPi (RPi) for robot control. The final prototype design excluded the Arduino Uno and left the Teensy 3.6 to handle all low-level gear such as motors and sensors, and a RPi 3 was added to host the high-level components and the LiDar.

The schematic of the low-level and high-level architecture can be seen in Figure 2.2 and transcribed in Table A.1. A software architecture for the low-level and high-level layer is exposed in Chapters 5 and 6 respectively. Following is a presentation of the main components used.

3.1.1 GamesOnTrack (GoT) Receiver Subscriber Rev.3

GoT is a system based on a combined ultrasound and radio technology using active beacons (transmitters) or satellites. This is a low-cost solution to global indoor localization compared to optical systems for indoor positioning. It works by having an ultrasonic receiver mounted on the robot - see Figure 3.1 and transmitter beacons mounted on the ceiling. The system is similar in principle to the NAVSAT GPS system.

The advantages of an indoor system using a mix of ultrasound and radio technology are [22]:

- low-cost components;
- low-energy consumption;
- precision down to 10 mm;
- reduced latency due to high sampling rate;
- tracks up to 20 devices;
- not affected by other radio frequencies;
- distance for precise measurement is 8 meter between transmitter and receiver.

The system suffers from ultrasonic interference and hence any electro-magnetic obstacle in the line-of-sight of the transmitters distorts the measurements. The types of errors due to interference are systematic errors - have the same amplitude in the same areas. A solution to solve for the systematic errors is presented in Chapter 6.

Figure 3.1: The GoT Receiver Subscriber - device listening to the satellites mounted on the ceiling of the AAU MP laboratory.

The raw data received from GoT is a distance to the robot from each beacon installed on the ceiling. There are 5 beacons. The position of each beacon is precisely known. Using the time-difference-of-arrival (TDOA) for both the US and RF signals the distance can be calculated. Furthermore, using the distance to each beacon from the robot, the global position of the robot can be computed. Both algorithms are briefly described in Chapter 5.

3.1.2 Range-finder scanner - RPLidar

The RPLidar is a range-finder scanner based on laser emitter and receiver. RPLidar A1 is a low-cost 5.5 HZ 360° omnidirectional 2D laser scanner with 8000 sample points per second. It is used for the purpose of obstacle avoidance based on laser triangulation ranging principle. Detection range is 12 m. The raw data obtained from the scanner is distance and angle to the obstacle. Through use of ROS RPLidar package a set of points is generated describing all visible obstacles around the robot.

3.1.3 GY-88 IMU - HMC5883L

The magnetometer Honeywell HMC5883L is part of GY-88 IMU board[1]. It combines 5 sensor data to provide the 10 DOF: a 3-axis accelerometer, 3-axis gyroscope, 3-axis magnetometer, barometric pressure, temperature. The board uses the I^2C-bus [2] to interface with Teensy.

In this project, only the magnetometer is used from the GY-88 board in order to measure the orientation of the robot. In order to access the HMC5883L sensor the MPU6050 has to be brought up at every access, hence it is important to initialize all three sensors. The magnetometer is required to be calibrated due to the geographical location and magnetic interference in the environment. Magnetometer calibration is needed in order to eliminate the effects of the interference. As robot orientation is crucial for navigation, a magnetometer must be calibrated before reading measurements.

Interference is observed by plotting the measurements. In order to check whether the magnetometer is affected by interference or fabrication defects, the values read should draw a perfect circle. There are two types of interference: hard and soft iron. Hard iron effects refer to the noise sources from the circuit itself or rather fabrication defects. Hard iron effects shift the origin of the circle (2D) or sphere (3D) drawn by the measurements. Soft iron effects come from objects surrounding the magnetometer that distort the magnetic field. These effects stretch and tilt the circle/sphere by making it look like an ellipsoid. If the magnetometer measurements suffer from interference, the algorithm for calibration is described in Appendix F.

3.1.4 Motors and Encoders XH-500-37D

The two wheel motors with encoders attached have the following technical specifications: 350 RPM, 1:30 gear ratio, 64 CPR encoder. Each of the specifications is important for the calculation of the ticks. Ticks read are used for odometry. 350 RPM refers to the

[1]Inertial Measurement Unit - can provide 6, 9 and 10 axis for measurement meaning it offers 10 degrees-of-freedom (DOF)

[2]Inter-Integrated Circuit (IIC) is a communication protocol between multiple slave devices and one master. Teensy acts as the master in this transmission and the IMU board the slave device.

number of revolutions per minute executed by the gear, with a ratio of 30 gear motion per one revolution. 64 CPR refer to cycles per revolution and is strictly related to the count of the ticks: at the falling edge or rising edge of the pulse.

The type of encoders attached to the motors are **quadrature phase encoders** that have 2 channels a and b producing 90 degrees out-of-phase waveforms. These type of encoders are used because the direction of spin can be calculated easily depending on the leading signal: if channel a leads then the motor rotates clockwise, if b is leading then it rotates counter-clockwise. Knowing the spin direction allows knowing if the robot moves forwards or backwards. This type of encoder with 2 signal waveforms is shown in Figure 3.2.

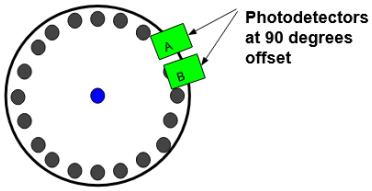


Figure 3.2: Illustration of a quadrature phase encoder and the 90 degree out-of-phase photodetectors necessary to detect motor direction. Inspired from source: [23]

Encoders are read using the interrupt capability of the micro-controller in order to allow other programs to run while reading the encoder. The interrupt function can be called for one or both signals of the encoders - the encoder's specification giving the number of counts per revolution, as seen in Chapter 1, 64 CPR refers to both channels. If only one channel is connected to the interrupt pin, the cycles per revolution is halved.

The algorithm for calculating local robot position and for counting the number of ticks is shown in Chapter 5.

3.1.5 Teensy 3.6

A Teensy microcontroller has been chosen to support the low-level architecture instead of an Arduino Uno. As mentioned above, the prototype design has changed few times and the choice for Teensy3.6 was based on the number of serial ports available to connect all needed gear. Table A.1 transcribes the pins used.

3.1.6 RaspberryPi 3

RPi3 hosts the high-level software implementation together with the LiDar. It is also connected to Teensy3.6. LiDar is connected to RPi3 because of the ROS RPLidar library interfacing with ROS which runs on RPi3. RPi3 executes *ros_core* and hosts the ROS_MASTER and all nodes allowing communication with all robot components through Teensy3.6. A remote workstation is also connected to ROS_MASTER in order

to run data visualization programs such as RViZ that are not possible to run on RPi3. The workstation sends commands to be executed by ROS on RPi3. The configuration between RPi3 and the workstation is presented below.

Configuring RPi3 and the Remote Workstation

The RPi computer is set-up to use an OS image from Ubiquity Robotics [24]. It comes pre-installed with Ubuntu 16.04, ROS Kinetic and a wireless access point (AP) that allows connecting to the workstation without an internet connection. Following the documentation from Ubiquity the only change required on the Raspberry Pi is to change its hostname, which for this project is set as `pi3`. AP comes as an alternative to using a Wi-Fi connection between the two - due to AAU Wi-Fi network security configuration, the RPi and the workstation were not able to 'see' each other on the network.

The remote workstation is a mobile PC running Ubuntu 18.04 and ROS Melodic installed according to the ROS documentation [25]. For this purpose RPi3 is designated as `ROS_MASTER` by default so the remote workstation is configured to know the address of the master on the network.

To verify both RPi3 and the workstation are connected to the same network append the next command to the end of `/.bashrc`:

```
export ROS_MASTER_URI=http://pi3:11311
```

Finally, the robot can be seen in Figure 3.3. This represents the working prototype not the final design which should contain a container for carrying the production pieces from station to station as depicted in Figure 2.1.

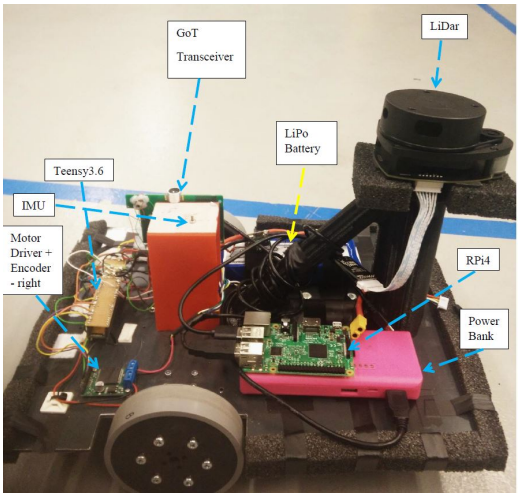


Figure 3.3: The working prototype of the robot - all important components and sensors can be seen in the image.

3.2 Software

This section gives a brief introduction to the main concepts in ROS and Gazebo used later in the project.

The software framework used in this project is composed of:

- Robotic Operating System (ROS)
- Gazebo multi-robot simulation environment

As the name suggests ROS represents a framework of tools, libraries and communication standard for different types of robots not only mobile robots. ROS has two main levels of concepts: the filesystem level and the computation level. The concepts from ROS used in this project explained below are:

- Computation Graph Level
  - Nodes
  - Messages
  - Topics
  - Transforms
  - Bags
- Filesystem Level
  - Package
  - URDF file
  - Launch file

3.2.1 Computation Graph Level

This level describes the exchange of data through messages between different types of ROS structures, i.e. nodes and topics. A general schematic of ROS message communication among nodes, topics and Master is shown in Figure 3.4. It is important to understand the different types of messages each structure accepts and outputs so that the structures are linked.

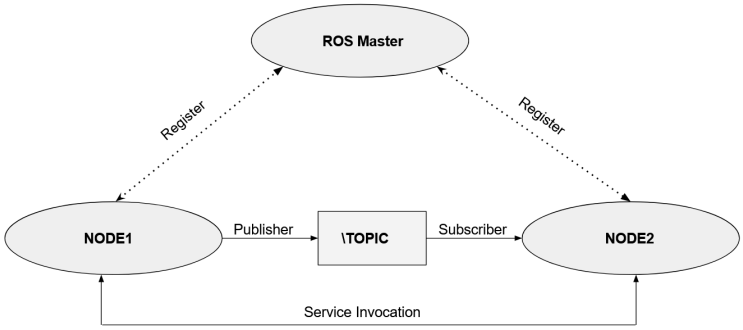


Figure 3.4: Schematic of ROS message communication between nodes and topics. The ROS Master supervises the communication. Source: inspired from [26].

Nodes are structures or executable files that perform calculations. In a robotic solution there are many nodes: one node for the laser range-finder, one for the wheel encoders, one for robot localization and so on. A node can be a subscriber (receives data) to a topic publishing data and a publisher (sends data) to a topic which other nodes can use.

Messages are data structures passed between nodes. The data structure consists of primitives - integer, boolean, floating point, but it can also be a combination of structures. For a mobile robot a typical data structure is $geometry\_msg$ which comprises several message types among which the most important for this project are $Pose$, $Twist$, $PoseWithCovariance$, tf. Other data structures such as nav_msgs create different message types to handle GPS information.

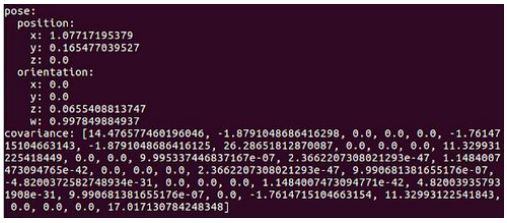


Figure 3.5: Content of ROS message type PoseWithCovariance.

A $Pose$ type of message is a combination of two other message types $Point$ defining 3 floats - x, y and z of the robot and $Quaternion$ defining the orientation in quaternion form by 4 floats: x, y, z, w. The construction of such message types is shown in Listing 3.1. $PoseWithCovariance$ is also a combination of message type $Pose$ and a primitive $float$ defining a 6×6 covariance matrix. Its definition is shown in 3.1 and an example of its content can be seen in Figure 3.5. $Twist$ messages are used for linear and angular velocities - defined as shown in Listing 3.1 from $Vector3$ type messages.

```
# Point message structure
float64 x
float64 y
float64 z

# Pose message structure
```

```
geometry_msgs/Point position
geometry_msgs/Quaternion orientation

# PoseWithCovariance$ message structure
geometry_msgs/Pose pose
float64[36] covariance

# Twist message structure
geometry_msgs/Vector3 linear
geometry_msgs/Vector3 angular
```

Listing 3.1: ROS message types definition

Topics are a container of one message type which name indicates the content of the messages. Nodes subscribe to a topic to get messages with values from ROS processes such as nodes. A topic contains only a type of message. Nodes also use topics to publish messages of a single type from computations. A node can be both a publisher and a subscriber to different topics.

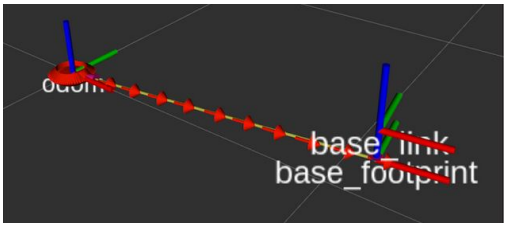


Figure 3.6: The tf tree of the 2D mobile robot. It contains the *odom* frame, *base_link* and *base_footprint*.

Nodes and topics can be seen using a GUI for visualizing the computation graph level for a robotic system - *rqt_graph*. In a graph generated by *rqt* nodes are encircled while topics are mentioned on top of the arrows; the direction of each arrow indicating whether the node is a publisher or a subscriber to the topic. In Appendix G an *rqt_graph* of the simulated robotic system used in this project is shown. In the figure can be seen that Gazebo - the simulation environment becomes a node in the ROS environment both receiving and publishing data for other ROS nodes.

All nodes and topics communicate and are supported by a master programme *roscore* which has to be initiated before any process can take place. *roscore* is a collection of nodes, services and parameters to allow creation of nodes and topics as well as support inter-communication [27].

In a robotic system there are many **coordinate frames** each relative to each other and in order to calculate the position and orientation of each component in the system. ROS transforms are defined according to REP103 - Standard Units of Measure and

Coordinate Conventions and REP105 in naming, units and frame conventions [28]. In short, all coordinate frames in ROS are 3D and right-handed meaning positive x-axis in front, y-axis to the left and z-axis upwards. The tf library is the tool to keep track of the frames and transforms. Each robotic system has a tf tree which for this project can be seen in Figure 3.6. The rqt_graph for the frames can be seen in Appendix H. Most important frames related to this project are: map, $odom$, $base\_link$, $base\_footprint$, $base\_laser$.

- map: static global frame equivalent to world frame
- $odom$: robot static frame
- $base\_link$: robot moving rotational frame
- $base\_footprint$: robot moving translational frame
- $base\_scan$: sensor static frame

The information extracted from these transforms help track the robot and obstacles.

- map -> $odom$: tracks robot's global pose
- $odom$ -> $base\_link$: tracks robot's local pose
- $base\_link$ -> $base\_scan$: used for obstacle avoidance
- $base\_footprint$ -> $base\_link$: reference relationship between robot center of mass and ground. This distance can be seen in Figure 3.6 between the two frames.

A transform is defined as a rotation and translation by [29]. There are frames that only rotate in reference to others, $base\_link$ -> $base\_link$, and others that are only translated in reference to other, $base\_link$ -> $base\_scan$ - see Figure 3.7 (b). According to [29] the transform to get from one frame to another is done according to Eq.3.1.

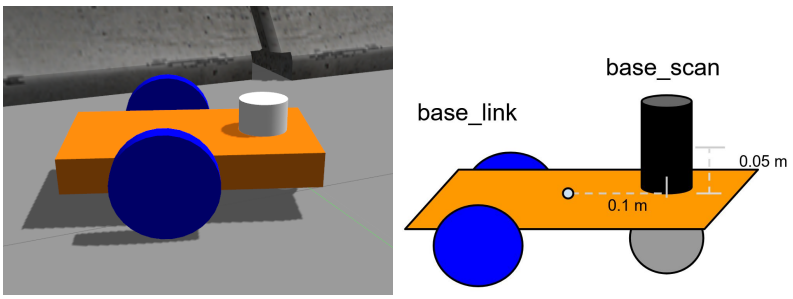
$$T_a^c = T_a^b * T_b^c \tag{3.1}$$

ROS bags are containers for saving and playing-back the ROS messages registered during robot motion. This tool subscribes to the specified nodes and saves the messages transmitted during a process. This represents a way to perform hardware-in-the-loop (HITL) simulations of the real robot system by feeding this data to localization algorithms and mapping.

3.2.2 Filesystem Level

The filesystem level contains files with meta-data about the robot, the libraries used, the inter-dependencies, etc. **Packages** represent the main units of software in ROS. A package creates one or more nodes, contains the configuration files, the libraries and dependencies, launch files, parameters, etc. An example of package used in this project is *robot_localization* providing sensor fusion functionality for the robot.

When working with more than one node initializing each node is time-consuming, however a ***.launch*** file is created to initialize all specified nodes and parameters. A launch file is written in XML and is a target for the package *roslaunch*. The *roslaunch* package has to find the requested package from the launch file and bring up the requested node with the specified parameters. Since packages can create more nodes, a *.launch* file is created for the package being initialized as shown in Listing 3.2.



(a) Simulation robot model (b) Robot frames

Figure 3.7: (a):The simulated 2D mobile robot to be used in the Gazebo environment. The URDF file displays the base, wheels, the caster wheel and the LiDar. (b): Two frames *base_link* and *base_scan* separated by a translation. This translation is important for obstacle avoidance of the robot.

Main packages used in ROS for this research are: *robot_localization*, *hector_slam*, *gazebo*, *turtlebot*3, *move_base*.

- *robot_localization*: performs sensor fusion;
- *hector_slam*: performs EKF SLAM;
- *gazebo*: allows ROS nodes to communicate with Gazebo simulation;
- *turtlebot*3: simulates TurtleBot3;
- *move_base*: robot navigation.

To simulate the real robot into Gazebo a **Unified Robot Description Format (URDF)** file has to be created. An URDF file is an XML description of the robot components -

model and dimensions, kinematic or dynamic model and sensors attached. Sensors are simulated in Gazebo using the URDF file. The sensor location on the robot is specified as well as its technical specifications. URDF files are used for visualizing, adding physical properties and motion of a robot in a simulated environment, i.e. Gazebo.

```
$ roslaunch package_name file.launch

#example for package robot_localization
$ roslaunch robot_localization ekf_node.launch
```

Listing 3.2: Package launch file command

All components of the robot are considered **links** and are described with the corresponding coordinate frame in reference to the robot's coordinate frame. After modelling all robot's components - i.e. wheels, frame, sensors, the connections between the moving links of the robot are modelled as **joints** - translational or rotational. Links and joints of the simulated 2D mobile robot are shown in Appendix I.

The Gazebo simulation environment communicates with ROS nodes sending and receiving measurements. Gazebo displays and simulates the robots created in the URDF files commanded by the model and control in launch files. The simulated 2D mobile robot is shown in Figure 3.7.

4 SLAM and Robot Navigation

This chapter introduces the reader to the main idea of the SLAM algorithm. This is meant to aid the understanding for later in the chapter when the Robotic Operating System (ROS) is introduced and SLAM is performed using ROS. The sensor fusion as an EKF methodology is presented at the end of the chapter.

4.1 Robot Localization

The section gradually presents the main concepts that build the SLAM algorithm from robot localization, mapping and combining them into SLAM.

As mentioned before, SLAM is a methodology for performing localization and mapping simultaneously. There are different algorithms for achieving SLAM. In this project, SLAM is achieved by using EKF. The lead robot needs to perform EKF SLAM to improve the localization estimation however the client robot uses only sensfor fusion based on EKF as it receives the map from the lead robot. Both SLAM and sensor fusion are methodologies based on EKF in this project. Hence, with a known environment (map), the robot does not need SLAM but only localization. Another case is when the map is unknown but the robot can localize itself and can start mapping the environment. This chapter presents 3 important concepts:

1. Robot Localization;
2. Robot Mapping;
3. SLAM;
4. Sensor Fusion.

4.1.1 Hidden Markov Models and Recursive Bayesian Filter

Robot localization is a state estimation problem hence probabilistic localization algorithms are based on the Bayes filter. One application of the Bayes filter for the localization problem is called *Markov Localization* [18].

Markov localization is based on Markov Chain for fully observable states or Hidden Markov Model (HMM) for partially observable states. HMM describes a sequence of states where the probability of each state at time t depends only on the previous state at time $t-1$ [18]. It assumes there is a hidden state x that cannot be directly measured but can be estimated by observing another random variable y related to x. Transition probabilities refer to the change from one state to another.

The transition state matrix is build when probabilities of changing from one state to another are known. The statistical properties of the model's state can be predicted. The Kalman filters are models where the Markov process over the hidden states are linear or non-linear dynamical systems where inference is possible.

There are 3 important quantities used to estimate robot's location: x_t, u_t and z_t. Although the quantities are deterministic in nature, robot localization is probabilistic. This is because the pose of the robot cannot be measured directly (HMM) - it must be inferred from different sources of data [18]. These sources are: x_{t-1}, u_t, z_t. The fact that state x_t is generated from x_{t-1} makes the filter recursive. Hence, localization is determined through a probabilistic distribution (posterior) conditioned on all past states of x_t, the control input u_t and measurements z_t taking the form: $p(x_t|x_{0:t-1}, z_{1:t-1}, u_{1:t})$. There are two conditional probabilities resulting:

- State Transition Probability: specifies how the state changes with each control input - written as $p(x_t|x_{t-1}, u_t)$ [18].
- Measurement Probability: specifies that state x_t is sufficient to generate measurements z_t - has the form of $p(z_t|x_t)$ [18].

The Bayes filter has 2 essential steps: prediction and measurement update. Each of the two probabilities mentioned above are used in these steps. Initial probability $p(x_0)$ is a third very important probability needed for the implementation of the filter as it centers all probability on the correct value around x_0 and 0 everywhere else [18].

To explain the 2-steps algorithm prediction-measurement update, the filter family is narrowed down to Gaussian filters and more specifically to the Kalman filters: the Kalman Filter (KF) and Extended Kalman Filter (EKF). The two filters are related, KF handles linear system and EKF handles non-linear systems. Both filters are described in detail in Appedix B.

Gaussian filters assume Gaussian white process noise hence using normal distributions where x_t is a state vector. Normal distributions over vectors are called multivariate distributions and their density function is of the form[18]:

$$p(x) = \det(2\pi\Sigma)^{-\frac{1}{2}} \exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)) \tag{4.1}$$

In Eq. 4.1 μ is the mean vector, Σ is positive semi-definite, quadratic and symmetric, co-variance matrix. An advantage of using normal distributions is the uni-modality - meaning that there is a single maximum around the true state with a small margin of uncertainty [18] - unimodality can be seen in Figure B.1 a). The mean and the co-variance represent the parameters of the posterior and these two value are estimated by the filters.

4.1.2 State Vector and Environment

Robot localization refers to estimation of its coordinates in relation to an external reference frame. According to [18] the robot needs a map known *a priori* to determine its position relative to the map given its perception of the environment and motion. A map is given in global coordinates and localization is the process of mapping the robot's local coordinate to the global one. The robot and the environment is described by a state. According to [18], a state is a collection of robot and environment aspects (variables) that are static, i.e. walls, or dynamic, i.e. robot pose, people in environment, robot velocity, etc. A source of information regarding its position is given by the robot motion. A kinematic model of the model gives information on how much has moved since its last location. Other relevant sensors are environment perception sensors such as: LiDar, ultrasonic-based range finder, etc.

If x_t denotes a column vector containing all relevant states for the robot and its environment, typical state variables used are:

- robot *pose*: differs in size depending on the rigid body, but for a 2D mobile robot the pose is composed of 3 variables: 2 Cartesian coordinates x and y and 1 angular orientation: $yaw(\theta)$.
- object location and features of the robot's environment: the state variables for such objects can be their coordinates x and y, as well as features such as color, texture, etc. Landmarks are static, distinct and robust objects to be recognized in the environment.
- sensor state: communicates whether the sensor is faulty, broken or battery level, etc.

The state vector x_t for a mobile robot has a minimal dimension of 3×1 representing the robot's pose, however this can grow very fast when adding environment states as in the example below where the state vector contains the robot's pose and the coordinates of 2 landmarks.

$$x_t = \begin{bmatrix} x \\ y \\ \theta \\ x_{landmark_1} \\ y_{landmark_1} \\ x_{landmark_2} \\ y_{landmark_2} \\ \vdots \end{bmatrix}, \quad u_t = \begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix}, \quad z_t = \begin{bmatrix} z_1 \\ \vdots \\ z_k \end{bmatrix} \tag{4.2}$$

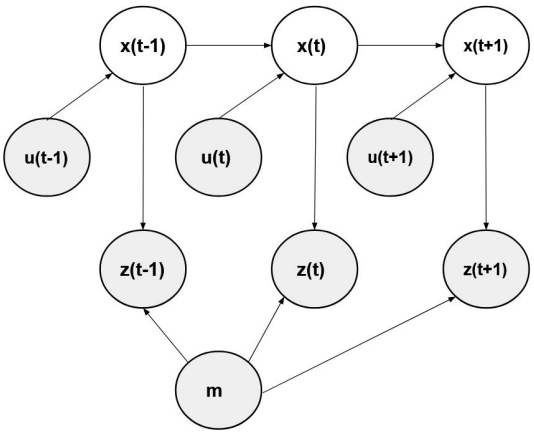


Figure 4.1: Graph of a Bayesian network of HMM. Robot localization estimation based on known data as gray circles, and results from computation - white circles. Inspired from source: [18].

Graphically the mobile robot localization problem is shown in Figure 4.1 - grayed circled represent known data and white circles represents estimation. The arrows going into a circle mean that in order to compute the respective variable it needs the input from the arrows. Arrows going out is the result of a computation or measurement feeding into another variable.

Environment Interaction

The robot can interact with its environment in two main ways:

- taking measurements of the environment's state z_t;
- modifying environment's state through actuation u_t.

The focus is on measurements of the environment's state by means of relevant sensors (LiDar, ultrasonic range finder, camera, bumpers). Measurements or observations can be: range scans, images, tactile data.

Actuation of the robot assumes control actions that change the state of the environment. In mobile robotics, an example of actuation is the control input to the robot to move few centimeters(cm) in the environment [18]. The change of state is known through the control input u_t at time t to move i.e. 10 cm/seconds(s) for 10 s it is expected to see the robot at a new distance of 100 cm.

Odometry is an alternative source of control data [18]. Odometry is computed from wheel encoders that count the revolutions of the robot's wheels. In localization problems

it is common to use odometry as a control input since it measures the effects of a control input [18].

4.1.3 Robot Perception

Perception models the environment on which measurement z_t is based. As there are a variety of sensors that model the environment - range sensors, cameras, tactile sensors, barcode detector etc., the model of the environment is specific to the sensor available. As mentioned in Section 3.2, the environment-measuring sensor available is a LiDar. A laser-based range scan works by emitting a light-beam and recording its echo. Measurements are based on time-of-flight and light beams are spaced out at one degree increment.

To generate measurement of the environment, a map is needed. A map is a list of the landmarks of the environments [18]. A map also specifies the location and other features of the landmarks. Eq. C.1 shows the notation of a map m and each component is a landmark [18]. Landmarks are used for robot navigation. They represent features of indoor environments: doors, wall corners, etc. and for outdoor environments: building corners, trees, etc.

$$m = \{m_1, m_2, \dots m_N\} \tag{4.3}$$

It was mentioned before that in robot localization, the map is known and 'given' to the robot as a ground-truth. Even more, there is assumed that a correlation function exists between the map and the local map created by the scans of the range scanner.

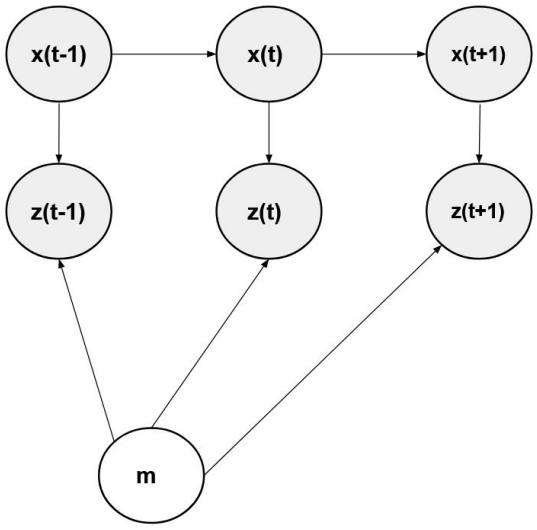


Figure 4.2: Graph of mapping when robot pose is known. Grayed circles variables are known. Inspired from source: [30]

This technique is called map matching or scan-to-map as it was introduced in Chapter 1 in Figure 1.4. A number of consecutive scans are compiled into a local map m_{local} and compared with the known global map m, having a higher probability that the two match if $p(m_{local}|x_t, m)$ is high. This technique has the ability to transform scans into occupancy maps.

Occupancy maps is part of robot mapping process not robot localization. Scan-to-map is still a robot localization technique that completes the state vector with environment variables. Scan-matching or scan-to-map uses the likelihood fields to maximize the likelihood of the current pose and map relative to the previous pose and map. Then pose correction is done following Eq. C.2 where $p(z_t|x_t, m_{t-1})$ is the current measurement, m_{t-1} is the map constructed so far and $p(x_t|x_{t_1}, u_{t-1})$ is the pose with u_{t-1} as the motion. The algorithm for computing the the posterior $p(z_t^k|x_t, m_{t-1})$ is detailed in Appendix C.

$$x_t = \underset{x_t}{argmax}\{p(z_t|x_t, m_{t-1}) \quad p(x_t|x_{t_1}, u_{t-1})\} \tag{4.4}$$

4.2 Robot Mapping

In this section the process of building a map is presented assuming the robot knows its location on the map.

Even when the position of the robot is known, there are difficulties mapping an environment if:

- it is a large environment;
- noise in sensors and actuators;
- similarity across places in the environment;
- missing to close loops around the environment.

To continue exposing the mapping problem in a graph similar to Figure 4.1, this can be seen in Figure 4.2. As mentioned before, the grayed circles represent the knows data used to computed the variables in the white circle - here, the map.

The Occupancy Grid Mapping Algorithm

There are many types of mapping algorithms but the one used in this project and predominantly in robotics is the occupancy grid mapping algorithm. According to [18] the occupancy grid represents a field of binary random variables arranged in an evenly space grid. The goal of the algorithm is to calculate the posterior $p(m|z_{1:t}, x_{1:t})$ where:

- m stands for the map;

- $z_{1:t}$ measurements up until time t;
- $x_{1:t}$ path taken through robot poses.

An occupancy grid map partitions the space into grid cells m_i, where each m_i has attributed a binary occupancy value - 1 for occupied and 0 for free. Hence, probability $p(m_i = 1)$ refers to the probability that the cell is occupied [18]. The problem with finding the posterior $p(m|z_{1:t}, x_{1:t})$ is its dimension. A map can have tens of thousands grids and the total number of maps that can be represented from the cells is 2^{cells} [18]. As this is not possible to calculate, the problem is broken down into calculating $p(m_i|z_{1:t}, x_{1:t})$ - the probability of each grid cell conditioned on the measurements and pose. As such the entire map posterior is calculated as the product of its marginals [18].

$$p(m|z_{1:t}, x_{1:t}) = \prod_i p(m_i|z_{1:t}, x_{1:t}) \tag{4.5}$$

A grid map is better than a blueprint of the environment because it also presents the obstacles and objects in the environment besides walls and doors. In Figure 4.3 it can be seen an occupancy grid map of a simulated environment in Gazebo. The map was created for the robot to navigate to build the map. The gray level in the map specifies the occupancy posterior over the grid - the darker the cell the more likely to be occupied [18]. While the map is probabilistic in nature, due to the measurements and motion, the values tend to the extremes - 1 or 0 [18].

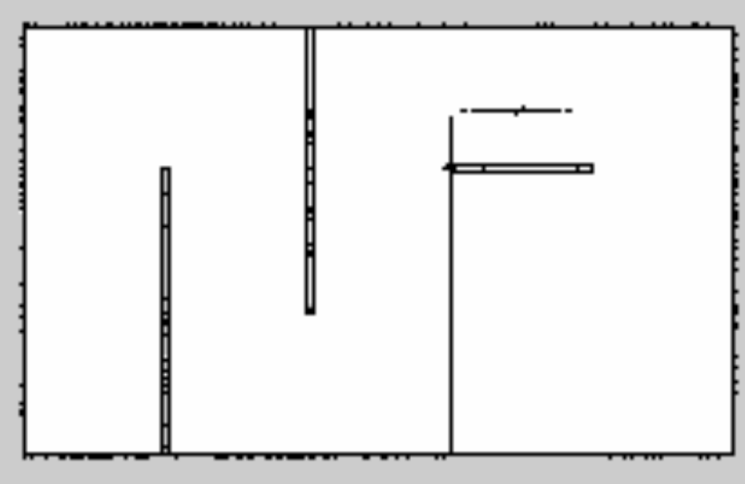


Figure 4.3: Occupancy Grid map saved from the robot simulation environment - Gazebo.

.

An occupancy grid map provides a cleaner perspective of the environment than a map build from LiDar observations. A map build using raw LiDar measurements is polluted

with points also representing dynamic obstacles i.e. people, which the robot will try to avoid when navigating. The occupancy grid algorithm iterates through each map cell and calculate the posterior $p(m_i|z_{1:t}, x_{1:t})$ clearing the map of obstacles that are not actually there.

The caveat to remember is the different frames used to build the map. More exactly, the LiDar's data is obtained in local coordinates, however in order to be used in the occupancy grid algorithm it needs to be in global coordinates. If the robot pose is $x_t = [x \quad y \quad \theta]^T$ and grid cell m_i is defined by $(x_{m_i}, y_{m_i})^T$ for a landmark-based map or polar coordinates $(r_t^i, \phi_t^i)^T$ for a 2D laser line map, then the coordinates of the grid cell in robot frame is given by Equation 4.6 [18].

$$\begin{pmatrix} x_{m_i}^{\{R\}} \\ y_{m_i}^{\{R\}} \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_{m_i} - x \\ y_{m_i} - y \end{pmatrix} \tag{4.6}$$

4.3 Simultaneous Localization and Mapping - SLAM

SLAM is a difficult problem to solve compared to robot localization or robot mapping as SLAM has to provide a solution to both problems at the same time. There are mainly 2 types of SLAM algorithm categorized on the type of result obtained:

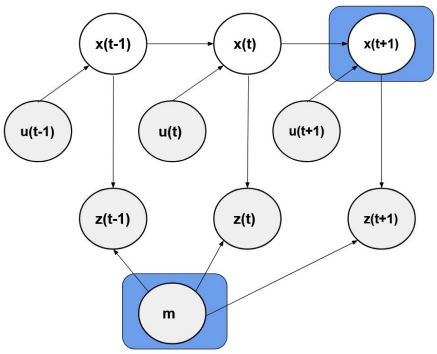


Figure 4.4: Graph of the results from solving the online SLAM algorithm: the current posterior of the pose and map at current time t. Source: [18].

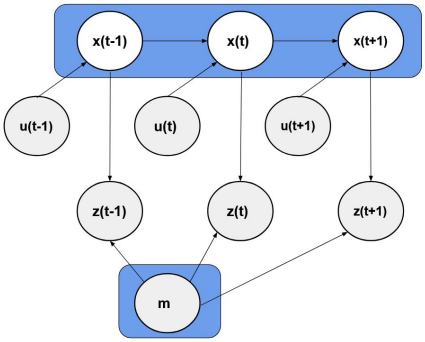


Figure 4.5: Graph of the results from solving the full SLAM algorithm: the entire posterior of the path over the map. Source: [18].

- online SLAM: computes the posterior of the current pose and map at current time t: $p(x_t, m|z_{1:t}, u_{1:t})$ - see Figure 4.4.
- full SLAM: calculate the entire posterior along its path over the map: $p(x_{1:t}, m|z_{1:t}, u_{1:t})$ - see Figure 4.5

According to [18] both types of SLAM algorithm are used in the robotics industry, but the online SLAM is used in this project. There many approaches to SLAM among which the following are most used:

- Extended Kalman Filter SLAM (EKF SLAM)
- Particle Filter SLAM (FastSLAM)
- GraphSLAM

EKF SLAM applies EKF to the online SLAM. EKF SLAM is delimited by the following assumptions [18]:

- builds feature-based maps;
- Gaussian noise in motion and perception;
- linear and linearizable NL systems.

The EKF SLAM estimates the robot pose and the features of all landmarks met along the path. Defining y_t as the combined state vector comprised of the robot pose and landmarks features, it is described in Eq.4.7. The dimension of this vector grows according to $3 + 2N$, where N is the number of landmarks observed. Hence, EKF SLAM computes the posterior $p(y_t|z_{1:t}, u_{1:t})$.

$$y_t = \begin{pmatrix} x_t \\ m \end{pmatrix} = (x \quad y \quad \theta \quad m_{1,x} \quad m_{1,y} \quad m_{2,x} \quad m_{2,y} \quad \dots \quad m_{N,x} \quad m_{N,y})^T \tag{4.7}$$

Initialization of the algorithm assumes to be at the origin of the coordinate system. The features of the landmarks are not known initially and are initialized as infinite. The initial mean and covariance is defined as Eq. 4.8 and 4.9. The dimension of the covariance matrix is $(3 + 2N) \times (3 + 2N)$. The first three columns are for the robot pose and the rest are initialized for the features of the landmarks.

$$\mu_0 = (0 \quad 0 \quad 0 \quad \dots \quad 0)^T \tag{4.8}$$

$$\Sigma_0 = \begin{pmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 \end{pmatrix} \tag{4.9}$$

As the robot moves and uncertainty is increasing in its position estimate as well as the uncertainty of the landmarks position, the covariance matrix Σ_t is updated according to the equations of the EKF. The covariance matrix Σ_t is shown in Eq. 4.10 where robot's pose covariance is highlighted with red and the landmarks position is highlighted in blue. The rest of the matrix represents correlations between robot's pose and landmarks positions. All these correlations are 0 at the start of the algorithm. As the robot moves through the map, estimating its pose and observing landmarks, the uncertainty of its position and of the landmarks grows. When the robot re-observes a landmark the uncertainty of that position estimates becomes small as well as for the position of the observed landmarks.

$$\Sigma_t = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xm_{1,x}} & \sigma_{xm_{1,y}} & \cdots & \sigma_{xm_{N,x}} & \sigma_{xm_{N,y}} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} & \sigma_{ym_{1,x}} & \sigma_{ym_{1,y}} & \cdots & \sigma_{ym_{N,x}} & \sigma_{ym_{N,y}} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta\theta} & \sigma_{\theta m_{1,x}} & \sigma_{\theta m_{1,y}} & \cdots & \sigma_{\theta m_{N,x}} & \sigma_{\theta m_{N,y}} \\ \sigma_{m_{1,x}x} & \sigma_{m_{1,x}y} & \sigma_{\theta} & \sigma_{m_{1,x}m_{1,x}} & \sigma_{m_{1,x}m_{1,y}} & \cdots & \sigma_{m_{1,x}m_{N,r}} & \sigma_{m_{1,x}m_{N,y}} \\ \sigma_{m_{2,y}x} & \sigma_{m_{2,y}y} & \sigma_{\theta} & \sigma_{m_{1,y}m_{1,x}} & \sigma_{m_{1,y}m_{1,y}} & \cdots & \sigma_{m_{1,y}m_{N,x}} & \sigma_{m_{1,y}m_{N,y}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sigma_{m_{N,x}x} & \sigma_{m_{N,x}y} & \sigma_{\theta} & \sigma_{m_{N,x}m_{1,x}} & \sigma_{m_{N,x}m_{1,y}} & \cdots & \sigma_{m_{N,x}m_{N,x}} & \sigma_{m_{N,x}m_{1,y}} \\ \sigma_{m_{N,y}x} & \sigma_{m_{N,y}y} & \sigma_{\theta} & \sigma_{m_{N,y}m_{1,x}} & \sigma_{m_{N,y}m_{1,y}} & \cdots & \sigma_{m_{N,y}m_{N,x}} & \sigma_{m_{N,y}m_{N,y}} \end{pmatrix} \tag{4.10}$$

When a landmark is re-observed - otherwise known as loop closure, and the uncertainty of the estimations decreases, the correlation between robot's pose and landmarks decreases through data association. If loop closure does not happen, the uncertainty continues to grow making the algorithm less accurate in estimation. Loop closure is a problem when the robot has moved through a long path and does not recognize a previously visited area. Errors in the SLAM solutions rise exponentially if the wrong data association is made. Few remarks can be drawn:

- the determinant of any sub-matrix under the map covariance matrix (blue highlight in Eq. 4.10) decreases as new measurements are done;
- when a new landmark is observed, its uncertainty is maximum;
- landmark uncertainty decreases as new measurements are done.

The derivation of the EKF SLAM includes the kinematic model of the robot explained in the next chapter. The only difference stands in using the combined vector y_t instead of x_t. Changes in the mathematical formulations in EKF SLAM from EKF are due to manipulating an extended vector. After the initialization of the robot in SLAM using Eq.4.8 and 4.8, the motion updates the combined vector to include the velocity model derived in Section 5.1.1. The combined vector for EKF SLAM is given in Eq.4.11 [31].

$$y_t = \begin{pmatrix} x \\ y \\ \theta \\ m_1 \\ m_2 \\ \vdots \\ m_{N-1} \\ m_N \end{pmatrix} = y_{t-1} + \begin{pmatrix} \frac{\phi_r r + \phi_l r}{2} + \cos(\theta_{t-1} + \frac{\phi_r r - \phi_l r}{2d}) \\ \frac{\phi_r r + \phi_l r}{2} + \sin(\theta_{t-1} + \frac{\phi_r r - \phi_l r}{2d}) \\ \frac{\phi_r r - \phi_l r}{d} \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \tag{4.11}$$

To summarize a lengthy section, SLAM is an algorithm to recover the robot's path x_t and a map of the environment from odometry u_t and measurements z_t. In this project, it is sought to improve the process model of the robot by sensor fusion. Several sensors that provide both new type of data about the robot's pose as well as redundant data are used.

4.4 Sensor Fusion

This section presents the EKF from a sensor-fusion approach. The algorithm of the filter remains the same as EKF only the perspective is changed to suit the need of fusing different sensor information to render a more accurate input to SLAM.

In this project the online SLAM used is based on a EKF taking as input odometry as control variable and LiDar as localization and mapping measurements. As mentioned before, odometry is inherently inaccurate for long-term localization of the robot. Sensor fusion is used to improve the control input to SLAM by fusing together odometry (velocity) with other available sensors such as GoT (global positioning) andor IMU (acceleration). The same can be done for the measurement variable where LiDar can be fused with other environment measuring sensors such as RGB camera. ultrasound beacons, etc.

Sensor fusion is used to improve the control input to SLAM for the robot localization part of SLAM. While the SLAM state vector x_t to be determined was composed of the robot pose and the landmark positions on the map, in sensor fusion the state vector y determines robot's pose, velocity, andor acceleration depending on the sensors available. For a 3D rigid body the state vector can cover all DOF as described in Eq.4.12 with 15 or even 18 states while for a mobile robot Eq.4.13 describes its 2D motion in 6 states.

$$\begin{aligned} y_{3D} = [x \quad y \quad z \quad \phi \quad \psi \quad \theta \quad \dot{x} \quad \dot{y} \quad \dot{z} \quad \dot{\phi} \quad \dot{\psi} \quad \dot{\theta} \quad \ddot{x} \quad \ddot{y} \quad \ddot{z}]^T \quad where \\ \phi - roll \quad angle, \quad \psi - pitch \quad angle, \quad \theta - yaw \quad angle \end{aligned} \tag{4.12}$$

$$y_{2D} = [x \quad y \quad \theta \quad \dot{x} \quad \dot{y} \quad \dot{\theta}]^T \tag{4.13}$$

The state vector in Eq.4.13 can be determined if sensors are available to provide information about such states. A sensor can directly supply measurements regarding a state, i.e. GoT provides direct information about the x and y coordinates of the robot. Other information such as velocity or acceleration not directly measured can be inferred from streams of data.

The covariance matrix Σ_0 shows the uncertainty in the pose and velocity of the robot. The initialization values of the covariance matrix are based on the sensor's accuracy: if the sensor is accurate then the values corresponding to the state should be low; otherwise the values should be high meaning that the sensor is not to be trusted as much. This matrix also explains the convergence of the filter to the correct measured values. High values help the filter converge relatively fast to the correct values.

As an EKF methodology, sensor fusion also follows the 2 steps: prediction and measurement update. An important component of the prediction step in sensor fusion is still the system model. The system model can be specific to the robot used or can be a generalized system model, i.e. constant velocity model or constant acceleration model briefly described in Appendix D. The odometry model described in Section 5.1.1 can also be used.

The measurements available are selected in the measurement matrix. For a linear system, the measurement matrix C_t looks as in Eq. 4.14 where the measurements come from odometry and IMU:

$$H_t = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{4.14}$$

In this project the global positioning system GoT makes available the position of the robot which can also be included in H_t. In Eq. 4.15 GoT provides information about the robot position on the xy axes.

$$H_t = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{4.15}$$

In sensor fusion there can be redundancy in data provided by sensors however measurements made available through the measurement matrix have to be the same unit and type. Hence if a sensor does not provide direct data for the measured variable then the mapping formula has to be specified.

4.5 Robot Navigation

Robot navigation concerns moving the robot to a desired place using a map. The research on algorithms for moving a robot is extensive and differ on the specifications of the task.

Once the environment is mapped, the robot can move to user-specified points on the map (waypoints) which describe the pose of the robot at the end destination. A path is calculated between the position of the robot and the goal position. This path is set depending on the completeness, efficiency and optimality of the algorithm.

There are several types of navigation algorithms: potential fields, roadmaps, bug algorithms, etc. The bug algorithms are tested in this project. There are 4 main bug algorithms: Bug 0, Bug 1, Bug 2 and Tangent Bug. Each of the algorithms represents an improvement of the other in certain environments. The first three algorithms imply that the robot has a tactile sensor for detecting obstacles. The algorithms describe a simple behaviour: from start position move towards goal and if an obstacle is encountered then follow the obstacle contour until closing in to the goal position.

The difference in behaviour between Bug 0 and Bug 1 and 2 is that it uses no memory of the points visited. The main difference between algorithms Bug 1 and 2 is that Bug 1 uses an exhaustive search to find the optimal point to leave for the goal while Bug 2 uses an opportunistic approach meaning that it takes the first promising leaving point for the goal [32]. Tangent bug builds on Bug 2 and uses a range finder guaranteeing a shorter path to the goal.
Examples of Bug algorithms used on the simulated robot can be found in Appendix J.

5 Low-Level Design and Implementation

Chapter 2 exposes the design of the low-level architecture of the mobile robot. The low-level architecture is supported by a Teensy micro-controller. This platform is connected to the main sensors and gear processing the raw data coming from: IMU, GoT, motor encoders, battery, motors and ultimately the RaspberryPi. The following sections present the main program of the Teensy handling raw data from sensors and transforming it into useful information about robot's position, orientation and motion.

A differential drive 2D mobile platform can translate on 2 dimensions x, y and rotate on one dimension θ. The robot has 3 degrees of freedom (DOF) and its pose is characterized by (x, y, θ). These quantities have to be calculated from the sensors used on the robot.

A kinematic approach to modelling the robot's translational and angular motion is used. This means that its pose is described by a function of wheels' movement and robot's motion disregarding the forces and moments that make it move (dynamics approach).[33]

5.1 Local and Global Robot Positioning

This section explains the algorithms to determine the translational position of a 2D mobile robot using 2 sensors: encoders and ultrasonic global positioning system (GoT).

Encoders are used for local positioning and GoT for global positioning. It follows that 2 noisy sensors providing redundant information can be fused together to output a better estimate of the position of the robot. Sensor fusion as introduced in 4 is a high-level task described in Chapter 6.

5.1.1 Odometry

Odometry or localization-by-odometry is used to calculate the robot's position using the encoders mounted on each wheel. The robot has a diferentially-steered drive system, meaning that each of the 2 wheels can be independently powered and speed controlled providing different steering functions. [33] The logic behind differential-steering drive systems is simple:

1. wheels move with equal speed —> robot moves forward;
2. right wheel moves faster than left —> robot yaws counter-clockwise;
3. right wheel moves slower than left —> robot yaws clockwise;
4. wheels move in opposite directions —> robots spins around its axis.

Local position estimates are based on wheel revolutions and their speed of turning - how many revolutions per second. Wheel revolutions are calculated based on encoder readings (or ticks) - how many ticks per revolution. These ticks depends on the type of encoder and its technical specifications.

As mentioned in section 3.2 from Chapter 1 the number of ticks per revolution depends on the interrupt functions applied. The resolution of the ticks depends on the number of interrupt functions implemented. As there are 2 channels in each encoder and there are 2 encoders, there can be 4 interrupt functions implemented - one for each channel. Good counts can be obtained with one interrupt function - one for both encoders. Listing 5.1 shows how ticks are counted using one interrupt function.

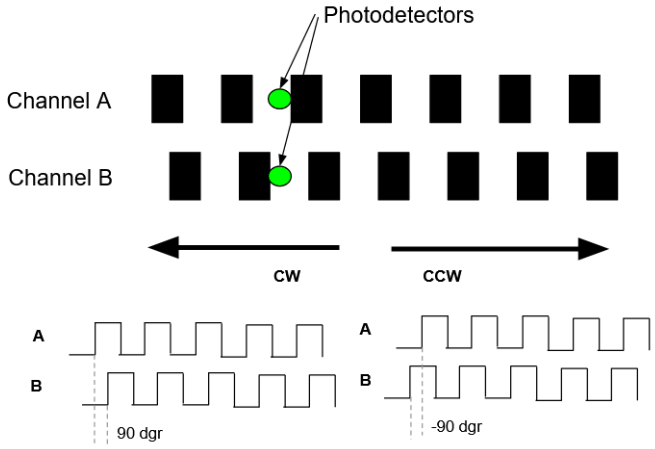


Figure 5.1: Encoder mechanism with photodetectors to detect the direction of spin. A read that channel A leads channel B means wheel spins CW while channel B leading A means wheel spins CCW. Inspired from source: [34].

In Figure 5.1 can be seen the photdetectors at 90° phase shift reading if channel A leads channel B or vice-versa. As can be seen in Listing 5.1 there is a counter for the encoder adding ticks when channel A leads and subtracting ticks when channel B leads. The cumulative value of the counter indicates the angular position of the wheel [34].

```
1
2  // defines interrupt pin for signal A of right-hand encoder
3  #define RH_ENCODER_A 2
4  // defines interrupt pin for signal B of right-hand encoder
5  #define RH_ENCODER_B 3
6
7  // initialize hardware interrupts for one channel
8  attachInterrupt(2, readRightEncoder, CHANGE);
9
10   if (digitalRead(RH_ENCODER_A) == HIGH) {
```

```
    if (digitalRead(RH_ENCODER_B) == LOW) {
        Count++;
    } else {
        Count--;
    }
  } else {
    if (digitalRead(RH_ENCODER_B) == LOW) {
        Count--;
    } else {
        Count++;
    }
  }
```

Listing 5.1: Encoder ticks count for determining motor spin direction. If the count is increasing, then the motor is driving forward and vice-versa.

The Odometry Model

The kinematic model of a robot describes the relation between the pose of the robot and velocity. The speeds of motors can give information about two quantities: rate of moving forward and rate of turn of the robot - these two are the linear and angular speeds, which integrated give the linear and angular position of the robot (x, y, θ_x). Since these two quantities do not have analytical expression to calculate the integral, the values are integrated numerically - divide time in very small intervals in the order of milliseconds and add all the quantities over the period of time. [23]

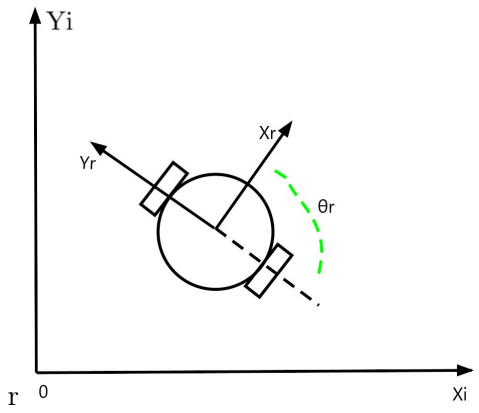


Figure 5.2: The coordinate frames of a differential wheel drive used to derive the forward kinematic model. The i-index stands for inertial and r-index for robot frame.

The odometry model is a kinematic model of the differential drive robot meaning a robot with non-holonomic constraints - the wheel of the robots moves only forwards and backwards without any side-way movement [30]. The model assumes access to odometry

information given by wheel encoders.

There are many approaches to describing the forward kinematics for a 2D mobile robot but in this project the approach is taken by the use of inertial and body/robot frame. Figure 5.2 shows the inertial fixed frame $\{X_i, Y_i\}$ in reference to which the velocity state vector is calculated. The state vector is calculated in the robot frame $\{X_r, Y_r\}$. As the robot frame is attached to the robot, its position is always 0, however the velocities in this frame are of interest as these map the position of the robot in the inertial frame.

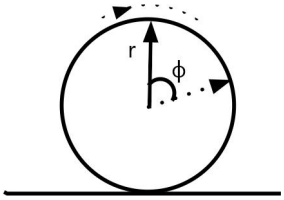


Figure 5.3: The forward velocity on the x-axis of the robot is calculated using the angular speeds of the wheel $\dot{\phi}$ and the radius of the wheel r.

The velocity vector $\dot{X}_r$ in robot frame is composed of the robot velocities on the x and y axis as well as the angular speed θ - rotation about the imaginary z-axis sticking out of the ground. The velocity vector is mapped onto the inertial frame as the state vector X_i through a transform matrix. The two vectors can be seen below.

$$\dot{X}_r = \begin{bmatrix} \dot{x_r} \\ \dot{y_r} \\ \dot{\theta} \end{bmatrix}, \quad X_i = \begin{bmatrix} x_i \\ y_i \\ \theta \end{bmatrix} \tag{5.1}$$

The transformation matrix from one frame to another can be found out by mapping the velocity vector from the robot frame to the inertial frame. Knowing that a movement of on the x-axis in the robot frame has components on both the x- and y-axis in the inertial frame, the velocity vector components in the inertial frame can be written as Eq. 5.2 [30].

$$\dot{X}_i = \begin{pmatrix} \dot{x_i} \\ \dot{y_i} \\ \dot{\theta_i} \end{pmatrix} = \begin{pmatrix} \cos(\theta)\dot{x_r} - \sin(\theta)\dot{y_r} \\ \sin(\theta)\dot{x_r} + \cos(\theta)\dot{y_r} \\ \dot{\theta_r} \end{pmatrix} \tag{5.2}$$

From Eq. 5.2 the transformation matrix can be extracted and vector $\dot{X}_i$ can be written in compact form as Eq. 5.3 [30].

$$\dot{X}_i = {}^{I}_{R}T(\theta)\dot{X}_r \quad where \tag{5.3}$$

$$ {}^{I}_{R}T(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.4} $$

Having the transform from the robot frame to the inertial frame, the velocity vector in robot frame can be calculated considering the degrees of freedom of the wheel - the kinematic constraints. In Figure 5.3 the components needed for computing the forward velocity on the x-axis are shown: $\dot{\phi}$ is the angular speed of the wheel and r is the radius of the wheel. Hence, forward velocity of each wheel is computed as $\dot{x} = \dot{\phi} r$ [30].

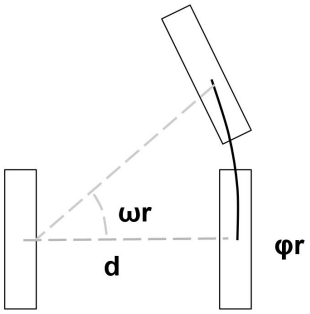


Figure 5.4: The kinematic components of a CCW rotation of the robot. The diameter d and the rotation angle around the left wheel gives the arc length created by the right wheel. Inspired from source: [30]

Anchoring the robot frame at the midpoint between the two wheels, the speed of the robot is expressed as the speed of the midpoint. Hence the speed of the robot on the x_r-axis can be written as Eq. 5.5 where $\dot{\phi}_l$ stands for the angular speed of the left wheel and $\dot{\phi}_r$ for the angular speed of the right wheel respectively.

$$ \dot{x_r} = \frac{r\dot{\phi}_l}{2} + \frac{r\dot{\phi}_r}{2} \tag{5.5} $$

The rotation around the z-axis follows a positive rotation which according to the right-hand rule is CCW. A CCW rotation or a left turn means that the right wheel of the robot moves faster than the left wheel.

This means that the distance travelled by the right wheel is given by the arc length formed equal to the distance between the two wheels - the diameter d times the angle of rotation ω_r around the left wheel. The formula is given in Eq. 5.6. The left turn kinematics can be seen in Figure 5.4. Taking the derivative of Eq.5.6 results in Eq. 5.7 which leads to final Eq. 5.8 of the heading.

$$\omega_r d = \phi_r r \tag{5.6}$$

$$\dot{\omega_r} = \frac{\dot{\phi_r} r}{d} \tag{5.7}$$

$$\dot{\theta} = \frac{\dot{\phi_r} r}{d} - \frac{\dot{\phi_l} r}{d} \tag{5.8}$$

Assembling all state equations so far, the differential state of the robot in the inertial frame is given by Eq. 5.9 [30].

$$\dot{X}_i = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{r\dot{\phi_l}}{2} + \frac{r\dot{\phi_r}}{2} \\ 0 \\ \frac{r\dot{\phi_r}}{d} - \frac{r\dot{\phi_l}}{d} \end{pmatrix} \tag{5.9}$$

The robot heading θ of the robot is calculated according to Equation 5.9 and updates the heading of the robot based on its motion. The encoder-based heading calculation has to be bounded between [0, 2π].

As Eq. 5.9 presents the differential state in the inertial frame, calculating the pose means to integrate Eq. 5.9 from 0 to current time T. The solution to the differential equation is shown in the combined vector y_t for the SLAM algorithm in Chapter 4.

To integrate the pose, sum the speeds over a discrete time interval $[0, T]$. As the time interval grows, the approximations become inaccurate due to accumulation of errors from changing velocities due to dynamics, road surface, wheel slipping, misread ticks, asymmetrical wheels etc. [30].

5.1.2 GoT Positioning

The algorithm for calculating the global coordinates of the robot is based on the input from the GoT Receiver. The input represents the distances from each satellite (beacon) to the robot. The distance is calculated using time difference of arrival (TDOA) of the signals sent by the beacons. The configuration of the beacons in the AAU MP Laboratory can be seen in Figure 5.5.

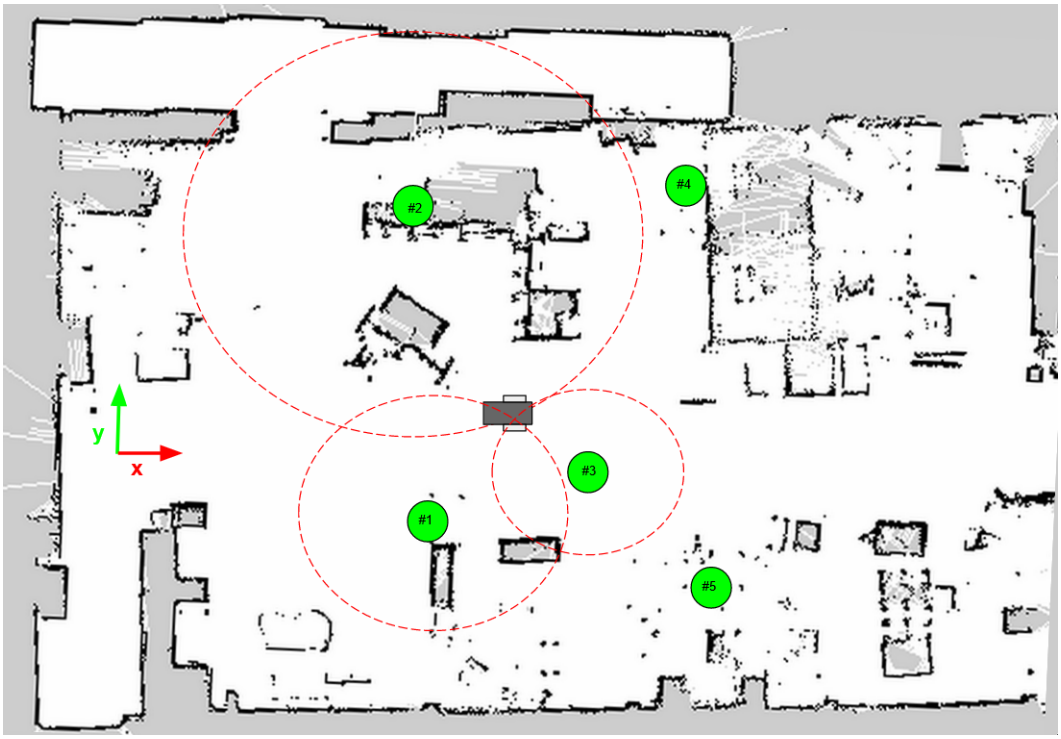


Figure 5.5: Top-view illustration of the beacon positions in the AAU MP Laboratory. The position of each of the 5 beacons is precisely know. An illustrative example of estimating the position of the robot using TDOA is shown for beacons ID 1,2,3 - estimated position is at the intersection of the three circles. The coordinate frame of the Got system is shown on the left which also represents the origin of the system.

In Figure 5.5 the principle of trilateration is illustrated. Multilateration can also be done using more than 3 beacons. Trilateration computes the robot's coordinates based on the distance measured between transmitter and receiver. More exactly, TDOA determines this distance by measuring the time it took for the transmitted signal from each beacon to reach the receiver. This difference in arrival times determines the distance: the more distant the beacons the higher the time difference, hence in Figure 5.5 beacon #2 is the more distant beacon compared to the robot position. This time difference represents the foci of the circles created by the arrival times. The robot position is estimated to be at the intersection of circles. The distance to the receiver is given in Equation 5.10 where c is the speed of sound, t_{beacon} is the time of the beacon and $t_{receiver}$ is the time of the receiver:

$$d_i = c \cdot (t_{beacon} - t_{receiver}) \tag{5.10}$$

The time difference between beacon and receiver has to be accurate. Hence, one important aspect of TDOA is that all beacons have to be synchronised over a very precise

clock, emitting the signal at the same time.

As mentioned before, GoT uses both ultrasound and radio technology for location determination, hence for a radio signal the power of the received signals from the beacons - the received-strength-signal-indicator (RSSI) is used to qualify measurements. The RSSI decreases in value when the distance between receiver and transmitter increases.

Once the distance d_i is determined either by Equation 5.10 or from the signal strength, the location of the robot can be estimated using Equation 5.11 of radius d_i for each of the beacons:

$$d_i^2 = (x - x_i)^2 + (y - y_i)^2 \tag{5.11}$$

The code determining the distances between the receiver and beacons as well as computing robot's position is in the project's repository [35] under GoT_Serial.

A special case is made for the ROS simulation of the GoT system in Chapter 7. To closer reproduce the systematic error of GoT one of the beacons is set to report an erroneous distance. Namely the true distance to the robot reported by one of the beacons is multiplied by a constant value, in this case chosen as 1.1. This solution for simulating GoT in ROS and Gazebo represents an assumption on the interference errors the real GoT system exhibits.

5.1.3 Sanity Check

Sanity checks are tests used to evaluate a claim in a basic manner. The claim to check is the accuracy of the odometry versus the global positioning measurements.

This test verifies the variance in measurements between the two algorithms when the robot is static and when the robot moves in a straight line. In Figure 5.6 the measurements of GoT are plotted with the robot at the origin. The standard deviation is 0.1 m, the variance is 0.01 m and mean is 0.23 m.

Figure 5.7 plots the positions of the robot as measured by odometry and GoT. The robot is commanded to move 2m in a straight line from origin on the x-axis. The figure illustrates the GoT signal is rather precise than accurate. Odometry can be observed to quickly drift.

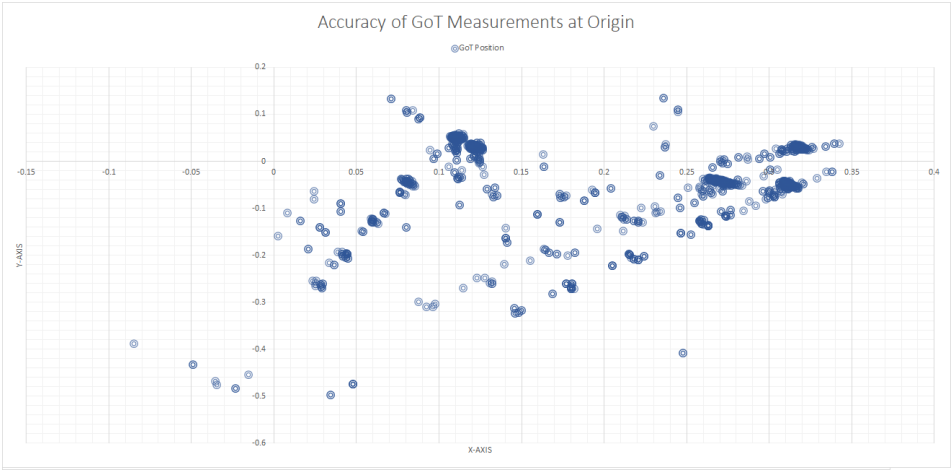


Figure 5.6: GoT measurements of the robot position at origin. The robot is static.

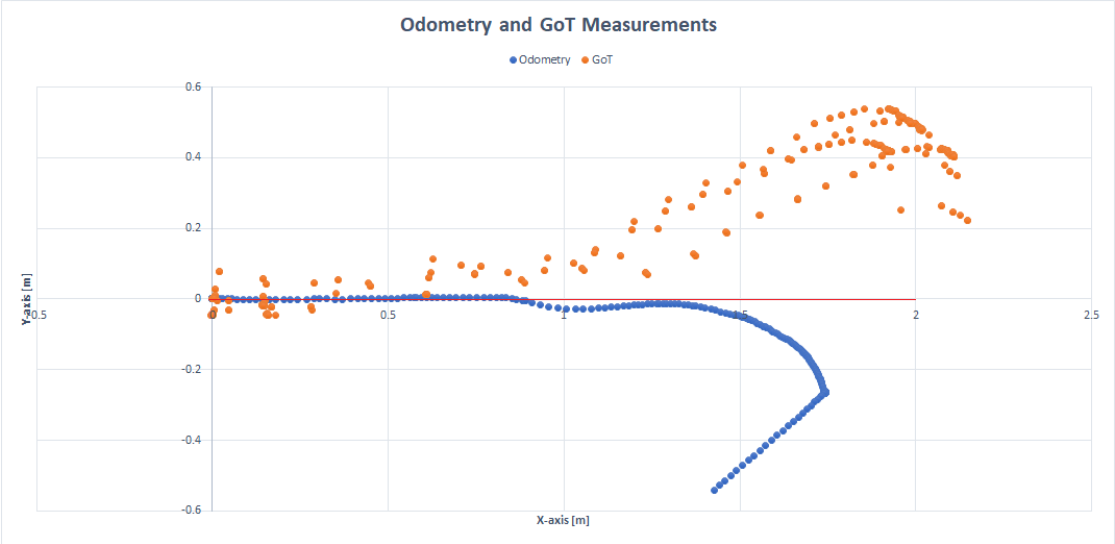


Figure 5.7: Robot position estimation through encoders versus GoT system when the robot moves 2m on the x-axis. Inaccuracies of the GoT system (orange) are the most striking especially because of the double estimations. These double estimations may happen due to interference.

The results of the sanity test confirms that both sensors provide inaccurate measurements of the robot position and sensor fusion may be able to provide a better estimate between the two.

5.2 Robot Orientation

This section describes the algorithm implemented to calculate robot's orientation relative to a given target position. This is done by reading the magnetometer, process target position and calculate the difference between actual and desired heading. Heading is also obtained from magnetometer and encoders.

There are several ways available to compute robot's heading in a Cartesian space: odometry (encoders), magnetometer and directional vectors. For robot heading, the directional vector algorithm is used, however for sensor fusion both the odometry and magnetometer are considered in order to reduce the accumulation of error from encoder readings and interference errors from the magnetometer.

5.2.1 Directional Vector Heading

The robot needs a defined behaviour when navigating towards goal points. A behaviour for travelling between two points needs to be set. The behaviour in this project implies the robot to yaw in the direction of the line between two waypoints and follow this line until the goal is reached. If an obstacle is found on the way, the Bug 1 algorithm can be initiated as described in Chapter 4.

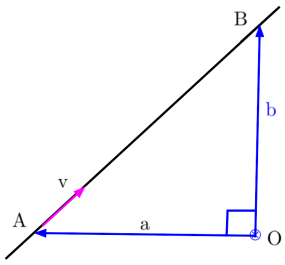


Figure 5.8: Line equation defined by two points. $\overrightarrow{v}$ is the directional vector with direction parallel to line l. O is the origin. $\overrightarrow{a}$ and $\overrightarrow{b}$ are the position vectors of A and B.

The robot behaviour for line following is described by a line equation between 2 known points. In the case of navigation, these 2 known points are the waypoints - one is the position of the robot and the second is the goal position. A line l can be described if a point on it and a directional unit vector $\overrightarrow{v}$ are known.The line equation dictates that if two points A and B are known then $\overrightarrow{v} = \overrightarrow{AB} = (b - a)$ gives a line equation as in Eq. 5.12.

$$l = (1 - \lambda)a + \lambda b \tag{5.12}$$

Eq.5.12 is a equation with the only unknown in λ. If λ is computed, distance and point of intersection to line l can be found. λ is a scalar between [0,1] which can be found by

normalizing the difference between robot position and desired location with the distance to the target location.

The directional vector is given by the z-axis component of the cross product between the two position vectors a and b. If the origin O is at the center between the two wheels in Figure 5.9, u is the magnetometer heading vector and v is the reference vector to the line perpendicular to u - Eq. 5.13, then the cross-product between vectors $\bar{v}$ and $\bar{u}$ is the heading angle away from the reference. In Eq. 5.13 R denotes the 90 degree rotation matrix between the u and v. The cross product is given by Eq.5.14.

$$v = R_{\frac{\pi}{2}} u \tag{5.13}$$

$$(v \times u)_3 = |v||u| \sin(\angle v, u) \tag{5.14}$$

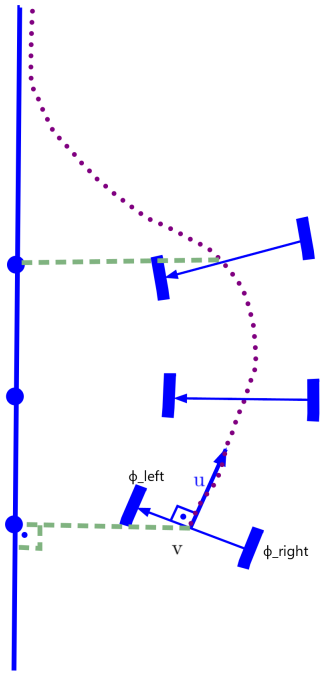


Figure 5.9: Robot described by two wheels connected by an axle. Figure shows evolution of robot orientation while converging to the line.

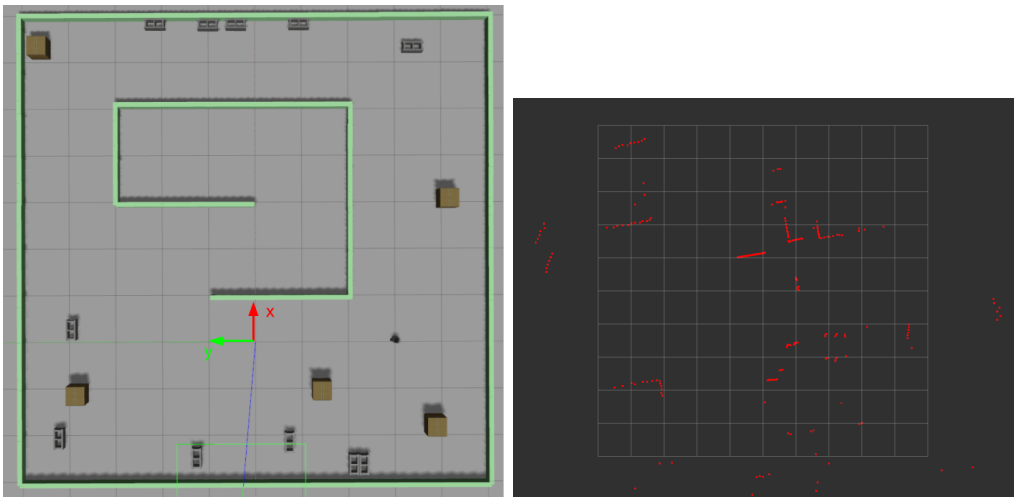
The desired behaviour of the robot when navigating towards a goal position is to yaw towards the line between the two waypoints and converge to the line steadily. This is pictured in Figure 5.9.

6 High-Level Design

This chapter describes the high-level design for robot localization, mapping and obstacle avoidance. It describes the ROS packages used, setting up the simulation environment and the fingerprinting of GoT errors in position estimations.

6.1 Gazebo Multi-Robot Simulation Environment

As mentioned in Chapter 1 the robotic solution envisions a low-cost swarm of mobile robots for navigation in the industrial context. The swarm of robots is formed by a leader robot equipped with necessary sensors to map and fingerprint position errors on the map for other robots that are equipped with basic sensors: encoders, magnetometer and GoT. A simulation of a lead and client robot is created to verify the accuracy of the client robot when navigating the map created by the leader.



(a) Gazebo bug_map_world (b) Rviz LiDar Measurements inside the world

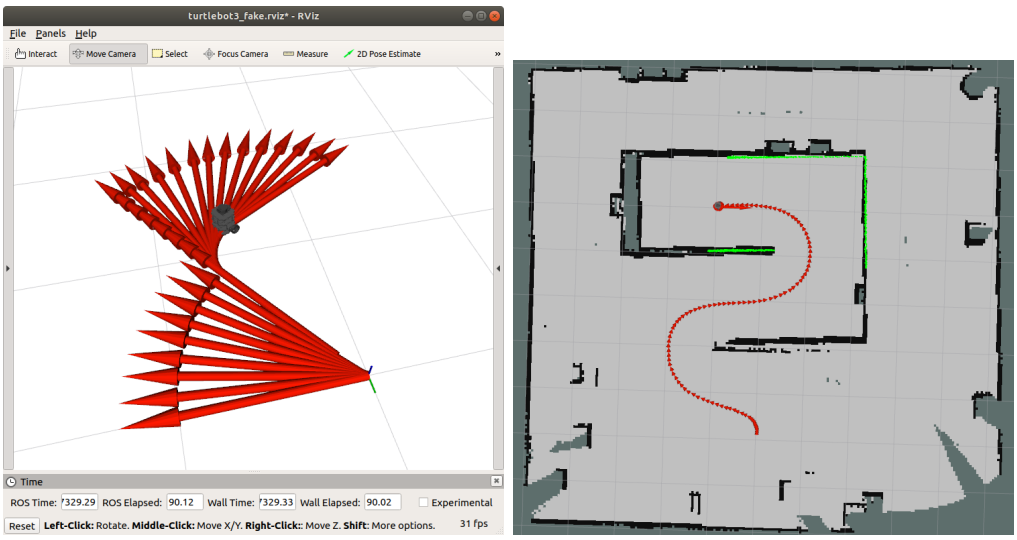
Figure 6.1: Image (a) shows a created bug_map world with obstacles, features of symmetry and Bug2 algorithm; (b) shows the LiDar measurements from the robot inside the world. This output can only be seen in Rviz.

The simulation of the robots and the environment is realized with the help of Gazebo [36] a popular simulation program which can be used with ROS. Gazebo requires a robot URDF [37] model, which describes the kinematic model of the robot. Due to incompatibilities between the dimensions of the simulated robot and the maps in Gazebo, the simulation relies on the packages provided by Turtlebot[38]. Turtlebot's URDF file has been modified to include only the sensors available on the real robot. This means that the magnetometer, IMU, GoT and LiDar have to be simulated. Gazebo plugins [39] to

simulate a differential drive controller, IMU, LiDar and magnetometer sensors are used. The only system left to simulate is the GoT. This is described in Section 7.1.

The files for configuring the ROS framework and Gazebo can be found in the repository [35] and following the steps provided in Appendix K. In the simulation the robot can move either manually by velocity commands using the keyboardjoystick or autonomously by using a script to move the robot to desired coordinates. The commands for adding the joystick package are explained in Appendix K Section K.2.

A Gazebo world was developed to contain few features: symmetry, obstacles and Bug1 algorithm challenges. The world is shown in Figure 6.1 (a). Alongside Gazebo the sensor output visualization environment Rviz is started. Rviz shows the LiDar output of the Gazebo world. To move the robot autonomously in a Gazebo world a script *go_to_point* implements the directional vector algorithm and waypoint navigation.



(a) RViz windows showing the simulated robot and its commanded movement.

(b) Rviz LiDar, odometry and map output of the gazebo_bug_map world.

Figure 6.2: Image (a) shows an illustration from Rviz of the odometry (red) of a navigating robot. (b) shows the sensors output in Rviz from a navigating robot: Lidar, odometry and SLAM of the world. This output can only be seen in Rviz.

In Figure 6.2 images show the sensor output in Rviz as the robot is navigating the Gazebo world. The script is in the repository [35] under */ros/got_node/scripts* and is launched using the following command:

```
$ rosrun got_node go_to_point.py
```

As a result, the robot navigates through several waypoints on the world. These points are listed in Listing 6.1.

```
# (x,y) position goal. Robot start position is (2.0, -4.0)
goal_points_ = [
    (-2.0, -4.0),
    (-2.0, -2.0),
    (2.0, -2.0),
    (2.0, 0.0),
    (-2.0, 0.0),
    (-2.0, 4.0),
    (2.0, 4.0),
    (0.0, 2.0)
]

rospy.set_param('des_pos_x', goal_points_[0][0])
rospy.set_param('des_pos_y', goal_points_[0][1])
```

Listing 6.1: Waypoints for the robot to navigate in Gazebo

The script loads the first goal coordinates and moves the robot in position. Once at the goal destination the next point is loaded and the robot moves again until it reaches the last point. New waypoints can be added to the path while the robot is navigating.

6.2 ROS SLAM

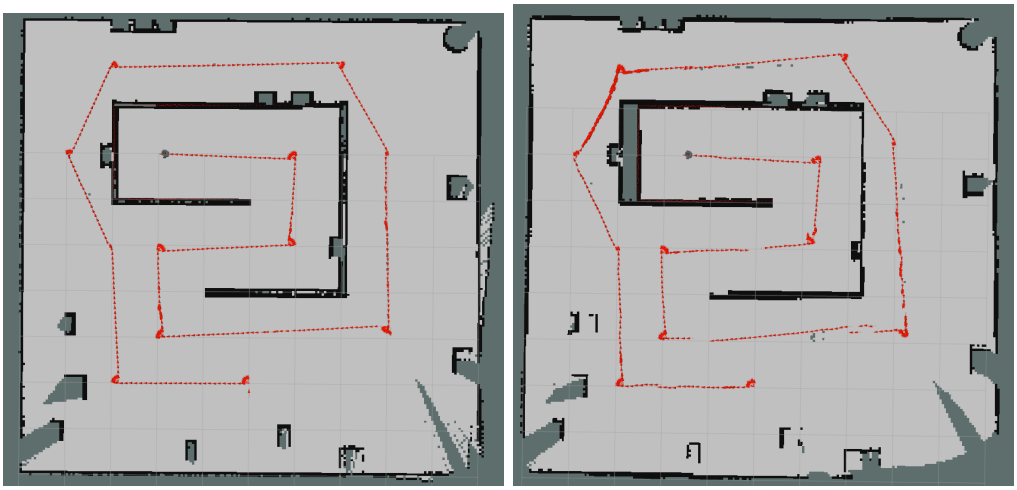
There are few SLAM implementations in ROS however **hector_slam** [40] is used in this project mainly because of its use of EKF or UKF filters for the robot state estimation. The EKF SLAM algorithm is preferred to other filters due to the estimation of the state vector to be able to separate pose estimations from mapping.

In Figure 6.1 (a) the Gazebo world is shown along with frame coordinates of the robot. After starting ROS SLAM in the Gazebo world, while the robot is moving, it is possible to see in RViz that a map is being built. The map is assembled by using the Lidar output and the odometry data transmitted by the simulated robot - Figure 6.2 (b). ROS SLAM is both implemented on the robot and used in the simulation.

A comparative case between 2 ROS SLAM packages is shown below: *hector_slam* and *gmapping*. *hector_slam* uses EKF while *gmapping* uses the particle filter. One example of the two packages performing SLAM using the `go_to_point.py` script can be seen in Figure 6.3.

A visual inspection of the different filters used for SLAM in Figure 6.3 shows that both perform good in localization and mapping. It is however preferred to use *hector_slam*

as it implements EKF SLAM and the output is pose estimation. The only output of *gmapping* is the map.



(a) Rviz illustration of *hector_slam* performing `go_to_point.py`.

(b) Rviz illustration of *gmapping* performing `go_to_point.py`.

Figure 6.3: Comparative example of 2 ROS SLAM packages hector_slam and *gmapping* executing the `go_to_point.py`.

At any point in time the built map can be saved to a file. To do so run the following command in a new terminal:

```
$ rosrun map_server map_saver -f ~/map
```

The command creates 2 files `map.pgm` and `map.yaml`. The *.yaml* file contains some parameters of the map relevant for RViz, but map.pgm is a binary file which can be opened by an image viewer.

6.3 Sensor Fusion

Sensor fusion is the concept of simultaneously using more sensor signals to improve performance. An approach is to apply one of the KF types: KF, EKF, UKF. The ROS package *robot_localization* implements EKF for robot localization based on multiple sensors providing redundant information about the robot's pose. The package can estimate the full 3D state vector containing 12 states shown in Eq. 6.1. [41]

$$\begin{aligned} y_{3D} = [x \quad y \quad z \quad \phi \quad \psi \quad \theta \quad \dot{x} \quad \dot{y} \quad \dot{z} \quad \dot{\phi} \quad \dot{\psi} \quad \dot{\theta}]^T \quad where \\ \phi - roll \quad angle, \quad \psi - pitch \quad angle, \quad \theta - yaw \quad angle \end{aligned} \tag{6.1}$$

The input to *robot_localization* is several sensor signals including the output from *hector_slam* node. It is aimed to achieve an improvement in localization. The results are presented in the next chapter. The interaction between sensor fusion and the other ROS packages is shown in Figure 6.4. The *robot_localization* package performing sensor fusion creates 2 frames: local and global. The local frames accepts sensor information from continuous sensors, while the global frame accepts sensor data from global positioning systems such as GoT and *hector_slam* for the map. From sensor fusion the updated pose is sent to the navigating stack - either *move_frame* or *go_to_point* to execute the motion.

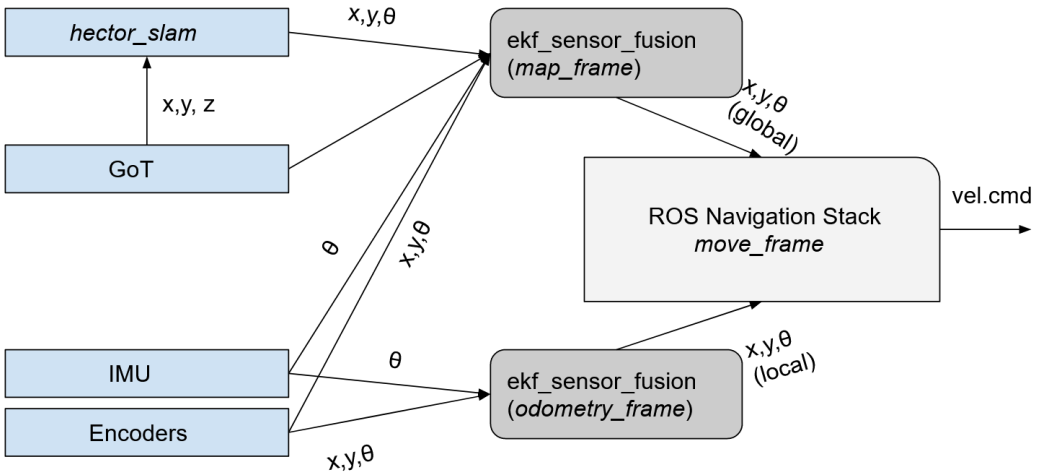


Figure 6.4: Diagram illustrating how the ROS package *robot_localization* performs sensor fusion. Connection with other ROS packages is shown: *hector_slam* for SLAM and *move_frame* for robot navigation.

As the ROS package uses an EKF for sensor fusion a generalized nonlinear system model is used to fuse data. The model is based on the constant velocity model presented in Appendix D. The configuration vector for data fusion can be seen in Table 6.1.

Sensor	***Configuration Vector*** ***0 = false, 1 = true***											
	x	y	z	ϕ	ψ	θ	$\dot{x}$	$\dot{y}$	$\dot{z}$	$\dot{\phi}$	$\dot{\psi}$	$\dot{\theta}$
Odometry	0	0	0	0	0	0	1	1	0	0	0	1
IMU	0	0	0	0	0	0	0	0	0	0	0	1
GoT	1	1	1	0	0	0	0	0	0	0	0	0

Table 6.1: Sensor fusion configuration vector for available sensor data.

7 Implementation

This chapter describes the implementation of the GoT error correction system. A step-by-step approach is used, starting with getting the raw reading from the GoT system and ending with providing a corrected GoT position estimate.

7.1 GoT ROS node and Fingerprinting

This section presents the simulation of the GoT positioning system in ROS. It also describes the position error fingerprinting which patch the real and simulated GoT measurements.

It was mentioned in Chapter 1 the lead robot is equipped with sensors in order to produce accurate position estimates through SLAM so that client robot that are equipped only with basic sensors can navigate the map based on GoT measurements. Given that the GoT system suffers from systematic errors, the client robot can not correct them.

By fingerprinting error areas in the laboratory, the client checks the server for fingerprinted area at each position estimation. If a fingerprinted area is found, the robot requests an error correction made by the lead robot. The error correction and patching the GoT signal is explained in steps 6-8 below.

An example of fingerprinted areas on the AAU MP laboratory map can be seen in Figure 7.1. The red circles represent the fingerprinted areas where GoT error corrections are available. The green squares represent the resolution of the fingerprinted area. Resolution is discussed later in the section.

If the client robot does not enter the circled areas in Figure 7.1 it will use the GoT positioning without any corrections as they do no exist. The client can perform sensor fusion between the corrected GoT positioning and odometry to improve it pose estimation.

To re-create in ROS the configuration of the beacons as illustrated in Figure 5.5 from the AAU MP Laboratory, a fixed number of immovable beacons is used to obtain the position estimate and an error is introduced in one of the beacons such as to simulate the systematic GoT error. The map used to place the beacons is illustrated in Figure 6.1 (a).

The configuration of the beacons in the map is shown in Figure 7.2 where the faulty beacon is marked with red. The area of effect introducing errors to the GoT measurements is marked with the green interrupted line. This fingerprinting of the error area is only available in the simulation. Fingerprinting for the laboratory is described in Step 7 below. Moreover, in the simulation all other beacons than the faulty one report the ground-truth position of the robot.

Figure 7.1: Illustration of the fingerprinting of GoT systematic position errors. It is assumed most positioning error happen around obstacles. The area (red) around the obstacle would have an error correction associated to it so that when the client robot navigates inside the area, it receives the GoT error correction assigned for that position. Green represents the resolution of the fingerprinting.

The implementation of the simulated beacons, fingerprinting and GoT correction follow the steps enumerated below. Steps 1-5 regard the simulation of the GoT beacons, steps 6-8 refer to fingerprinting and error correction in both the simulation and real-world implementation.

1. Define 5 fixed positions of the beacons on the map.
2. Obtain the ground truth robot position by using the *libgazebo_ros_p3d* (seen in the robot URDF file).
3. For each of the defined beacons find the Euclidean distance to the robot.
4. Make one beacon to report a slightly longer distance, to simulate GoT error.
5. Use beacon distances to define a robot position estimate in Cartesian coordinates.
6. Patch the GoT position using a ground truth reference.
7. Store all errors and the respective ground truth estimate.
8. Distribute the error correction to all robots in the systems.

7.1.1 Steps 1-5 Simulating the GoT Beacons

In Figure 7.2 is seen that the beacons are placed symmetrically about the map and Listing 7.1 shows the code implementation how the beacon positions are defined and queried to obtain a robot position estimate in the ROS simulation.

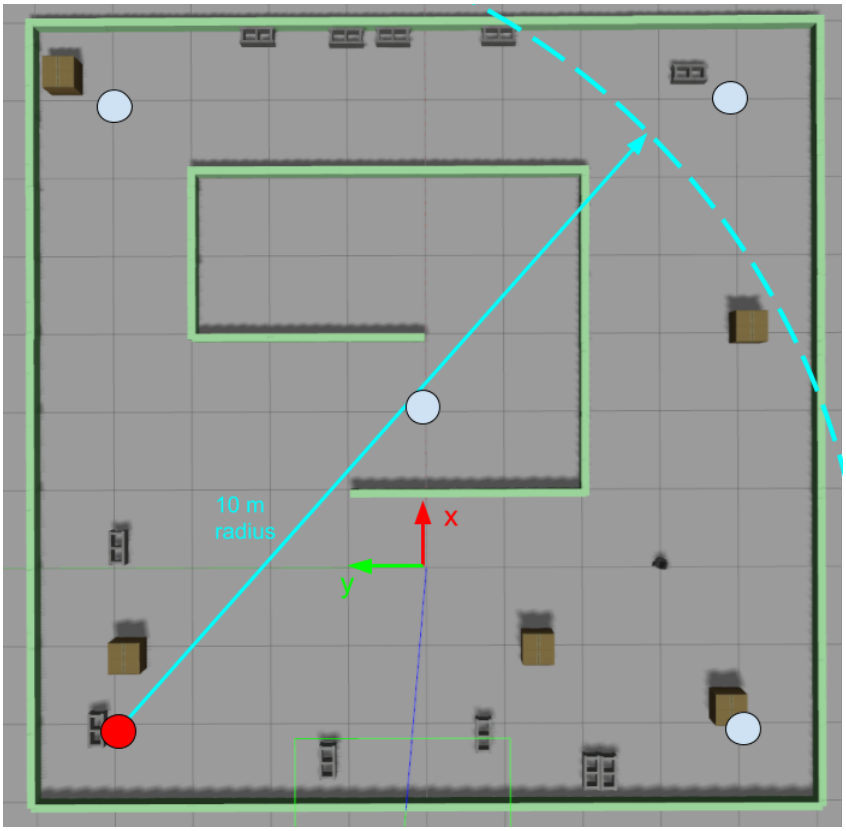


Figure 7.2: Gazebo simulation map where the position of the beacons is indicated by the light-blue circles in the four corners and the middle of the map. The faulty beacon is colored in red and its range represented.

As described in Section 5.1.2 the same algorithm of trilateration for estimating the robot position is used in Gazebo. The code in the listing below calculates the distance between the beacons and the robot by means of the ground-truth produced by the simulation. It is one of the computed distances that reports an erroneous distance. Specifically the correct distance gets multiplies with a factor of 1.1 to simulate a higher distance to the robot or an interference from an obstacle. From the distances the global position of the robot is estimated. Steps 1-5 are marked in the listing.

```
1
2  // STEP1: Define 5 fixed positions of the beacons on the map.
3
4  #define NUM_BEACONS 5
5  const double beacon_locations_[NUM_BEACONS][3] = { {-2,4,5}, {-2,-4,5},
       {2,0,5}, {6,4,5}, {6,-4,5}};
6
7  double got_x_=0, got_y_=0, got_z_=0;
8  // Cycle through all beacons
9  for(uint8_t index = 0; index < NUM_BEACONS; index++)
10 {
11
12 // STEP2: Obtain the ground truth robot position
13 // STEP3: Find the Euclidean distance to the robot
14
15   // Euclidean distance in meters from beacon to robot
16   // ground_x,y,z is the ground_truth reported by Gazebo
17   double distance = pow(beacon_locations_[index][0] - ground_x, 2);
18   distance += pow(beacon_locations_[index][1] - ground_y, 2);
19   distance += pow(beacon_locations_[index][2] - ground_z, 2);
20   distance = sqrt(distance);
21
22 // STEP4: Simulate GoT interference error
23
24   // Insert distance error for lower-left beacon
25   if (index == 0)
26   {
27     distance *= 1.1;
28   }
29   // Ignore beacon if reported distance is outside acceptable range
30   if(distance>10 || distance<1)
31       continue;
32
33 // STEP5: Use beacon distances to define a robot position estimate
34
35   // Euclidean distance from the beacons to the position estimate got_x,y,z
36   double dp=pow(got_x_- beacon_locations_[index][0], 2);
37   dp+=pow(got_y_- beacon_locations_[index][1], 2);
38   dp+=pow(got_z_- beacon_locations_[index][2], 2);
39   dp=sqrt(dp);
40
41   // Normalizing factor
42   double lambda = 1 - dp / distance;
43
44   // Estimated Cartesian coordinates
45   got_x_=got_x_/(1-lambda)-beacon_locations_[index][0]*lambda/(1-lambda);
46   got_y_=got_y_/(1-lambda)-beacon_locations_[index][1]*lambda/(1-lambda);
47   got_z_=got_z_/(1-lambda)-beacon_locations_[index][2]*lambda/(1-lambda);
48 }
```

Listing 7.1: Code fragment showing simulation of GoT beacons in ROS and Gazebo

7.1.2 Step 6 - Patching the GoT Position Error

This step identifies the error of the GoT position compared to another reference/ground-truth. While in the simulation environment it is possible to calculate the error as the difference between GoT position and ground-truth, in the real world the ground-truth is not available. Instead the GoT position is corrected with a certain confidence level by taking either fused odometry (odometry and IMU) or EKF SLAM pose estimate as the reference. More exactly, using a ground-truth estimate x_{gt}, y_{gt} with a respective GoT position x, y and the error between $\Delta x, \Delta y$ the error patch is defined in Eq. 7.1.

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} * e^{\frac{-(x-x_t)^2-(y-y_t)^2}{s}} \tag{7.1}$$

, where s is the variance between the two signals considered here a scaling factor, high values favoring the ground-truth estimate and lower values favoring the default GoT position.

The listing below shows the code implementation of the error patch, where the obtained position error err_x, err_y is added to the reported GoT position from Listing 7.1.

```
136     double dx = ground_x - got_x_;
137     double dy = ground_y - got_y_;
138 
139     err_x_ = -(pow(dx,2)) - (pow(dy,2));
140     err_x_ = dx * exp(err_x_ / err_s_);
141 
142     err_y_ = -(pow(dx,2)) - (pow(dy,2));
143     err_y_ = dy * exp(err_y_ / err_s_);
```

Listing 7.2: Implementation of the GoT error patch, where err_x and err_y are later used to correct GoT position

7.1.3 Step 7 - Store all errors and the respective ground truth estimate.

Error storage requires a method that would allow fast access to the stored data. Assuming data would be stored as a pair of GoT position x, y and error x_e, y_e the first idea would be to create a multi-dimensional array that would hold one position error pair per cell. This approach however is less than ideal since it would require iterating through the array for every position which increases computation time and can negatively impact the performance of the ROS node in charge of reporting the corrected GoT position.

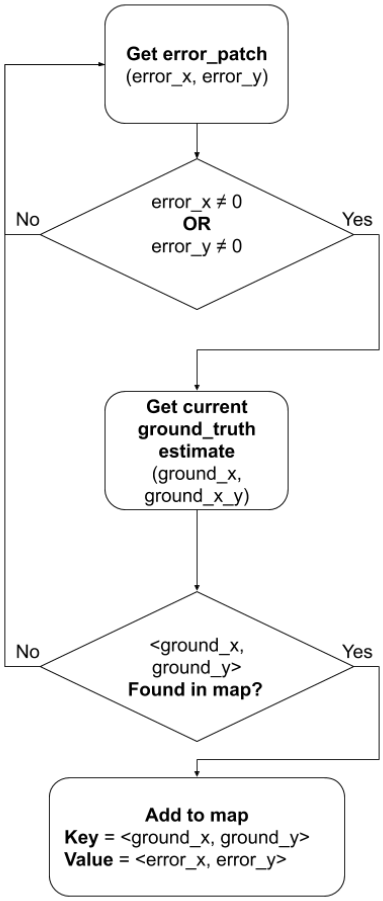


Figure 7.3: Flowchart for storing error patches in an unordered map.

The approach taken in this project is to use the *C++ unordered map*. An unordered map stores data as *key, value* pairs where the keys are unique.

Data is stored and sorted by computing the hash of each key [42]. The steps taken in this process can be summarized by the flowchart in Figure 7.3.

Fingerprinting Resolution

The resolution of position estimation represents a challenge related to the GoT positions recorded by the lead robot during navigation from one point to another. Given the same navigation waypoints the client robot may not pass through the same positions on the map as the lead robot. If there exists a GoT error patching at position 2.00309987644 but the client does not navigate through the same position it will not receive the patch.

The solution is to create error-patching areas for zones where the error between the GoT signal and ground-truth are above a certain admitted threshold. These areas have been marked with red in Figure 7.1. The specified threshold is 0.2 m as the mean of the GoT signal computed at origin in Chapter 5.

The code implementation of reducing the resolution of the position estimate is shown in Listing 7.3.

```
1       // Reduce resolution to areas of 0.001 m
2       double key_x = ((int)(ground_x * 1000)) / 1000.0;
3       double key_y = ((int)(ground_y * 1000)) / 1000.0;
4
5       Position pos = {key_x, key_y};
```

Listing 7.3: Implementation of the fingerprinting areas to avoid position estimate resolution problems.

7.1.4 Step 8 - Distribute the GoT Error Correction to All Robots in the Multi-Robot Setup

The last step is ensuring that the error correction data, in the form of an unordered map is available to all the robots running on the same ROS server. In this case only a single robot, referred by as *lead* is running the ROS node responsible for error correction and it makes the data available for all other robots referred by as *clients*.

For accomplishing this goal it was decided to make use of the *ROS Service* functionality. This means that the GoT position node running on the lead robot acts as a server and whenever it receives a request from a client robot it will send the error path as a response. Flowchart in Figure 7.4 shows the performed operations.

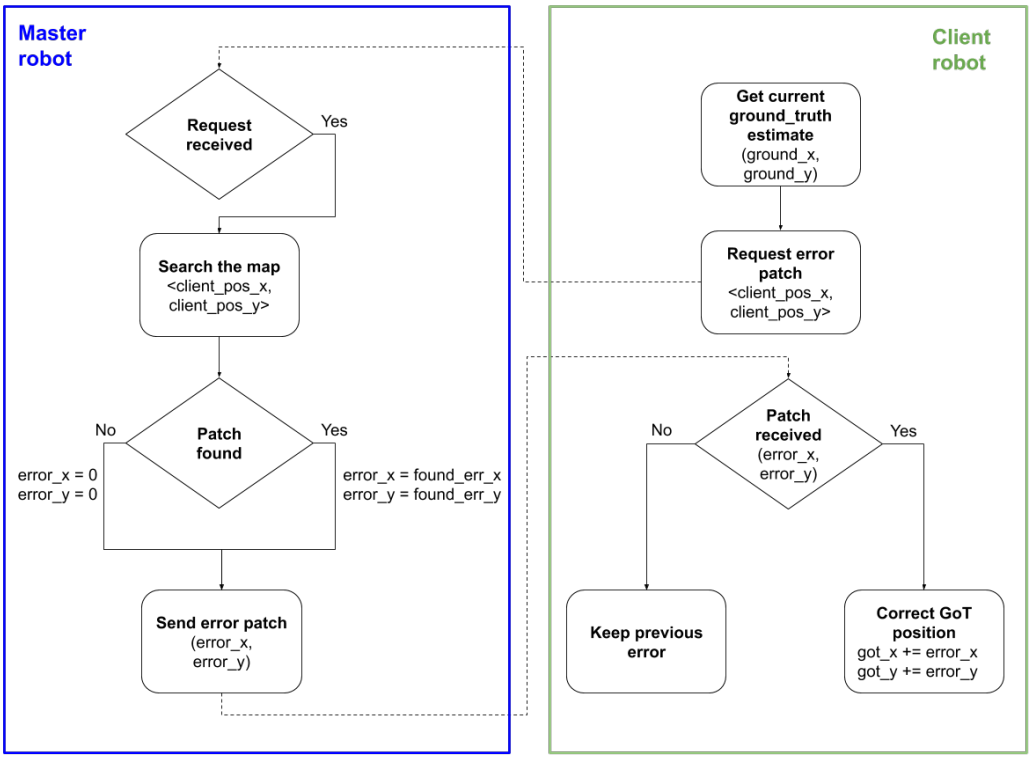


Figure 7.4: Flowchart showing the required steps for the error patch to be made available by means of ROS Service

For clarification, the following figure shows the running master(on top) and client(on bottom) nodes where debugging messages are printed:

```
[ INFO] [1590445759.305596269]: sending back response: ex:[-9.429862] ey:[-0.333385]
[ INFO] [1590445759.306875937]: sending back response: ex:[-9.429862] ey:[-0.333385]
[ INFO] [1590445759.308066140]: sending back response: ex:[-9.446573] ey:[-0.333518]
[ INFO] [1590445759.309281669]: sending back response: ex:[-9.446939] ey:[-0.332653]
[ INFO] [1590445759.310512463]: sending back response: ex:[-9.446939] ey:[-0.332653]

[ INFO] [1590445756.338161339]: Server response: -9.466518, -0.378308
[ INFO] [1590445756.385600951]: Server response: -9.460497, -0.378225
[ INFO] [1590445756.438341247]: Server response: -9.460368, -0.378544
[ INFO] [1590445756.490684553]: Server response: -9.460368, -0.378544
[ INFO] [1590445756.538816195]: Server response: -9.460368, -0.378544
```

Figure 7.5: Messages exchanges between master(on top) and client(on bottom) nodes

For this implementation to work it is required to construct a new .srv file describing the request and response message. The file can be found in the repository [35] in `./got_slam/ros/got_node/srv/GetPosError.srv` with the following contents:

```
float32 x
```

```
float32 y
---
float32 err_x
float32 err_y
```

Above the dashed line the request message is defined. Meaning that the client sends its x, y coordinates as reported by GoT. The Master in turn sends err_x, err_y as the error patch that needs to be added to GoT to obtain a corrected measurement.

The commented code for both the Master and the Client robot nodes can be found in `/.got_slam/ros/got_node/src/got_position_pi_master.cpp` and `/.got_slam/ros/got_node/src/got_position_pi_client.cpp`

7.1.5 GoT Error Correction Parameters

The designed master ROS node for handling and correcting GoT position also has several parameters that can help in deciding on the error correction method:

- `got_err_tolerance` - Set the threshold required to register a position error, default is 0.2;
- `got_err_s` - The error correction scaling factor. Higher number favor the ground truth estimate while lower values favor the actual GoT reading. Default is 3000;
- `got_ground_truth` - Choose the ground truth estimate, which can be either the *odom*, for robot odometry or *slam* to use SLAM position estimate.

For more details see Figure 7.6 where the ground-truth parameter is changed twice.

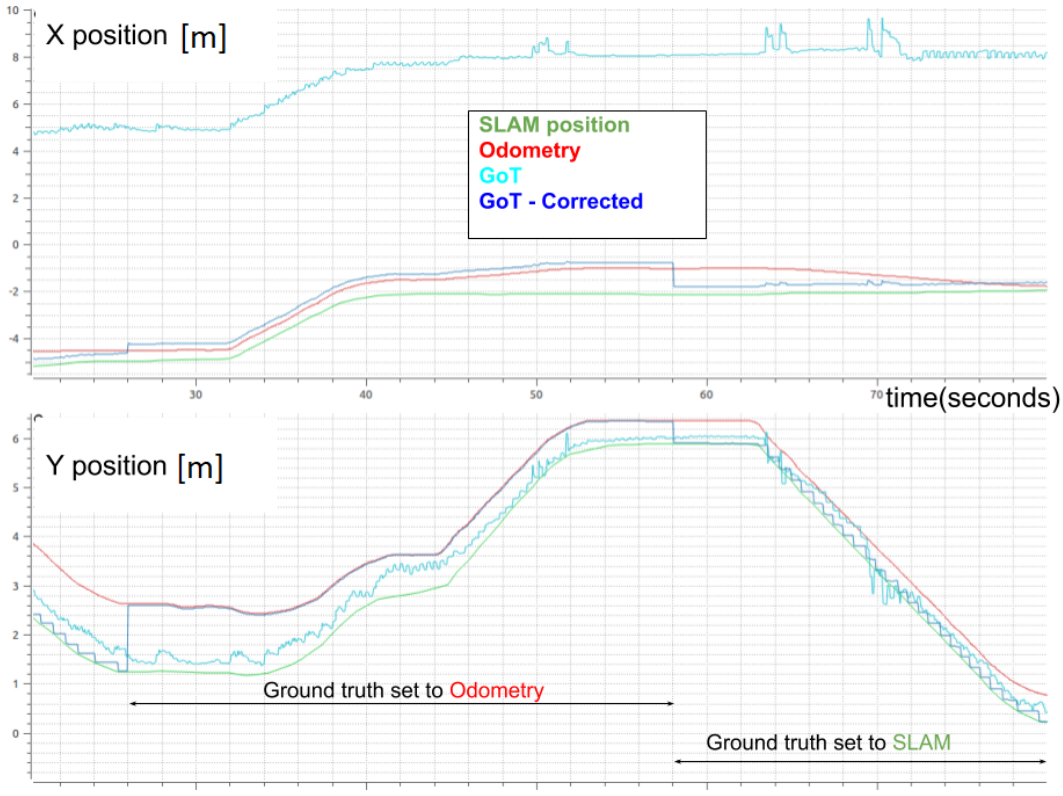


Figure 7.6: Plot showing the difference between the actual and corrected GoT signals for a moving robot. Upper plot shows the value of the X position and the lower one for Y position. Ground truth estimate is changed first to Odometry and then to SLAM. Since the scaling factor has a high value it can be seen how the corrected GoT reading is pulled towards the ground-truth.

7.2 Final Notes

After all design considerations the final functionality of the GoT error correction workflow can be summarized by Figure 7.7. This setup is used in both the simulation and real-word examples.

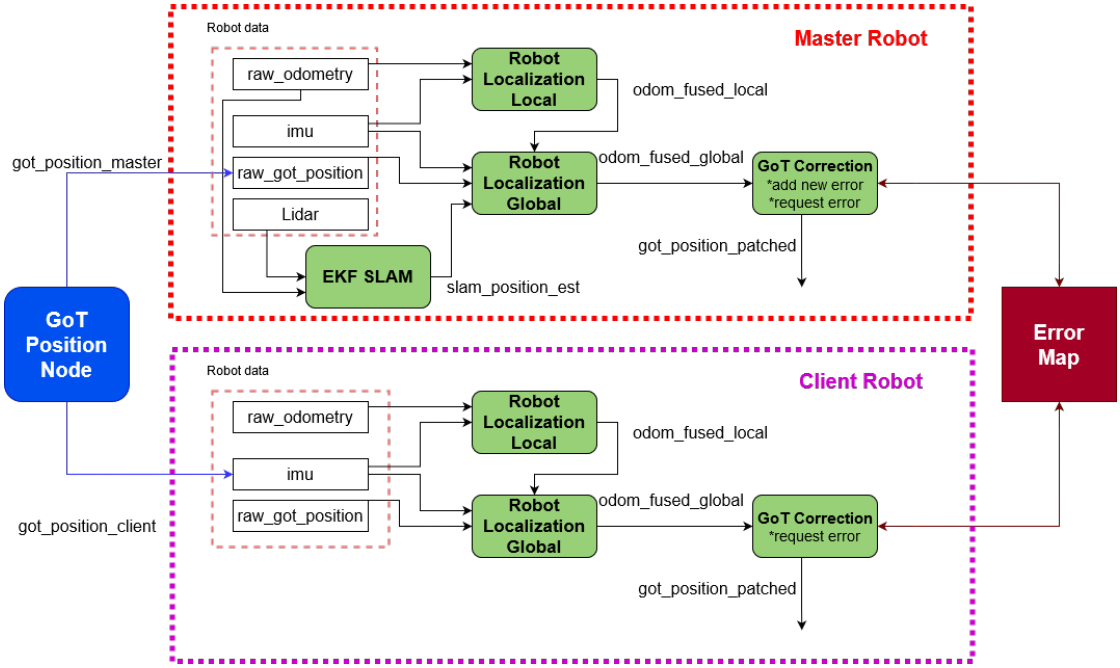


Figure 7.7: Flowchart describing the implementation of the GoT error correction system using ROS, where the green nodes, GoT position Node and Error Map are nodes running in the ROS framework.

The GoT Position node is a ROS node that handles the data received from the GoT system and reports the position of every robot in the system. Every robot is assumed to be access odometry, IMU and Lidar (in the case of the master robot).

All data is used across multiple nodes as seen in Figure 7.7 to obtain an optimal ground truth estimate (*odom_fused_global*) and use it to patch the GoT position estimate. Afterwards the ground estimate and detected GoT erors are inserted into **Error Map** node which makes the errors available to all the robots in the system.

8 Simulation and Implementation Testing

This chapter presents the results obtained from the simulation and laboratory work. It seeks to illustrate the difference of similar tests across the two media.

8.1 ROS and Gazebo Simulation Results

In this section the results of simulations-in-the-loop are presented. It starts with simulating the *go_to_point* script in the *gazebo_bug_map* world allowing the robot to visit all waypoints autonomously. To test the performance of sensor fusion and EKF SLAM, the indicators are:

- **Loop closure error**: the robot starts and ends the trajectory at the origin. The error at loop closure is the difference between origin and measurement of each signal at origin.
- **Standard deviation** of the error betwen the signal and ground-truth.

The tests performed to indicate improved accuracy for robot localization on both lead and client robot are mentioned below:

1. Fused Odometry:
   (a) Odometry and IMU referred to as fused odometry
   (b) Odometry, GoT and IMU referred to as fused odometry with GoT
   (c) (optionally) Odometry, patched GoT and IMU referred to as fused odometry with pached GoT
2. Fused Localization:
   (a) EKF SLAM
   (b) EKF SLAM + Fused odometry
   (c) EKF SLAM + Fused odometry with GoT

Accuracy of a sensor points towards the correct values measured. The more correct value the more accurate the sensor. In Gazebo the correct values are given by the transforms of robot location in the world. These represent the ground-truth. Accuracy of the tests is going to be measured against the ground-truth. The results performed in SITL testing are shown in Table 8.1.

Simulation		Lead Robot		Client Robot		
EKF Method	**Input Signal**	Loop Closure Error (x,y)[m]	Signal Accuracy Std.Dev. (x,y)[m]	Loop Closure Error (x,y)[m]	Signal Accuracy Std.Dev. (x,y)[m]	Figure
Fused Odometry	Odometry + IMU	0.04, 0.02	0.05, 0.12	6.1, 7.4	4.88, 5.0	8.2
	Odometry + GoT + IMU	0.08, 0.04	0.04, 0.02	0.0, 16	4.2, 6.8	8.3
	Odometry + patched GoT + IMU	n/a	n/a	1.48, 16.6	4.0, 6.9	8.1
Fused Localization	EKF SLAM	2.0, 1.9	1.58, 1.52	n/a	n/a	-
	(Odometry + IMU) + EKF SLAM	0.1, 0.02	0.04, 0.02	n/a	n/a	-
	(Odometry + GoT + IMU)+ EKF SLAM	0.13, 0.02	0.04, 0.02	n/a	n/a	-
GoT	Raw	0.04, 0.21	1.19, 0.85	0.04, 0.45	1.58, 1.46	8.1
	Patched	0.04, 0.04	0.06, 0.03	0.04, 0.21	1.48, 1.10	

Table 8.1: Result table for the SITL testing of robot localization using sensor fusion and SLAM for the lead robot and testing robot localization for the client robot.

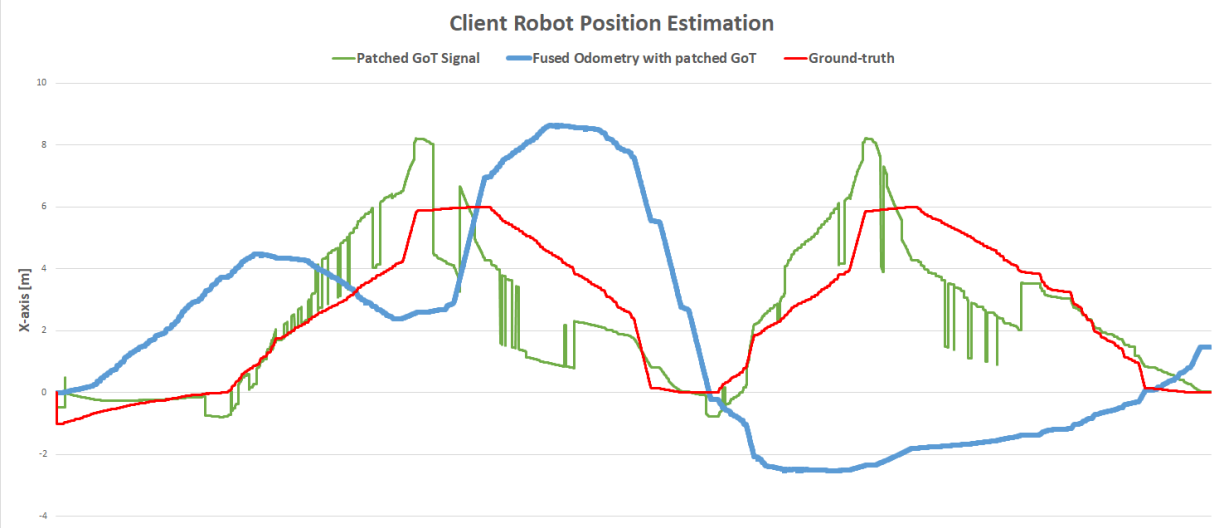


Figure 8.1: x-axis position estimation of the client robot when using fused odometry with patched GoT (blue). Patched GoT (green) estimate is visually closer to the ground-truth (red) supporting the results in Table 8.1.

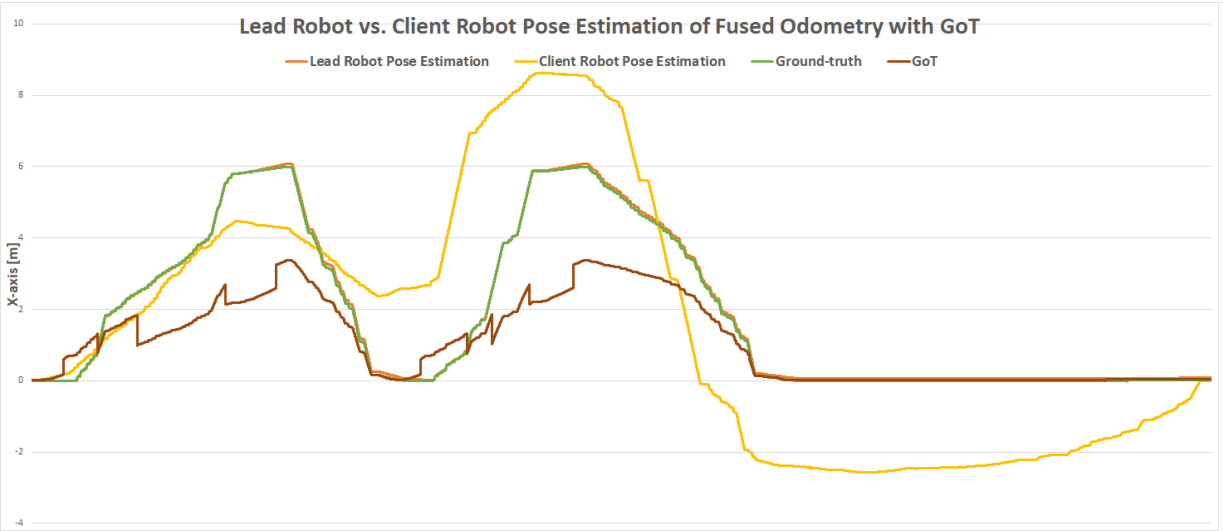


Figure 8.2: x-axis position estimates of the lead (green) versus the client (yellow) robot as measured by fused odometry with GoT. The GoT signal is shown in brown and ground-truth in red. The lead robot and ground-truth estimates are overlapping.

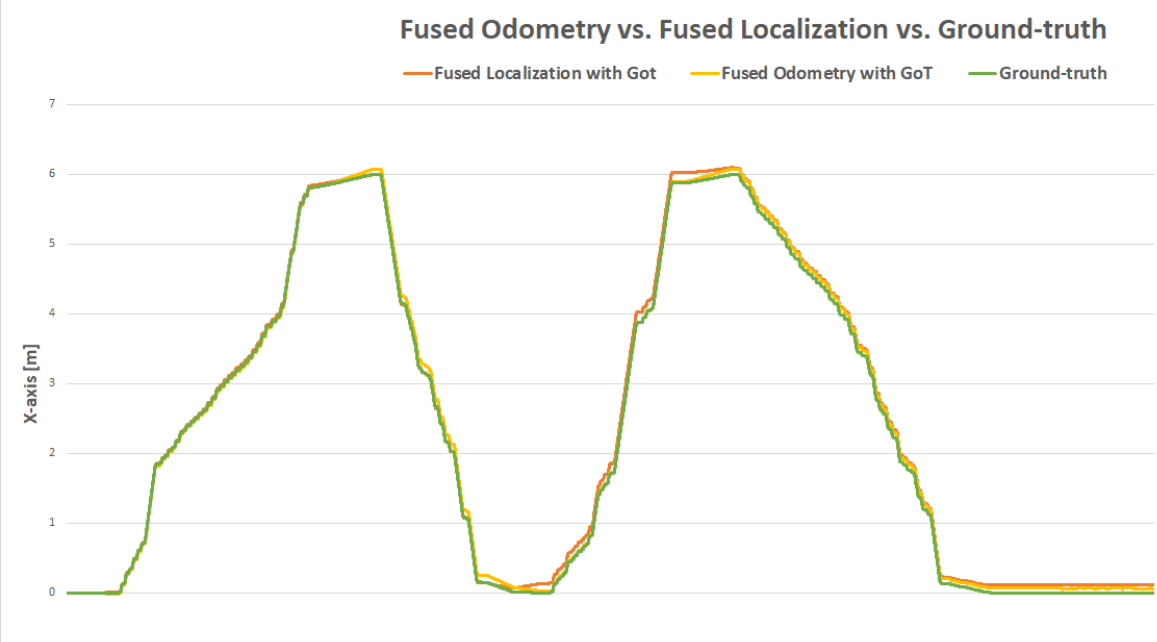


Figure 8.3: Comparison between fused odometry with GoT (yellow), fused localization with GoT (orange) and ground-truth (green) for the lead robot in the simulation. Visual inspection of the graph shows that the results obtained in Table 8.1 are correct.

8.2 Laboratory Results

This section presents the results of the tests performed in the laboratory. These tests bring together all work performed towards improving the localization of the robot on the map. The tests performed are similar with the ones for simulation excepting the blind robot which could not have been built during the timeframe of the project. As mentioned before, testing of the different signals is performed using *rosbag* as described in Chapter 1. In order to run the tests, a map of the AAU MP laboratory has been constructed using the EKF SLAM algorithm in ROS. The ROS map of the laboratory was already introduced in Chapter 5 and 6. For building the map, the robot was controlled remotely hence the data used for SLAM was odometry and LiDar. The saved map can be seen in Figure 8.4.

The functionality of *rosbag* can be seen in Figure 8.4 where a trajectory has been given to the robot to navigate autonomously. *rosbag* has recorded all sensors measurements including the output of the sensor fusion algorithm (red track). The *rosbag* recordings taken during navigation can be seen in red and magenta in Figure 8.4. Odometry recorded during navigation is the magenta trajectory while red is the fused odometry with IMU (magnetometer) for robot localization.

It can be seen from Figure 8.4 the loop closure error: the start and end point of the trajectory is the same, yet odometry (magenta) measurements have diverged from the robot position.

Figure 8.4: The binary map created by the occupancy grid algorithm using EKF SLAM in the MP AAU Lab. The trajectory used for robot localization can be seen drawn in red.

For testing all algorithms designed in ROS, a trajectory drawing a 2×2 square has been designed for the robot to navigate 5 times. The square has been clearly marked and measured on the floor to visually inspect robot's motion. It can be seen in Figure 8.5. This square trajectory is also drawn on the ROS map and acts as the ground-truth for checking the accuracy of the position estimates - see Figure 8.6. The accuracy of all pose estimators is shown in Table 8.2. Tests are conducted on the lead robot only.

Table 8.2 also indicates the figures made for comparison between the best position estimates from the table. Figures are used to illustrate robot's motion compared to the ground-truth. Figure 8.7 shows the difference between position estimations at lap 1 and 4 of the trajectory navigation. The ground-truth is marked with red. Figure 8.8 shows a comparison between fused odometry (odometry+IMU+GoT) and SLAM+fused odometry. Figure 8.9 compares the GoT position estimates of the robot and the corrected GoT estimated. A caveat regarding the corrected GoT is that patching was performed at a resolution of 1 m at the time of the tests. Much of the patching information can be lost due to high resolution.

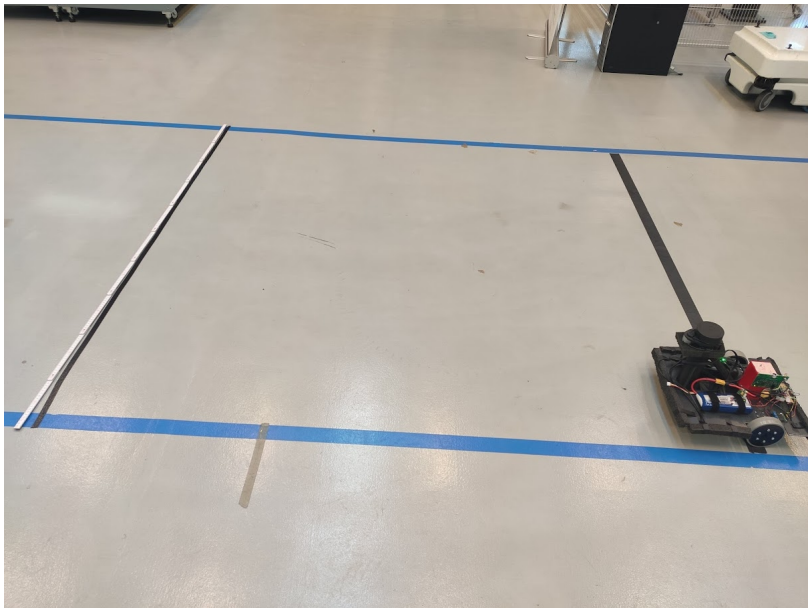


Figure 8.5: The 2×2 square marked and measured on the laboratory floor to visually validate the motion of the robot.

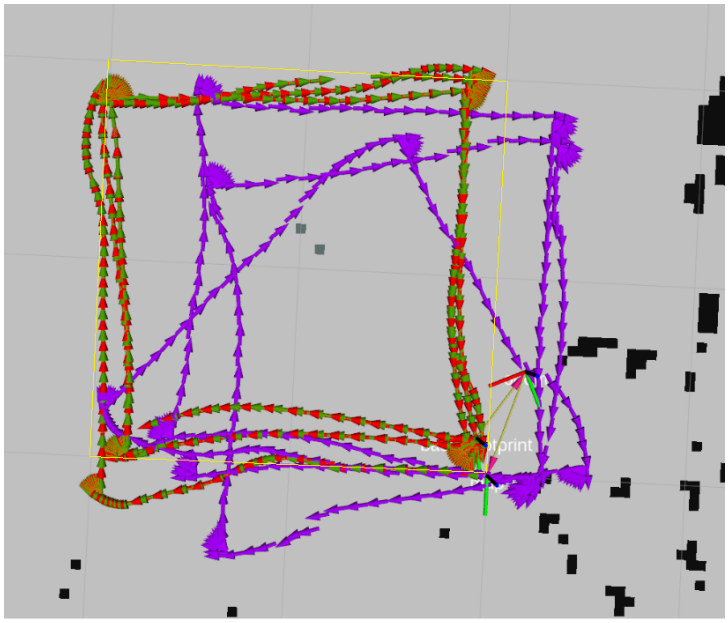


Figure 8.6: The 2×2 square marked (yellow) on the ROS laboratory map along with the measurements from fused odometry:odometry+IMU (red), fused odometry:odometry+IMU+corrected GoT (green) and odometry (magenta).

Laboratory Testing		**Lead Robot**				
		1st Run		*4th Run*		
EKF Method	**Signal**	Loop Closure Error (x,y)[m]	Signal Accuracy Std.Dev. (x,y)[m]	Loop Closure Error (x,y)[m]	Signal Accuracy Std.Dev. (x,y)[m]	Figure
Fused Odometry	Odometry + IMU	0.22, 0.31	0.61, 0.5	0.45, 0.5	0.59, 0.54	-
	Odometry + GoT + IMU	0.014, 0.014	0.68, 0.58	0.006, 0.15	0.62, 0.58	8.7
	Odometry + patched GoT + IMU - only client robot -	0.15, 0.64	0.63, 0.54	0.35, 0.2	0.74, 0.63	-
Fused Localization	EKF SLAM	0.3, 0.34	0.70, 0.55	1.9, 1.1	1.44, 1.33	-
	(Odometry + IMU) + EKF SLAM	0.035, 0.035	0.68, 0.57	0.038, 0.145	0.62, 0.58	-
	(Odometry + GoT + IMU) + EKF SLAM	0.04, 0.035	0.67, 0.56	0.017, 0.19	0.61, 0.56	8.8
GoT	Raw	0.25, 0.25	0.92, 0.74	0.019, 0.45	0.92, 0.83	-
	Patched	0.115, 0.25	0.62, 0.52	0.43, 0.49	0.59, 0.54	8.9

Table 8.2: Result table for the laboratory testing of robot localization using sensor fusion and SLAM for the lead robot performing the square trajectory. Highlighting is done to indicate best position estimate.

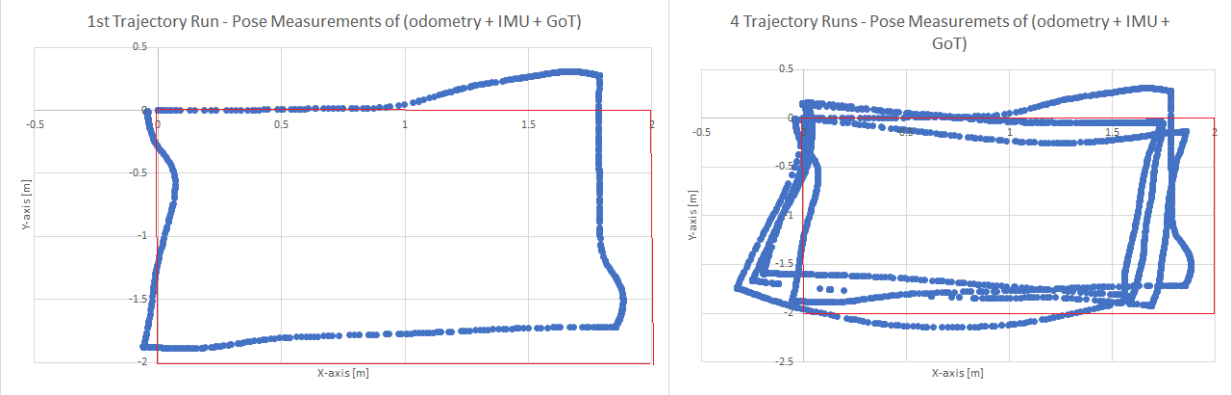


Figure 8.7: Position estimate of the fused odometry (odometry+IMU+GoT) shown for the first lap in the first figure to the left and all 4 laps of the square trajectory in the figure to the right.

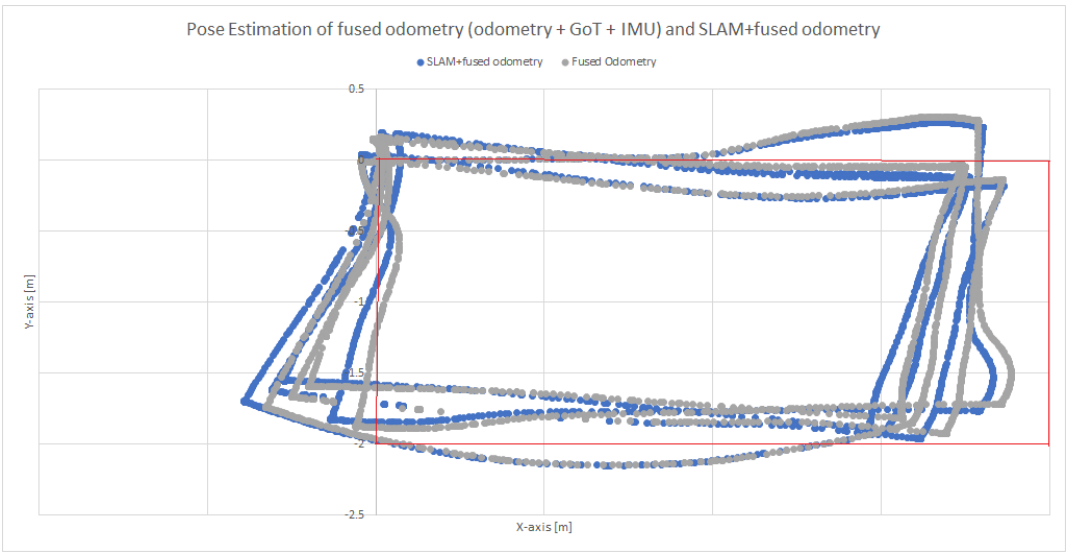


Figure 8.8: Position estimate of the fused odometry (odometry+IMU+GoT) and fused localization for the 4 runs of the trajectory.

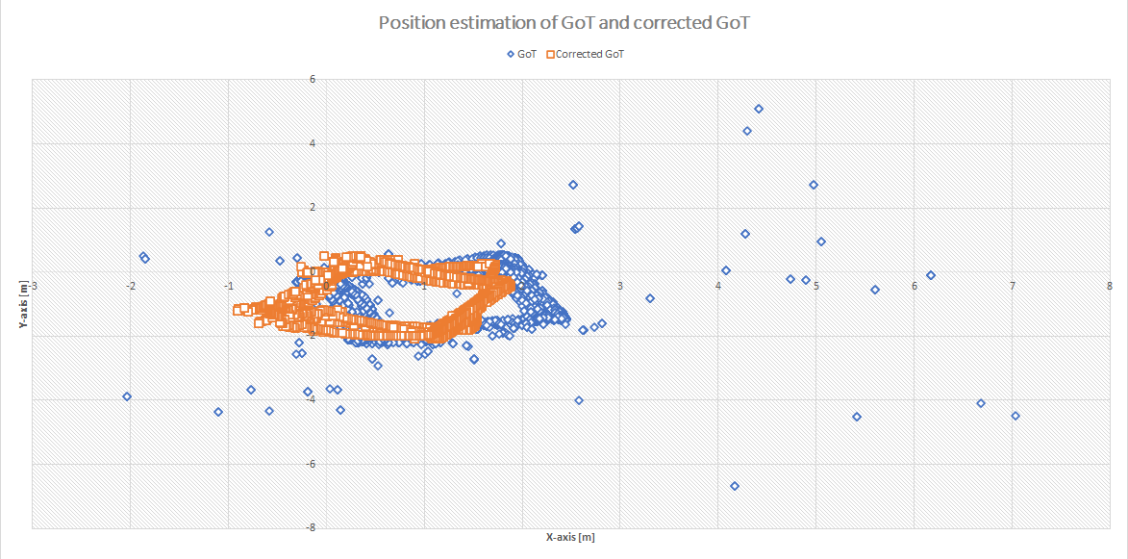


Figure 8.9: Position estimate of the GoT estimates and corrected GoT estimates.

The interpretation of the results obtained in the simulation and laboratory test are presented in the next chapter along with the evaluation of the requirement specifications.

9 Conclusions

This chapter concludes the work performed in this project by exposing the conclusions on the results and requirements achieved. It also mentions recommendations for further research and development of the solution.

9.1 Evaluation

This section presents the interpretation of the test results obtained in Chapter 8 and concludes on the solution developed. Both the simulation and laboratory work results are concluded.

The interpretation of the simulation and laboratory work results is done in relation to Tables 8.1 and 8.2. All best position estimates are summarized in Table 9.1 for the lead robot to visually inspect the differences between simulation and real-world implementation.

EKF Method	**Input Signal**	**Lead Robot**	
		Position Error Standard Deviation (x, y) - Simulation -	**Position Error Standard Deviation (x, y) - Laboratory -**
Fused Odometry	Odometry + IMU	0.05, 0.12	0.59, 0.54
	Odometry + GoT + IMU	0.04, 0.02	0.62, 0.58
Fused Localization	EKF SLAM	1.58, 1.52	1.44, 1.33
	(Odometry + IMU) + EKF SLAM	0.04, 0.02	0.62, 0.58
	(Odometry + GoT + IMU) + EKF SLAM	0.04, 0.02	0.61, 0.56
GoT	Raw	1.19, 0.85	0.92, 0.83
	Patched	0.06, 0.03	0.59, 0.54

Table 9.1: Summarizing table of the accuracy results obtained for robot localization in simulation and laboratory.

In Table 9.1 can be seen that simulation and laboratory results favor the simulation. Both the simulation and laboratory work results suggest EKF SLAM as an inefficient pose estimator. The simulated GoT signal shows higher error in pose estimation than the real one. However, patched GoT has improved accuracy in simulation than in real-world. Simulation can be used as a general guideline on how the systems may behave, but cannot be substituting the results of laboratory work.

9.1.1 Simulation Results Evaluation

Lead Robot

In Table 8.1 the yellow highlight marks the best position estimate for fused odometry, fused localization and GoT. There is clearly an very small error in position estimation for the lead robot regardless whether it is fused odometry or fused localization. The error is close to 0 with the exception of the EKF SLAM which does not include any fused odometry. The error of using just the EKF SLAM algorithm is 1.5 m. It is obvious that EKF SLAM with fused odometry provides a much better position estimate than simply using EKF SLAM.

When it come to GoT accuracy there is a significant difference between raw GoT measurements and patched GoT. While raw GoT has an error of approximately 1 m, the patched GoT error is approximately 0. The resolution for fingerprinting and error patching is 0.1 m.

Looking at the standard deviation for fused odometry, it can be seen that the fusing GoT measurements with odometry and IMU, positively impacts the position error. When it comes to fused localization for the same signals, the impact of GoT on EKF SLAM with fused odometry is not evident, however the signals have the same error.

Overall, for the simulation on the lead robot it can be said that using fused localization (EKF SLAM + fused odometry) significantly improves position accuracy. Moreover, it is recommended that this signal to be used as ground-truth when fingerprinting and error patching the GoT position estimates.

Client Robot

In this project, the results for robot localization on a client robot come only from the simulation environment. As the client robot cannot run a EKF SLAM, it only uses fused odometry to localize. The localization error is approximately 4 m. The results are mixed when it comes to proving the accuracy of each signal. While looking at the standard error, the fused odometry with the patched GoT signal is marginally better than the fused odometry with raw GoT. These two signals are marginally better than fused odometry with no GoT signal. This is also proved by the fact that the error between raw GoT and patched GoT is similar.

Summarizing the simulation results for the client robot, while the results proving that using the patched GoT for fused odometry is marginally better than raw GoT for fused odometry, it does show an increase in localization accuracy compared to fused odometry without GoT.

9.1.2 Laboratory Results Evaluation

These reults concern only the lead robot. The square trajectory was lapped 4 times and in Table 8.2 the results for the 1^{st} and 4^{th} lap are shown. Generally there is slight error increase in the results observed from the 1^{st} to the 4^{th} lap. This suggests that odometry drift impacts both fused odometry and fused localization, however the error remains bounded. Graphically, Figure 8.7 shows the 1^{st} lap in the figure to the left. It can been that this lap has the highest deviation from ground-truth on the x-axis compared to the rest. In the figure to the right, the deviation from the ground-truth on the y-axis is higher on all laps except the 1^{st}.

Figure 8.8 shows the difference in pose estimation between fused odometry (odometry + IMU + GoT) and fused localization (EKF SLAM + odometry + IMU + GoT). After a visual inspection, it is observed that the fused odometry is a better estimator than fused localization as it is closer to the ground-truth. To verify this statement, Table 8.2 show that these two signals have a similar standard error of approximately 0.7 m on the x-axis and approximately 0.6 m on the y-axis for the entire trajectory. The error of the fused localization signal (EKF SLAM + odom + IMU + GoT) is slightly lower than the fused odometry signal. It is then recommended to use this signal as a ground-truth for fingerprinting and GoT error patching.

It is observed that fused odometry with GoT has a slight increase in error compared to fused odometry (odometry + IMU) without GoT. When looking at the loop closure error however, the difference between the two signal is noticeable. Loop closure is what ultimately defines the performance of the signal as it shows how close to the desired position the localization estimate is. The loop closure error after the 1^{st} lap of the fused odometry with GoT shows an accuracy improvement from fused odometry without GoT of 0.2 m on the x-axis and 0.3 m on the y-axis. After the 4^{th} lap there is an accuracy improvement of 0.4 m on both axes. On the same note, fused localization (EKF SLAM + fused odometry) shows a slight improvement in position accuracy compared to fused odometry.

The patched GoT signals shows a smaller error in standard deviation than raw GoT, an improvement of 0.3 m. Loop closure error shows mixed results where the patched GoT signal has an improved accuracy of 0.1 m on the x-axis in the first run, however it shows the same results as raw GoT on the y-axis. A caveat for this test results regarding patched GoT is that the resolution of error patching is at 1 m. This means that for a range 1 m there is one only error applied. The raised the issue that if GoT measurements presents several outlier measurements in this 1 m range, the error is not recorded. The solution was to update the error vector with a higher error if found within the 1 m range. This solution is not reflected in the test results shown in Table 8.2. The improvement in accuracy when lowering the error patching resolution is expected following the results received in the simulation where a precision of 0.1 m was used for the patched GoT. The standard deviation of error is approximately 0 for the patched signal.

The error in standard deviation between raw GoT and patched GoT is significantly improved with an increase in accuracy of 0.3 m on both axes on the 1^{st} lap and 0.4 m on the 4^{th} lap. This suggests that the solution mentioned above might yield further improvement in accuracy for the trajectory navigation.

Optionally, a test was run for a fictive client robot receiving fused odometry with patched GoT. The standard deviation error in the 1^{st} lap is decreased by 0.05 m compared to fused odometry with raw GoT but it increased by 0.12 m in the 4^{th} lap. When it comes to loop closure error it is higher than fused odometry with raw GoT with maximum 0.5 m but it is improved compared to fused odometry (odometry + IMU) by at least 0.1 m. A caveat for this patched GoT besides the aforementioned high resolution of 1 m, the ground-truth used by the lead robot to correct the error is fused odometry (odometry + IMU). This clearly is not the best estimate for ground-truth and definitely an improvement in accuracy of the corrected GoT is expected.

Summarizing the laboratory tests results the following conclusions can be drawn:

- Fusing odometry with GoT measurements shows an improved loop closure error of at least 0.2 m compared to fused odometry with no GoT measurements;
- Fused localization (EKF SLAM + fused odometry) bring a further improvements in loop closure between 0.05 m to 0.2 m;
- Standard deviation of the error between the four signals: fused odometry with-/without GoT, fused localization with/without GoT is similar;
- Fused localization with GoT is to be used as ground-truth estimate for fingerprinting and error patching;
- Error patching the GoT signal shows a significant accuracy improvement in the simulation environment with a resolution of 0.1 m, compared to the decreased improvement of the patched GoT signal used in the laboratory with a resolution of 1 m;
- Fused odometry with patched GoT for the client robot showed improved accuracy compared to the fused odometry (odometry + IMU) but no improvement compared to fused odometry with GoT. The accuracy is expected to increase if the resolution of the error patching is lowered and the ground-truth estimate used by the lead robot is fused localization with GoT;
- The simulation results suggest using the patched GoT signal alone for the client robot is better than fusing it with odometry and IMU.

The goal of the project was declared in Section 2 as:

Develop a low-cost multi-robot solution to improve accuracy of localization estimation through SLAM, sensor fusion and fingerprinting. The multi-robot setup is comprised of a lead robot and a client robot. The lead robot uses SLAM and sensor fusion with sensors as encoders, magnetometer, LiDar and GoT to fingerprint GoT interference-affected locations. The client robot uses encoders, magnetometer and the interference-corrected GoT signal. The increased accuracy of localization estimation improves robot autonomy in map navigation.

The goal is generally considered as attained with the mention that proper testing and results are still to be performed on a real-world client robot. This is a cross-disciplinary project and there are several recommendations on continuing to improve the results received for this work. These can be seen in the Section 9.3. Following is the evaluation of the requirement specifications set for this project.

9.1.3 Requirements Specifications Evaluation

This section evaluates and discusses the fulfillment or partial fulfillment of the requirement specifications presented in Chapter 1.

Requirement specifications on the robot design

R1. Establish a reproducible wiring of all components for a 2^{nd} robot.
Status: Fulfilled.
Evaluation: A clear architecture was designed to distribute the components between low and high level of user involvement. This structure can be reproduced on other robots from the swarm.

R2. Finalize a unified hardware design of the robot.
Status: Not fulfilled.
Evaluation: The current design is a working prototype however it is not ready for deployment as it lacks robustness in design and lacks a place for a container.

R3. Build a 2^{nd} robot.
Status: Not fulfilled.
Evaluation: Components for a second robot did not arrive in the project's timeframe.

Requirement specifications on the interface between low-level and high-level architecture

R1. Wrap all sensor data in appropriate ROS message type.
Status: Fulfilled.
Evaluation: Data transmitted by all sensors is standardized according to the accepted ROS message types.

R2. ROS Master receives all sensor data.
Status: Fulfilled.
Evaluation: ROS Master connects to all nodes created for sensor data transmission.

Requirement specifications on ROS Framework

R1. Wrap sensor data into message types accepted by *robot_localization* package.
Status: Fulfilled.
Evaluation: A deeper understanding of frames and transforms was necessary to use sensor data for fusion.

R2. Wrap sensor data into message types accepted by *hector_slam* package.
Status: Fulfilled.
Evaluation: Data sent by *hector_slam* is used in the *robot_localization* package.

R3. Simulate a GoT system in ROS.
Status: Fulfilled.
Evaluation: Multiple considerations about GoT behaviour in real life had to be made in order to simulate the signal. The ROS GoT simulates the real GoT and can be used as regular node in ROS.

R4. Simulate the real robot in Gazebo.
Status: Partially fulfilled.
Evaluation: The real robot was simulated in the URDF file using the constant velocity model however there were problems with the motion of the robot in Gazebo as it was requesting dynamics. Dynamics were modelled with high damping and low inertia. The simulated robot fails to move accordingly.

R5. Simulate a navigation algorithm.
Status: Fulfilled.
Evaluation: 2 algorithms for navigation have been developed: the Bug1 algorithm and the *go_to_point*. These can be placeholders for an advanced path planning interface such as AAU software Uppaal.

R6. Simulate 2 robots in a map with different sensor configurations.
Status: Fulfilled.
Evaluation: The lead robot is configured with all sensors for SLAM (odometry, LiDar, IMU, GoT) and a blind robot is equipped with encoder for odometry, magnetometer for orientation and GoT.

Requirement specifications on sensor fusion and GoT

R1. Fingerprint all locations in Gazebo and laboratory where GoT transmits erroneous

measurements higher then threshold of 0.2 m.
Status: Partially fulfilled.
Evaluation: In simulation, although the lead robot has mapped the environment, not all locations on the map were fingerprinted as client robots were able to find locations were server call for error patching returned 0. In the laboratory, this criteria cannot be met due to missing a second robot.

R2. Patch the GoT signal at the fingerprinted locations with a scaling parameter.
Status: Fulfilled.
Evaluation: The scaling parameter s is meant to bring the patched GoT signal closer to the ground-truth, given that the ground-truth estimate is found.

R3. Observe an improvement in fused odometry with patched GoT of over 0.1 m in Gazebo for the client robot when passing in a fingerprinted location on the map.
Status: Partially fulfilled.
Evaluation: While some measurements indicate a decrease in the loop closure error of 5 m, there are also increased loop closure errors of 7 m. Standard deviation error shown a slight improvement compared to fused odometry without GoT. If comparing fused odometry (odometry + IMU + patched GoT) with the patched GoT signal, the requirement is not fulfilled as patched GoT is significantly better.

R4. Observe an improvement in fused odometry of over 0.1 m in laboratory for the client robot when passing in a fingerprinted location on the map.
Status: Not fulfilled.
Evaluation: As there is no second robot, the results obtained for fused odometry using the lead robot as a client robot are not conclusive but the values seem to confirm the simulation results: for the blind robot use only the patched GoT signal not fused odometry with patched GoT.

R5. Observe an improvement in fused localization with GoT measurements of over 0.1 m in Gazebo for the lead robot.
Status: Not fulfilled.
Evaluation: Fused localization with GoT is similar in results with fused odometry with GoT and both have mixed results compared to fused odometry with no GoT input.

R6. Observe an improvement in fused localization with GoT measurements of over 0.1 m in laboratory for the lead robot.
Status: Fulfilled.
Evaluation: The ranges of improvements is not constant over the map not over the two axes, however there have been accuracy improvements of 0.1 m compared to all other signals.

R7. Finalize a plug-and-play ROS infrastructure for the 2D robotic solution.

Status: Fulfilled.
Evaluation: ROS Master runs on RPi connected to the Teensy3.6 and the remote workstations. The launchfile that contains all necessary nodes, topics, parameter is at the core of launching the same configuration over multiple robots. The ROS Master so far tracks 20 nodes and 15 topics.

9.2 Perspective

This section discusses few issues that were crucial for the work to take place but were not mentioned in the report.

1. One of the repeating troubles of working with a prototype is the robustness of the hardware: unsoldered wires, LiPo battery swell, battery long charging time, unmounting LiDar from bumps, recalibrations, broken USB cables, etc.

2. Create a special network for the devices that need to connect and see each other on the network, as secured network do not allow this option. Moreover, Wi-Fi hotspot becomes inefficient for ROS infrastructure updates. On the current implementation the RPi hosts the ROS Master and the remote workstation is just a node connecting to the Master and logging data. Most often the RPi would not connect to a Wi-Fi hotspot. A large-scale implementation requires ROS to be hosted on another computer than the lead robot but on the same network. Logging data is requested by the remote workstation from the ROS Master. A SSH connection can not handle large amounts of data. Hence, networking is an important infrastructural component when considering swarms.

3. Computers used on the lead and client robots are important when it comes to computational speed and connectivity to the network. RaspberryPi 3 handled at a minimum the ROS infrastructure required to computed and log all necessary data. When adding a client robot, updating the computer should be a priority.

4. Designing and finalising the ROS infrastructure needed a steep learning curve. The documentation for ROS is often lacking or assumes the user has a specific ROS robot setup e.g. Turtlebot. Finding the right connections and parameters for a robot outside specifications, took a considerable amount of time.

5. Handling two media: simulation and laboratory work using the same ROS infrastructure gets confusing with the number of nodes created and the knowledge capital needed to maintain and run the ROS infrastructure is not documented. A solution to this has been to provide the tutorial on setting up the ROS infrastructure found in the Appendix K.

9.3 Future Work

This section describes the next steps that can be taken to continue and improve this research.

The work has started with building the hardware design of the first robot. After the final prototype has been done it became evident that for computationally intensive algorithms like SLAM and ROS, RaspberryPi 3 was not sufficient. It is hence important for the next steps to update the prototype with a RaspberryPi 4 or an Intel NUC computer. Furthermore, the connection between the computer (RPi) and the remote workstation must allow devices to see each other on the network. The current setup tested a one-to-one device connected to a Wi-Fi hotspot. The network where one-to-more devices are connected was not tested and it is expected to introduce network delays. With an increasing number of robots the effects of network delay are expected to negatively impact the localization accuracy of client robots.

Build the client robot and test the actual performance of the ROS infrastructure to sustain the nodes between the lead and client robot communicating without network delay. Test the performance of the GoT error patching for the client robot. The hardware design of the lead and client robots can continue by finalizing the design of the robots allowing space for containers or packets to be delivered by the robot. ROS integration with a path planning algorithm and a production planner can be done. A specialized path planning and tracking algorithm such as Uppaal can be interfaced with ROS to send waypoints to the robots on where to go just as the *go_to_point* script currently does.

Given the results obtained with the EKF SLAM which were marginally improving the sensor fusion for robot localization and knowing the main disadvantage of the EKF - linearization about one equilibrium point, research can advance in the direction of the UKF. Other filters such as particle filter can replace altogether the SLAM and sensor fusion methodology and compare the performance in accuracy.

Regarding sensor fusion other sensors can be used in order to improve localization and mapping. Sensors such as an RGB camera can improve localization and other types of SLAM algorithms can be used such as collaborative SLAM. EKF SLAM can also be extended with the scan-to-map algorithm for improved mapping and navigation. An integrated approach between localization, mapping and navigation can be taken. A comparative research between the type of SLAM algorithms for a multi-robot setup as described in this project can be performed.

Industrial environments are dynamic places with people and other moving robots. Even in dark factories where people are not present, robots should be able to avoid other robots. For the multi-robot setup in this work, it would mean that the path planner is aware of all moving obstacles and can plan trajectories for robots not to intersect.

Bibliography

[1] UK-RAS Network Robotics and Autonomous Systems. Manufacturing robotics - the next robotic industrial revolution, 2016. URL https://www.ukras.org/wp-content/uploads/2018/10/UK_RAS_wp_manufacturing_web.pdf.

[2] allaboutlean.com. A critical look at industry 4.0, 2015. URL https://www.allaboutlean.com/industry-4-0/.

[3] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[4] aethon.com Tony Melanson. What industry 4.0 means for manufacturers, 2018. URL https://aethon.com/mobile-robots-and-industry4-0/.

[5] Michael Rüßmann Manuela Waldner Jan Justus Pascal Engel Boston Consulting Group Philipp Gerbert, Markus Lorenz and Michael Harnisch. Mobility industry 4.0: The future of productivity and growth in manufacturing industries, 2015. URL https://www.bcg.com/publications/2015/engineered_products_project_business_industry_4_future_productivity_growth_manufacturing_industries.aspx.

[6] aethon.com Spencer Allen. Self-driving robot navigation methodology explained, 2018. URL https://aethon.com/our-navigation-methodology-explained/.

[7] Ross Mead, Rob Long, and Jerry B Weinberg. Fault-tolerant formations of mobile robots. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4805–4810. IEEE, 2009.

[8] Pedro M Shiroma and Mario FM Campos. Comutar: A framework for multi-robot coordination and task allocation. In *2009 IEEE/RSJ international conference on intelligent robots and systems*, pages 4817–4824. IEEE, 2009.

[9] Jian Tang, Yuwei Chen, Liang Chen, Jingbin Liu, Juha Hyyppä, Antero Kukko, Harri Kaartinen, Hannu Hyyppä, and Ruizhi Chen. Fast fingerprint database maintenance for indoor positioning based on ugv slam. *Sensors*, 15(3):5311–5330, 2015.

[10] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on robotics*, 32(6):1309–1332, 2016.

[11] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on robotics*, 32(6):1309–1332, 2016.

[12] Alonzo Kelly. *Mobile robotics: mathematics, models, and methods*. Cambridge University Press, 2013.

[13] Paul Newman, John Leonard, Juan D Tardós, and José Neira. Explore and return: Experimental validation of real-time concurrent mapping and localization. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, volume 2, pages 1802–1809. IEEE, 2002.

[14] Simon Lynen, Torsten Sattler, Michael Bosse, Joel A Hesch, Marc Pollefeys, and Roland Siegwart. Get out of my lab: Large-scale, real-time visual-inertial localization. In *Robotics: Science and Systems*, volume 1, 2015.

[15] Anastasios I Mourikis and Stergios I Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3565–3572. IEEE, 2007.

[16] Mark Maimone, Yang Cheng, and Larry Matthies. Two years of visual odometry on the mars exploration rovers. *Journal of Field Robotics*, 24(3):169–186, 2007.

[17] E Ackerman. Dyson's robot vacuum has 360-degree camera, tank treads, cyclone suction. *IEEE Spectr*, 2014. URL `https://spectrum.ieee.org/automaton/robotics/home-robots/dyson-the-360-eye-robot-vacuum`.

[18] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*, volume 1. MIT press Cambridge, 2000. Images to download were available on website as of June 2020: www.probabilistic-robotics.org.

[19] Faheem Ijaz, Hee Kwon Yang, Arbab Waheed Ahmad, and Chankil Lee. Indoor positioning: A review of indoor ultrasonic positioning systems. In *2013 15th International Conference on Advanced Communications Technology (ICACT)*, pages 1146–1150. IEEE, 2013.

[20] Santosh Subedi and Jae-Young Pyun. Practical fingerprinting localization for indoor positioning system by using beacons. *Journal of Sensors*, 2017, 2017.

[21] Lasse Klingbeil, Michailas Romanovas, Patrick Schneider, Martin Traechtler, and Yiannos Manoli. A modular and mobile system for indoor localization. In *2010 International Conference on Indoor Positioning and Indoor Navigation*, pages 1–10. IEEE, 2010.

[22] www.gamesontrack.com. Indoor positioning apps, Extracted: 2020. URL `http://www.gamesontrack.com/pages/webside.asp?articleGuid=164202`.

[23] Olson Edwin. A primer on odometry and motor control, 2004. URL `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-186-mobile-autonomous-systems-laboratory-january-iap-2005/study-materials/odomtutorial.pdf`.

[24] Ubiquity robotics pi system images, . URL https://downloads.ubiquityrobotics.com/pi.html.

[25] Ubuntu install of ros melodic, . URL http://wiki.ros.org/melodic/Installation/Ubuntu.

[26] Yong Li and Changxing Shi. Localization and navigation for indoor mobile robot based on ros. In *2018 Chinese Automation Congress (CAC)*, pages 1135–1139. IEEE, 2018.

[27] tf2 - ros documentation, . URL http://wiki.ros.org/tf2.

[28] Tully Foote. tf: The transform library. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6. IEEE, 2013.

[29] John J Craig. *Introduction to robotics: mechanics and control, 3/E*. Pearson Education India, 2009.

[30] Nikolaus Correll. *Introduction to autonomous robots*. 2016.

[31] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[32] Howie M Choset, Seth Hutchinson, Kevin M Lynch, George Kantor, Wolfram Burgard, Lydia E Kavraki, and Sebastian Thrun. *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.

[33] Lucas G.W. An elementary model for the differential steering system of robot actuators, 2000. URL http://rossum.sourceforge.net/papers/DiffSteer/DiffSteer.html.

[34] H. Harry Asada. Introduction to robotics - chapter 2: Actuators and drive systems, 2020. URL https://ocw.mit.edu/courses/mechanical-engineering/2-12-introduction-to-robotics-fall-2005/lecture-notes/chapter2.pdf.

[35] iarobotics. Got$_s lam$, 2019. URL.

Gazebo - robot simulation made easy., . URL http://gazebosim.org/.

Urdf - ros documentation, . URL http://wiki.ros.org/urdf.

Turtlebot3 simulation, . URL http://emanual.robotis.com/docs/en/platform/turtlebot3/pc_setup/.

Gazebo plugins in ros, . URL http://gazebosim.org/tutorials?tut=ros_gzplugins&cat=connect_ros.

Hector slam - ros documentation, . URL http://wiki.ros.org/hector_slam.

Thomas Moore and Daniel Stouch. A generalized extended kalman filter implementation for the robot operating system. In *Intelligent autonomous systems 13*, pages 335–348. Springer, 2016.

C++ reference - unordered map, . URL https://en.cppreference.com/w/cpp/container/unordered_map.

Raymond A Serway and John W Jewett. *Physics for scientists and engineers with modern physics*. Cengage learning, 2018.

SAMS Blog. Magnetometer calibration - calculating orientation pt3, 2016. URL http://www.camelsoftware.com/2016/03/13/imu-maths-calculate-orientation-pt3/.

Setup and configuration of the navigation stack on a robot - ros documentation, . URL http://wiki.ros.org/navigation/Tutorials/RobotSetup.

Turtlebot3 simulation, . URL http://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/#ros-1-simulation.

A Components Wiring to Teensy3.6 Pins

Component	Wire	Pin	Set as
Motor A	PWM	10	Output
	INA	11	Output
	INB	12	Output
Motor B	PWM	30	Output
	INA	28	Output
	INB	26	Output
Encoder A	Yellow	2	Input
	White	3	Input
Encoder B	Yellow	5	Input
	White	6	Input
IMU	SDA	18	Input
	SCL	19	Input
Voltage Divider	GND	GND	-
	3.3V	3.3V	-
	Read	23	Input
Power Brick	GND	GND	-
	5V	Vin	-
GoT	GND	GND	-
	RX	21	Input

Table A.1: Components attached to Teensy3.6. Connections from components to Pins can be seen.

B The KF and EKF

The Kalman Filters
This section presents the KF and EKF filters. Both types make use of the normal distributions and linearity of systems. As such two conditions need to be satisfied in order to apply the filter on a system:

- state transition probability $p(x_t|x_{t-1}, u_t)$ is linear with added Gaussian noise [1] [18].
- measurement probability $p(z_t|x_t)$ is linear with added Gaussian noise [2] [18].

KF for Linear Systems
Defining a linear system of the form that satisfies the conditions above:

$$x_t = A_t x_{t-1} + B_t u_t + \omega_t \tag{B.1}$$
$$z_t = C_t x_t + \upsilon_t \tag{B.2}$$

In the linear system above, A_t is a square state transition matrix of dimension $n \times n$, where n is the dimension of the state vector x_t, B_t is the control input matrix of dimension $n \times m$ where m is the dimension of the control input vector u_t, C_t has dimensions $k \times n$ where k is the dimension of the measurements vector z_t.

As Eq. B.1 defines the state transition probability, this can be computed by plugging Eq. B.1 into Eq. 4.1 resulting into Eq.B.3 as mentioned in [18]:

$$p(x_t|x_{t-1}, u_t) = \det(2\pi R_t)^{-\frac{1}{2}} \exp(-\frac{1}{2}(x_t - A_t x_{t-1} - B_t u_t)^T R_t^{-1}(x_t - A_t x_{t-1} - B_t u_t)) \tag{B.3}$$

From the superposition principle, it can be observed that the mean of the posterior B.3 is given by $A_t x_{t-1} + B_t u_t$ and co-variance R_t. The same follows for the measurement probability by the same operation Eq. B.4 is obtained:

$$p(z_t|x_{t-1}, u_t) = \det(2\pi Q_t)^{-\frac{1}{2}} \exp(-\frac{1}{2}(z_t - C_t x_t)^T Q_t^{-1}(z_t - C_t x_t)) \tag{B.4}$$

Having ensured that the necessary conditions are satisfied, the Kalman algorithm can be applied to the linear system. The Kalman algorithm is shown below [18]:

[1] Gaussian noise added to the state transition probability models uncertainty of the process. Gaussian or white noise has mean 0 and co-variance is here denoted R_t

[2] Gaussian noise added to the measurement probability models uncertainty of the measurements. Gaussian or white noise has mean 0 and co-variance is here denoted Q_t

$$\textbf{Kalman\_filter}(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t): \tag{B.5}$$

Prediction step:

$$\bar{\mu}_t = A_t\mu_{t-1} - B_t u_t \tag{B.6}$$

$$\bar{\Sigma}_t = A_t\Sigma_{t-1}A_t^T + R_t \tag{B.7}$$

Measurement Update step:

$$K_t = \bar{\Sigma}_t C_t^T (C_t\bar{\Sigma}_t C_t^T + Q_t)^{-1} \tag{B.8}$$

$$\mu_t = \bar{\mu}_t + K_t(z_t - C_t\bar{\mu}_t) \tag{B.9}$$

$$\Sigma_t = (I - K_t C_t)\bar{\Sigma}_t \tag{B.10}$$

$$\textbf{Return} \quad \mu_t, \Sigma_t \tag{B.11}$$

The Kalman filters represent distributions through parameterization of moments: mean μ_t and co-variance Σ_t. From the algorithm above, it can be seen that the input to the filter is given by Eq. B.5. The input is composed of the distribution moments at time $t-1$ - μ_{t-1} and Σ_{t-1} which are used in the prediction of moments at time t - $\bar{\mu}_t$ and $\bar{\Sigma}_t$ in Eq. B.6 and B.7; the input also contains control and measurements at time t to update the predicted moments according to Eq. B.9 and B.10. The output of the KF is given by Eq. B.11. Eq. B.8 represents the computation of the Kalman Gain which is further used in the two update equations B.9 and B.10. As the update of the prediction uses the current measurements, the Kalman gain represents the degree to which the measurement is trusted to be incorporated in the update process of the prediction.

Extended Kalman Filter for Non-Linear Systems

As mentioned above, the KF works well if 2 conditions about the system is applied on hold: state and measurements linearity. However, in real life the state and measurements are non-linear with various degrees of non-linearity. If the systems are not highly non-linear, the extended Kalman filter (EKF) can be applied through the process of linearization. After linearization is performed on the system, the normal KF algorithm can be applied. However, if the system is highly non-linear other types of procedures must be used such as the *unscented transform* component of the Unscented Kalman Filter (UKF).

EKF relaxes the assumption of linearity on systems. The non-linear (NL) system is modelled by non-linear function $g(x, u)$ replacing the state transition and control matrices A_t and B_t; and function $h(x)$ replacing matrix C_t. The NL system is shown below:

$$x_t = g(x_{t-1}, u_t) + \omega_t \tag{B.12}$$

$$z_t = h(x_t) + v_t \tag{B.13}$$

As both functions $g(\cdot)$ and $h(\cdot)$ are NL, the resulting distribution obtained from passing the state vector through will be unknown. This means that the resulting distribution is not Gaussian and the KF cannot be applied. EKF is hence a method to estimate the Gaussian distribution - estimates the mean and covariance from the obtained distribution. This estimation is obtained through a process of linearization via first-order Taylor expansion. Linearization approximates the NL function with a tangent linear function. This can be seen in Figure B.1 b) in the middle image.

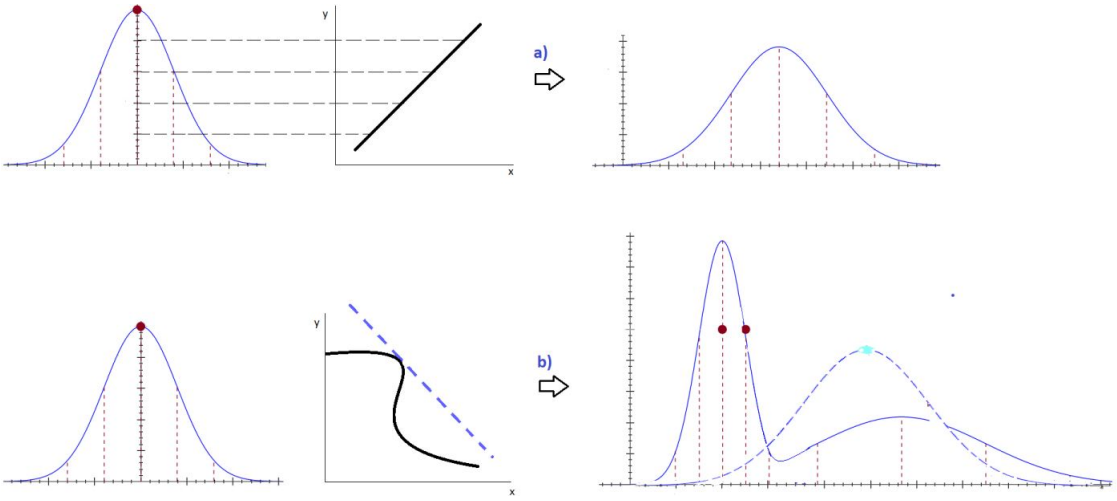


Figure B.1: a) Illustration of a normal distribution passing through a linear function results in a normal distribution with different parameter values than the original distribution. b) The linearization process of EKF. The original Gaussian distribution represented in the lower left corner is passed through the linear function tangent to $g(x_t)$ - dotted line instead of the actual NL function $g(x_t)$. The linearized function is tangent at the corresponding mean of the original Gaussian. The right image shows more details: the dual-modal distribution (continuous line) represents the obtained posterior from passing the original Gaussian through the NL function $g(\cdot)$, the dashed distribution represents the estimated Gaussian or the original Gaussian passed through the linearized function.

In the right image from Figure B.1, the difference between the two estimated Gaussian distributions represents the error due to the loss of information from the linearization of the function $g(\cdot)$. The main advantage of the linearization process stands in its efficiency [18]. Linearization via first-order Taylor expansion entails finding the slope of the linear function through the partial derivative of the NL functions $g(\cdot)$ and $h(\cdot)$. Noting the slope of the linearized function as $g^{'}(\cdot)$ respectively $h^{'}(\cdot)$ then the partial derivative is given by Eq. B.14 and B.15 [18].

$$g^{'}(x_{t-1}, u_t) := \frac{\delta g(x_{t-1}, u_t)}{\delta x_{t-1}} \tag{B.14}$$

$$h^{'}(x_t) := \frac{\delta h(x_t)}{\delta x_{t-1}} \tag{B.15}$$

As mentioned before, the linearization of $g(\cdot)$ is done at the mean of the Gaussian distribution, hence $g(x_{t-1}, u_t)$ is approximated at the value of the mean plus the slope as Eq. B.16 [18].

$$g(x_{t-1}, u_t) \approx g(\mu_{t-1}, u_t) + g^{'}(\mu_{t-1}, u_t)(x_{t-1} - \mu_{t-1}) \tag{B.16}$$

$$g^{'}(\mu_{t-1}, u_t) = G_t \tag{B.17}$$

$$G_t = \frac{\partial g(\mu_{t-1}, u_t)}{\partial x_{t-1}}$$

The same goes for the measurement function $h(\cdot)$ of which estimate is represented by Eq. B.18 [18].

$$h(x_t) \approx h(\bar{\mu}_t) + h^{'}(\bar{\mu}_t)(x_t - \bar{\mu}_t) \tag{B.18}$$

$$h^{'}(\bar{\mu}_t) = H_t \tag{B.19}$$

$$H_t = \frac{\partial h(\bar{\mu}_t)}{\partial x_t}$$

Both estimations of the NL functions are employing the partial derivative with respect to each state contained in the state vector. As the state x_t is a vector then the partial derivative is a matrix called the Jacobian. Hence, G_t is the state transition Jacobian matrix of dimension $n \times n$, while H_t is the measurements Jacobian matrix of dimension $k \times n$.

The EKF algorithm is presented below [18]:

Extended_Kalman_filter$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$: (B.20)

Prediction step:

$$\bar{\mu}_t = g(\mu_{t-1}, u_t) \tag{B.21}$$

$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + R_t \tag{B.22}$$

Measurement Update step:

$$K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + Q_t)^{-1} \tag{B.23}$$

$$\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t)) \tag{B.24}$$

$$\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t \tag{B.25}$$

Return μ_t, Σ_t (B.26)

C Robot Perception in SLAM

This chapter describes the modelling of the environment for which the measurement z_t is based on. As there are a variety of sensors that model the environment - range sensors, cameras, tactile sensors, barcode detector etc., the model of the environment is specific to the sensor available. As mentioned in Section 3.2, the environment-measuring sensor available is a LiDar. A laser-based range scan works by emitting a light-beam and recording its echo. Measurements are based on time-of-flight and light beams are spaced out at one degree increment.

Measurement model

To generate measurement of the environment, a map is needed. A map is a list of the landmarks of the environments [18]. A map also specifies the location and other features of the landmarks. Eq. C.1 shows the notation of a map m and each component is a landmark [18].

$$m = \{m_1, m_2, \dots m_N\} \tag{C.1}$$

The index used for a landmark depends on the type of map[18]:

- feature-based map: in landmark m_n the n is a feature-index; m_n stores the location of the feature [18].
- location-based map: landmark m_n represents a specific location on the map.

Both types of maps are used in robotics with their own differences. The map type used in this project is the occupancy grid map which is a location-based map conferring information about landmark position or their absence (free-space). Feature-based maps confer information about the shape of the landmark at specified locations [18].

Landmarks are used for robot navigation. They represent features of indoor environments: doors, wall corners, etc. and for outdoor environments: building corners, trees, etc. As landmarks are considered static, map are considered as well. However, environments of mobile robots are dynamic: people, other mobile robots, etc.

It was mentioned before that in robot localization, the map is known and 'given' to the robot as a ground-truth. Even more, there is assumed that a correlation function exists between the map and the local map created by the scans of the range scanner. This technique is called map matching or scan-to-map as it was introduced in Chapter 1 in Figure 1.4. A number of consecutive scans are compiled into a local map m_{local} and compared with the known global map m, having a higher probability that the two match if $p(m_{local}|x_t, m)$ is high. This technique has the ability to transform scans into occupancy

maps.

The robot with pose x_t denotes a grid cell of the local map as $m_{x,y,local(x_t)}$ mapped to a global coordinate (x, y) of the ground map [18]. Occupancy maps is part of robot mapping process not robot localization. Scan-to-map is still a robot localization technique that completes the state vector with environment variables.

Scan-matching or scan-to-map uses the likelihood fields to maximize the likelihood of the current pose and map relative to the previous pose and map. Then pose correction is done following Eq. C.2 where $p(z_t|x_t, m_{t-1})$ is the current measurement, m_{t-1} is the map constructed so far and $p(x_t|x_{t_1}, u_{t-1})$ is the pose with u_{t-1} as the motion.

$$x_t = \underset{x_t}{argmax}\{p(z_t|x_t, m_{t-1}) \quad p(x_t|x_{t_1}, u_{t-1})\} \tag{C.2}$$

To calculate the posterior $p(z_t^k|x_t, m_{t-1})$, the end points $\mathrm{z}_t^k of each sensor scan z_t$ are projected into the global coordinates of the map. These are calculated relative to the robot position, the LiDar relative position to the robot $(x_{k,LiDar}, y_{k,LiDar})^T$ and the angular orientation of the beam $\theta_{k,LiDar}$ relative to the robot heading [18]. The end points are mapped onto the global coordinates as in Eq. C.3[18].

$$\begin{pmatrix} x_{z_t^k} \\ y_{z_t^k} \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_{k,LiDar} \\ y_{k,LiDar} \end{pmatrix} + z_t^k \begin{pmatrix} \cos(\theta + \theta_{k,LiDar}) \\ \sin(\theta + \theta_{k,LiDar}) \end{pmatrix} \tag{C.3}$$

Defining d_i in Eq. C.4 as the Euclidean distance between the measurements coordinates $(x_{k,LiDar}, y_{k,LiDar})^T$ and the nearest obstacle m_i in the map, then the probability of the LiDar measurement is $p(z^k|x, m)$ which is d_i capturing the sensor noise, $\varepsilon(d_i)$ where ε represents the uncertainties of the environment modelled as Gaussian noise [18].

$$d_i = \underset{m_{1,x}, m_{1,y}}{min}\{\sqrt{(x_{z_t^k} - m_{1,x})^2 + (x_{z_t^k} - m_{1,y})^2}\} \tag{C.4}$$

The measurements provided by the sensors are not ideal and suffer from noise as depicted in Eq. B.13. Moreover, unexpected objects appearing in front of the robot can give measurements of drastic small ranges compared to other landmarks and include more uncertainty in the localization estimation. Other sensor failure is considered when obstacles are missed altogether due to light-absorbing surfaces or due to sunlight. These types of faults can be added in the robot-environment state vector and estimate their state but more simply, these can be added as sensor noise [18].

D Generalized System Model

The constant velocity model is of the form:

$$x_f = x_i + vdt \tag{D.1}$$

The constant acceleration model is of the form:

$$v_f = v_i + at \tag{D.2}$$

$$a = \frac{v_f - v_i}{t} \tag{D.3}$$

$$v_f^2 = v_i^2 + 2ad \tag{D.4}$$

$$d = v_i t + \frac{at^2}{2} \tag{D.5}$$

$$d = v_f t - \frac{at^2}{2} \tag{D.6}$$

$$d = \frac{(v_i + v_f)t}{2} \tag{D.7}$$

To find the position of a 2D rigid body moving in an uniform motion one of the two models above can be used. The constant velocity model is described in Eq. D.1 [43] where x_f stands for final position, x_i - initial position to which distance is added by travelling with speed v in the period of time Δt. Using the constant velocity model to represent the motion of the 2D robot, matrices A and B can be designed for sensor fusion. The constant acceleration model is described in Eq. D.2 - D.7 [43] where a - acceleration, d - displacement, v_i - initial velocity, v_f - final velocity.

E Mathematical Model of EKF SLAM

The state vector continues to grow with every obstacle observed and mathematical manipulations are needed to handle a growing vector. As motion affects only the robot pose and the landmarks remain unaffected, another way of writing Eq.4.11 is as in Eq.E.1 where F_x^T defines the matrix for all robot pose and 0 otherwise - described by Eq. E.2. The same mathematical principle is applied for the rest of the EKF equations as depicted in the Eq. B.20. The entire EKF SLAM algorithm as depicted in [18] and [31] is shown in Eq. E.5.

$$y_t = y_{t-1} + F_x^T \begin{pmatrix} \frac{\phi_r r + \phi_l r}{2} + \cos(\theta_{t-1} + \frac{\phi_r r - \phi_l r}{2d}) \\ \frac{\phi_r r + \phi_l r}{2} + \sin(\theta_{t-1} + \frac{\phi_r r - \phi_l r}{2d}) \\ \frac{\phi_r r - \phi_l r}{d} \end{pmatrix} \tag{E.1}$$

$$F_x^T = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \end{pmatrix} \tag{E.2}$$

The actual matching or data association happens when the difference between the actual measurement of landmark feature and the predicted feature of the landmark is minimal. The output prediction error can be seen in Eq.E.9. The output prediction is obtained by Eq. E.3. In equation E.4 H_t is calculated taking in consideration the difference between the two estimations - that of the robot pose and of the landmark, where h_t^i is the Jacobian of $h(y_t, N)$ at $\bar{\mu}_t$ with respect to state variables x_t and map landmarks m_N. This equation leads to the computation of the Kalman gain in Eq. E.8.

$$\bar{z_t^i} = \begin{pmatrix} \sqrt{(\mu_{N,x} - \bar{\mu_{t,x}})^2 + (\mu_{N,y} - \bar{\mu_{t,y}})^2} \\ \text{atan2}(\mu_{N,y} - \bar{\mu_{t,y}}, \mu_{N,x} - \bar{\mu_{t,x}}) - \bar{\mu_{t,\theta}} \end{pmatrix} \tag{E.3}$$

$$H_t^i = h_t^i F_{x,N} \tag{E.4}$$

EKF_SLAM$(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t)$:

$$F_x^T = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \end{pmatrix}$$

$$\bar{\mu}_t = \bar{\mu_{t-1}} + F_x^T \begin{pmatrix} \frac{\phi_r r + \phi_l r}{2} + \cos(\theta_{t-1} + \frac{\phi_r r - \phi_l r}{2d}) \\ \frac{\phi_r r + \phi_l r}{2} + \sin(\theta_{t-1} + \frac{\phi_r r - \phi_l r}{2d}) \\ \frac{\phi_r r - \phi_l r}{d} \end{pmatrix}$$

$$G_t = I + F_x^T \begin{pmatrix} \frac{\phi_r r + \phi_l r}{2} + \cos(\theta_{t-1} + \frac{\phi_r r - \phi_l r}{2d}) \\ \frac{\phi_r r + \phi_l r}{2} + \sin(\theta_{t-1} + \frac{\phi_r r - \phi_l r}{2d}) \\ \frac{\phi_r r - \phi_l r}{d} \end{pmatrix} F_x \tag{E.5}$$

$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + F_x^T R_t F_x \tag{E.6}$$

$$Q_t = \begin{pmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\phi^2 \end{pmatrix} \tag{E.7}$$

for all observed features $z_t^i = (r_t^1, \quad \phi_t^i)^T \quad do$:

if landmark N never seen before

$$\begin{pmatrix} \mu_{\bar{N},x} \\ \mu_{\bar{N},y} \end{pmatrix} = \begin{pmatrix} \bar{\mu_{t,x}} \\ \bar{\mu_{t,y}} \end{pmatrix} + \begin{pmatrix} r_t + \cos(\phi_t + \bar{\mu_{t,\theta}}) \\ r_t + \sin(\phi_t + \bar{\mu_{t,\theta}}) \end{pmatrix}$$

$$K_t^i = \bar{\Sigma}_t H_t^{iT} (H_t^i \bar{\Sigma}_t H_t^{iT} + Q_t)^{-1} \tag{E.8}$$

$$\mu_t = \bar{\mu}_t + K_t^i (z_t^i - \bar{z_t^i})) \tag{E.9}$$

$$\Sigma_t = (I - K_t^i H_t^i) \bar{\Sigma}_t \tag{E.10}$$

endfor

$$\mu_t = \bar{\mu}_t \tag{E.11}$$

$$\Sigma_t = \bar{\Sigma}_t \tag{E.12}$$

return μ_t, Σ_t (E.13)

F Magnetometer Calibration

As robot orientation is crucial for navigation, a magnetometer must be calibrated. The magnetometer is a sensor sensitive to Earth's magnetic field strength and outputs its heading relative to Earth's magnetic pole[1]. The magnetic field lines vary in strength and direction about Earth's plane.

The magnetometer reads its orientation relative to the magnetic pole on all three axes. Sensor reading are done on a flat surface away from ferrous deposits as these interfere with the sensor. In order to check whether the magnetometer is affected by interference or fabrication defects, readings of the sensor are done on a flat surface while rotating the sensor. The values read should draw a perfect circle. These readings are the maximum and minimum values of the magnetic field line strength on each of the axes. A magnetometer affected by interference or fabrication defects draws a ellipsoid.
Magnetometer calibration is needed in order to eliminate the effects of the interference. There are two types of interference: hard and soft iron. Hard iron effects refer to the noise sources from the circuit itself or rather fabrication defects. Hard iron effects shift the origin of the circle (2D) or sphere (3D). Soft iron effects come from objects surrounding the magnetometer that distort the magnetic field. These effects stretch and tilt the circle/sphere by making it look like an ellipsoid.

Hard iron effects are compensated for by finding the minimum and maximum values for each axis and calculating their average. This average is decreased from each magnetometer axis reading.

Soft iron effects are more difficult to eliminate and this involves calculating correction coefficients that multiplied with the magnetometer's readings transforms the ellipsoid in a circle/sphere. The correction coefficient is a scale of the average distance from the centre (radius) divided by the average value of respective axes [44].

The calibration algorithm is implemented in the Teensy as shown in snippet F.1.

```

#include "HMC5883L.h"
HMC5883L mag;

int16_t mx, my, mz;
float mx_cal, my_cal, mz_cal;

```

[1]Earth's magnetic field is similar to a dipole magnet. The field originates from a point close to the south pole and terminates at a point close to the north pole. These points are the magnetic poles. The geographic and magnetic poles are not the same - a difference of 11.5 degrees exists between the two. This angle can be added to the magnetic calculations knows as *declination angle* so that the magnetometer shows the heading relative to true north

```
const int MAGXMAX = -110;
const int MAGXMIN = -582;
const int MAGYMAX = 395;
const int MAGYMIN = -92;

mag.getHeading(&mx, &my, &mz);

// hard and soft iron calibration for x,y axes
mx_cal=(float)mx-(float)(MAGXMAX+MAGXMIN)/2;
mx_cal=2*mx_cal/((float)(MAGXMAX-MAGXMIN));

my_cal=(float)my-(float)(MAGYMAX+MAGYMIN)/2;
my_cal=2*my_cal/((float)(MAGYMAX-MAGYMIN));
```

Listing F.1: Hard and Soft Iron Effects - Magnetometer Calibration.

Once the calibration for each magnetometer axis is done, the heading of the robot is calculated using the $atan2$ function on the x and y axis. As mentioned before, the declination angle of the robot's location can be added to the heading measurement as shown in Listing F.2. The final heading measurement should point to True North. The accuracy of the measurement is compared to the heading calculated with the kinematic equations and the results are shown in the Sanity Check Section below.

```

// Calculate heading when the magnetometer is level, then correct for signs
    of axis.
  float heading = atan2(mx_cal, my_cal);

// http://www.magnetic-declination.com/
// Magnetic declination at AAU is: 3gr 22' W, which is 3.366667 Degrees, or
    0.058759423968 radians.
  float declinationAngle = 0.0587;
  heading += declinationAngle;

// Correct for when signs are reversed.
  if(heading < 0)
    heading += 2*PI;

// Check for wrap due to addition of declination.
  if(heading > 2*PI)
    heading -= 2*PI;

// Convert radians to degrees for readability.
   float headingDegrees= heading * 180/PI;
```

Listing F.2: Adding declination angle to the heading measurement.

G RQT Graph of the Simulated Robot SLAM: ROS Nodes and Topics

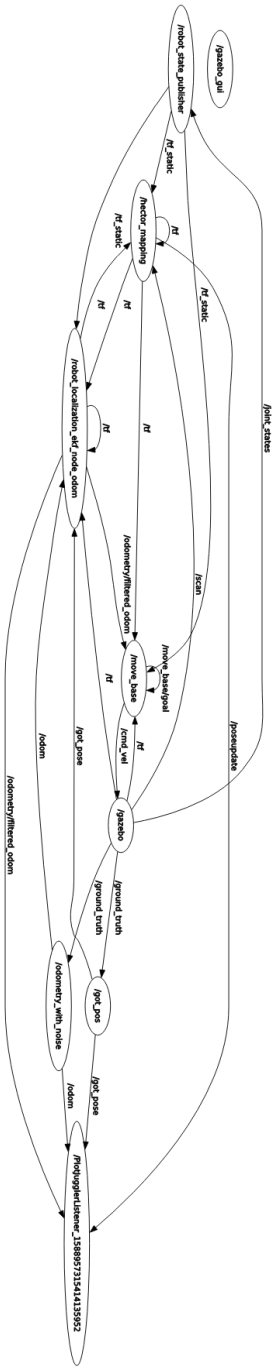


Figure G.1: Graph showing all the running nodes (ellipses) and topics placed on top of arrows (messages.)

H RQT Graph of the Simulated Robot SLAM: ROS Frames

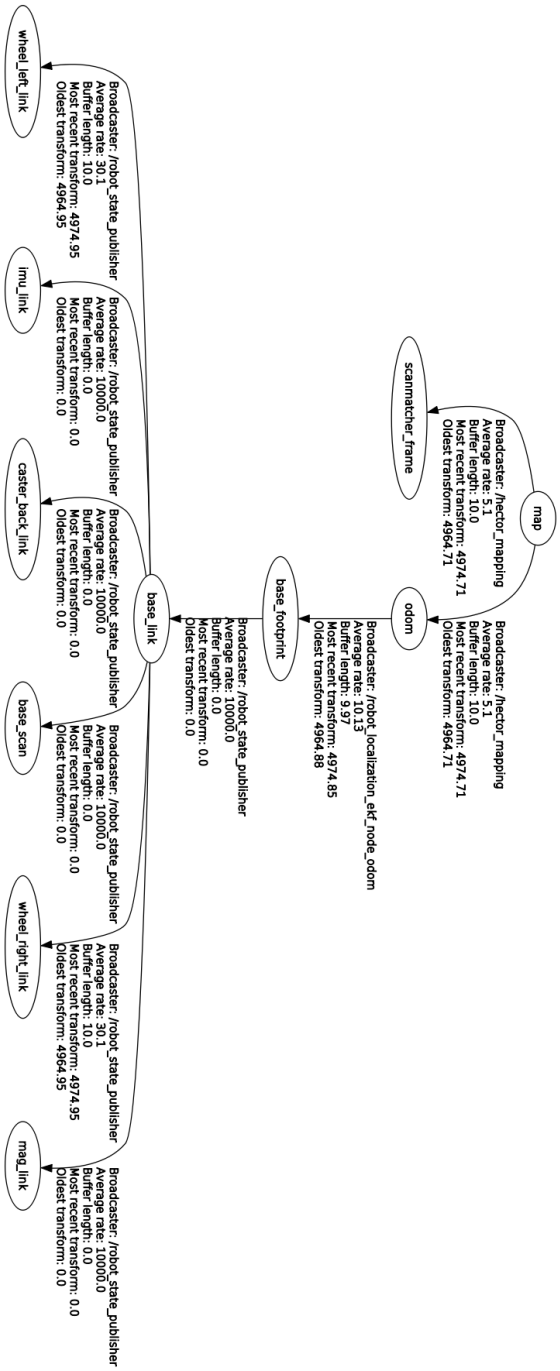


Figure H.1: Graph showing the running frames of the robot. Other details about the frames are also published such as the frequency of frame publishing.

I Simulated Robot in ROS using URDF Files

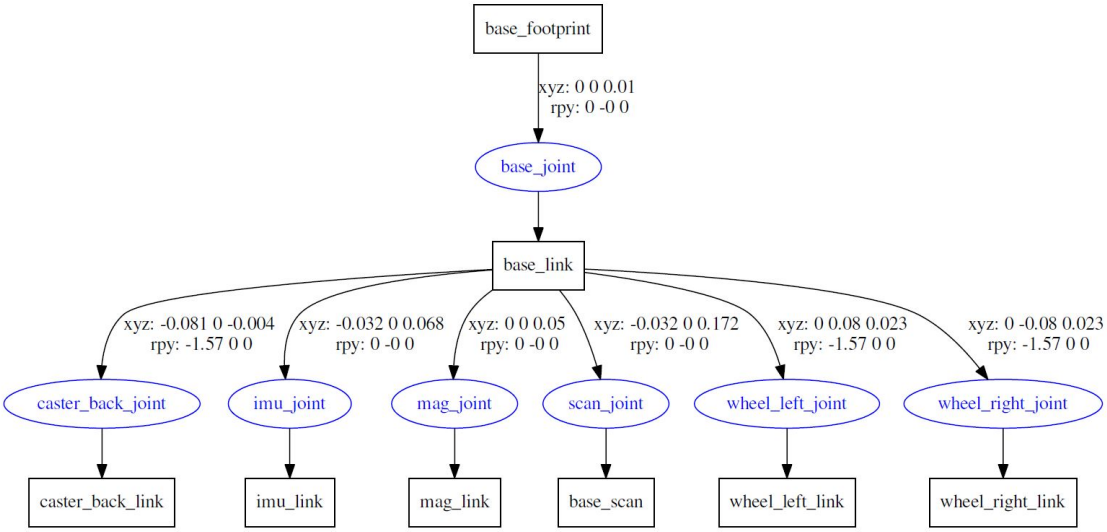


Figure I.1: Graph showing all links and joints created through the URDF file in order to visualize the robot in Gazebo.

J Robot Navigation Methods

This section provides a visualization of different robot navigation algorithms tested throughout this project. Namely bug algorithms and *move_base*. Description of the Bug algorithms can be found in [18] and for move_base in [45].
Bug0 algorithm is attempts to circumnavigate obstacles until there is a free path towards the goal The map is structured such as it exposes a weakness in the Bug0 algorithm such that it is stuck in a loop. Figure J.1 shows the path of the robot in red circumnavigating the obstacle multiple times and never reaching the goal position.

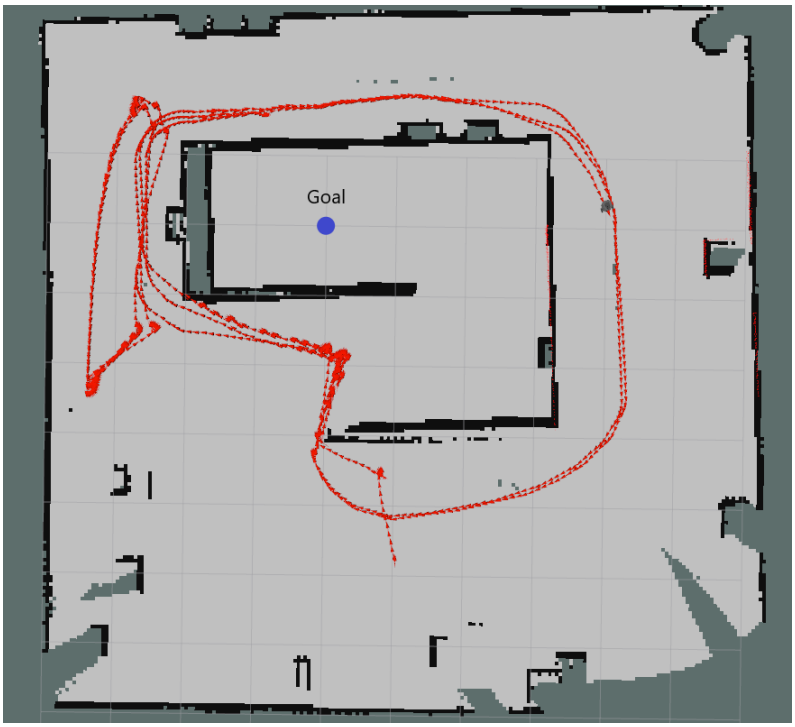


Figure J.1: Path of the robot using Bug0 algorithm

In the case of Bug1 the robot will circumnavigate completely every obstacle in its path to the goal position before deciding from which point to head towards the goal. The robot using Bug1 can reach the goal point that Bug0 can't, as seen in Figure J.2

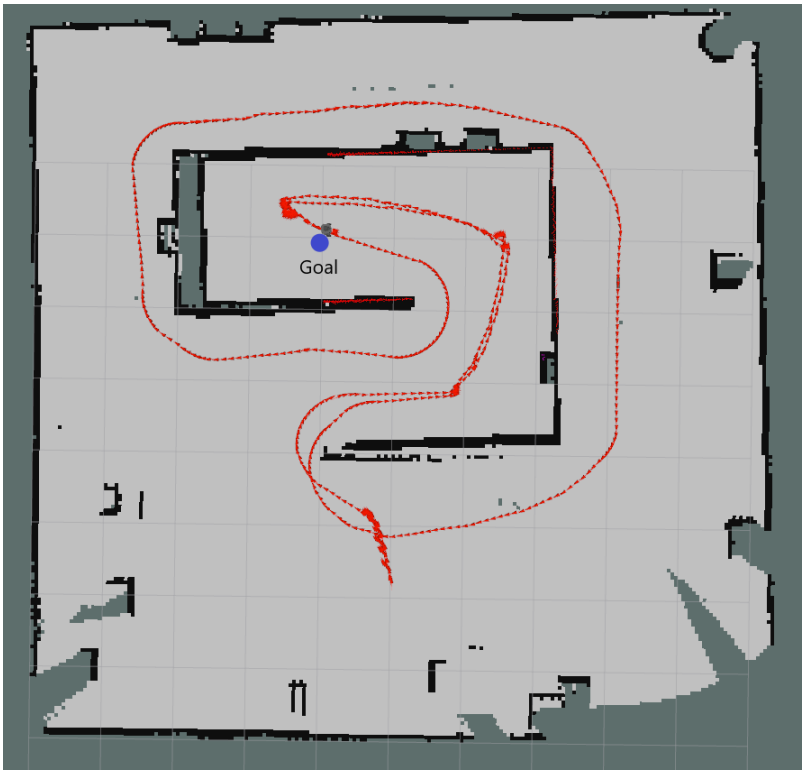


Figure J.2: Path of the robot using Bug1 algorithm

A robot using Bug2 draws a virtual straight line towards the goal point. If an obstacle is encountered, the robot circumnavigates it until it reaches the virtual line again. In this case the robot also can't reach the goal due to the structure of the map in Figure J.3

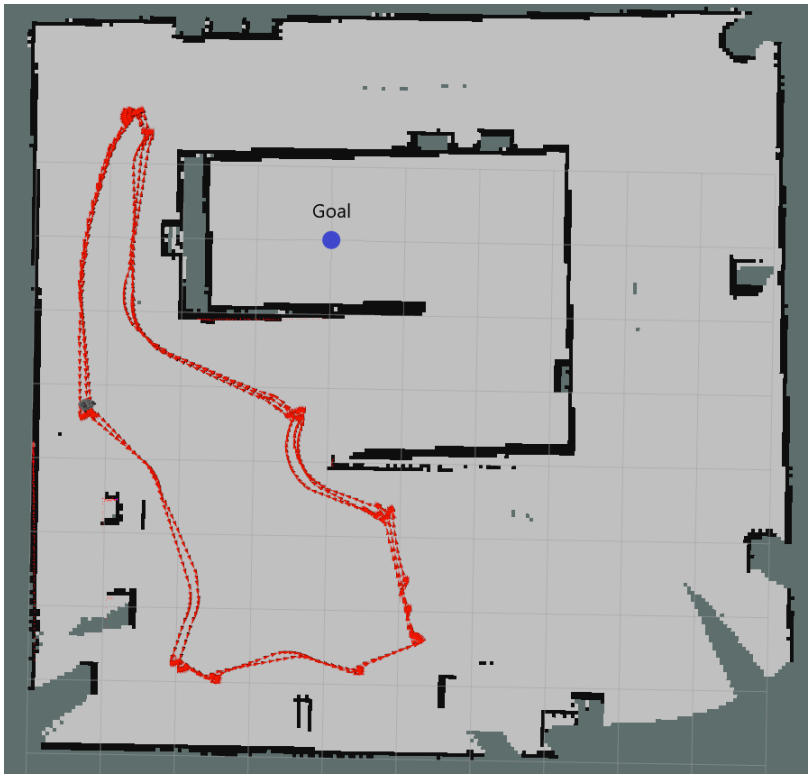


Figure J.3: Path of the robot using Bug2 algorithm

In the last case, the robot is configured to use move_base, a part of the ROS Navigation Stack. It makes use of the map published by a SLAM node or a pre-existing map to construct a cost map by assigning weights to map areas depending on how close they are to an obstacle. In the end it generates a complete path to the goal for the robot to follow. For the path taken by the robot see Figure J.4More information on *move_ base* and the ROS Navigation Stack can be found in[45].

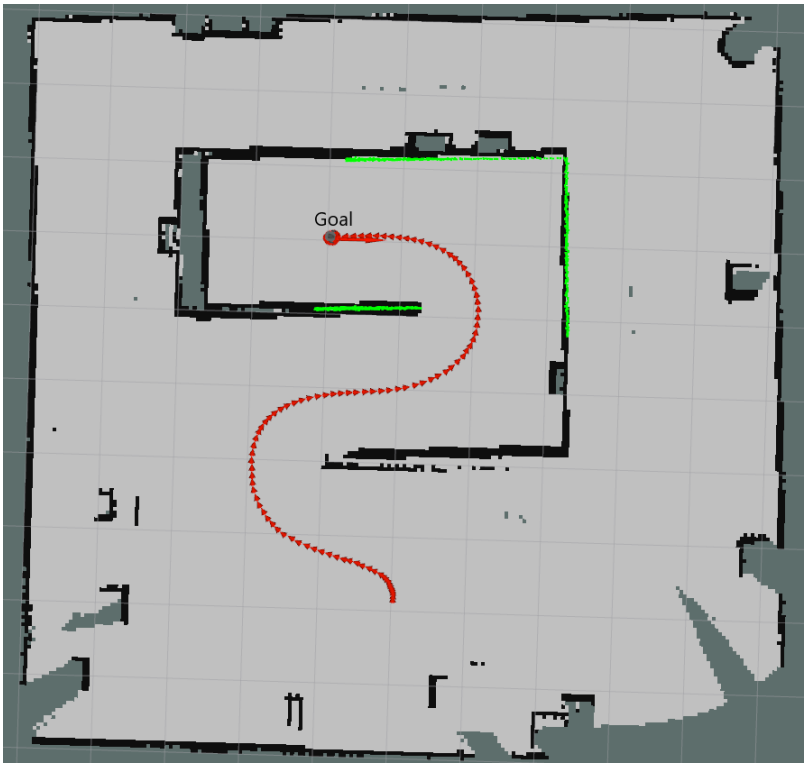


Figure J.4: Path of the robot using move_base from the ROS navigation stack

K Tutorial to Configuring the ROS Framework for the 2D Robotic Solution

K.1 Configuring RPi3 and Remote Workstation

1. The RPi computer is set-up to use an OS image from Ubiquity Robotics [24].
2. The remote workstation is a mobile PC running Ubuntu 18.04 and ROS Melodic installed according to the ROS documentation [25].
3. Appended the next command to the end of `/.bashrc`:

```
export ROS_MASTER_URI=http://pi3:11311
```

K.2 Configuring the ROS Joystick Package

A joystick or keyboard connection to the simulated robot is necessary for remotely operating the robot. A PS3 joystick is connected via Bluetooth. Start by installing the required `joy` package:

```
$ sudo apt-get install ros-melodic-joy
```

Test the joystick is working by using

```
$ sudo jstest /dev/input/js0
```

Note that some systems may allocate a different number to the joystick other than 0. Device name can be found by using:

```
$ ls /dev/input/
```

Set the required permissions for ROS to access the joystick node:

```
$ sudo chmod a+rw /dev/input/js0
```

If the joystick is assigned a different name than `js0` Test if the joystick node functions properly by running:

```
$ roscore
# If the joystick port is different than js0:
rosparam set joy_node/dev "/dev/input/jsX"
# Where X is the number assigned to the joystick
```

and in a two different terminals

```
$ rosrun joy joy_node
$ rostopic echo joy
```

In this terminal pressing joystick buttons should change the displayed values.
The next step is to create a package that would get the output from the joystick node and transform it into `geometry_msgs/Twist` messages that would allow the movement of the robot:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg joy_teleop roscpp joy
$ cd .. && catkin_make
```

Next create `joy_teleop/src/robot_teleop_joy.cpp` with the following contents:

```
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
#include <sensor_msgs/Joy.h>

class TeleopTurtle
{
public:
  TeleopTurtle();

private:
  void joyCallback(const sensor_msgs::Joy::ConstPtr& joy);

  ros::NodeHandle nh_;

  int linear_, angular_;
  double l_scale_, a_scale_;
  ros::Publisher vel_pub_;
  ros::Subscriber joy_sub_;

};

TeleopTurtle::TeleopTurtle():
  linear_(1),
  angular_(2)
{

  nh_.param("axis_linear", linear_, linear_);
  nh_.param("axis_angular", angular_, angular_);
  nh_.param("scale_angular", a_scale_, a_scale_);
  nh_.param("scale_linear", l_scale_, l_scale_);


  vel_pub_ = nh_.advertise<geometry_msgs::Twist>("robot1/cmd_vel", 1);


  joy_sub_ = nh_.subscribe<sensor_msgs::Joy>("joy", 10,
    &TeleopTurtle::joyCallback, this);

}
```

```
39
40 void TeleopTurtle::joyCallback(const sensor_msgs::Joy::ConstPtr& joy)
41 {
42   geometry_msgs::Twist twist;
43   twist.angular.z = a_scale_*joy->axes[angular_];
44   twist.linear.x = l_scale_*joy->axes[linear_];
45   vel_pub_.publish(twist);
46 }
47
48
49 int main(int argc, char** argv)
50 {
51   ros::init(argc, argv, "teleop_turtle");
52   TeleopTurtle teleop_turtle;
53
54   ros::spin();
55 }
```

Listing K.1: Implementation of the `robot_teleop_joy` ROS node which takes input from a connected joystick and transforms it into velocity commands for the robot

To ensure that the package can be built the following lines are added at the end of `joy_teleop/src/CMakeLists.txt` :

```
add_executable(robot_teleop_joy src/robot_teleop_joy.cpp)
target_link_libraries(robot_teleop_joy ${catkin_LIBRARIES})
```

Last step is to write a launch file to start all the required nodes.
Starting in the `joy_teleop/` directory create a `launch` folder and and create the launch file
`joy_teleop/launch/joystick.launch` containing the following:

```
1  <launch>
2   <!-- joy node -->
3    <node respawn="true" pkg="joy"
4          type="joy_node" name="turtle_joy" >
5      <param name="dev" type="string" value="/dev/input/js0" />
6      <param name="deadzone" value="0.12" />
7    </node>
8
9   <!-- Axes -->
10   <param name="axis_linear" value="1" type="int"/>
11   <param name="axis_angular" value="0" type="int"/>
12   <param name="scale_linear" value="2" type="double"/>
13   <param name="scale_angular" value="2" type="double"/>
14   <remap from="robot1/cmd_vel" to="/cmd_vel"/>
15
16   <node pkg="joy_teleop" type="robot_teleop_joy" name="teleop"/>
17 </launch>
```

Listing K.2: Content of the joystick.launch file that helps to execute all the relevant nodes instead of running them separately one by one

Where at line 14 the default topic of the `joy_teleop` node is changed to `/cmd_vel` which is used in the next section.
The package must be built again for the changes to take effect:

```
$ cd ~/catkin_ws && catkin_make
```

```
$ roslaunch joy_teleop joystick.launch
```

and in a new terminal

```
$ rostopic echo cmd_vel
```

At this screen by e.g. moving the left analog stick of the joystick an output of the following form can be seen:

```
---
linear:
  x: 0.00845917593688
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: -0.0
---
```

In this case the left analog stick was moved straight and the `joy_teleop` node created `Twist` message that commands the robot perform a linear movement along the x axis, which shows that the node is working correctly.

K.3 Configuring the Simulation Environment for the Robot

This section aims to be a "quick-start" guide, explaining the necessary steps to install and launch a ROS simulation environment using Gazebo. In the simulation the robot can move either by velocity commands using the keyboard or by using a script to move the robot to desired coordinates autonomously.

All the following steps assumed to be made on a computer or virtual machine running **Ubuntu 16.04.**.

- In a terminal launch the provided **install_ros.sh** script. It should install ROS Kinetic and all the required packages.
- Unzip the provided **got_slam.zip** file into `/catkin_ws/src`

- In a terminal run:

```
$ ~/catkin_ws/src && catkin_make
```

- Once the operation completes, in the same terminal run:

```
$ roslaunch got_node slam.launch
```

Do not close the terminal window as it is required to keep the programs running.At this point two new program windows should open, namely RViz and Gazebo. RViz is a visualization tool that only visualizes the output from all sensors. Gazebo is the program responsible for simulating the robot and the environment as seen in Figure K.1.

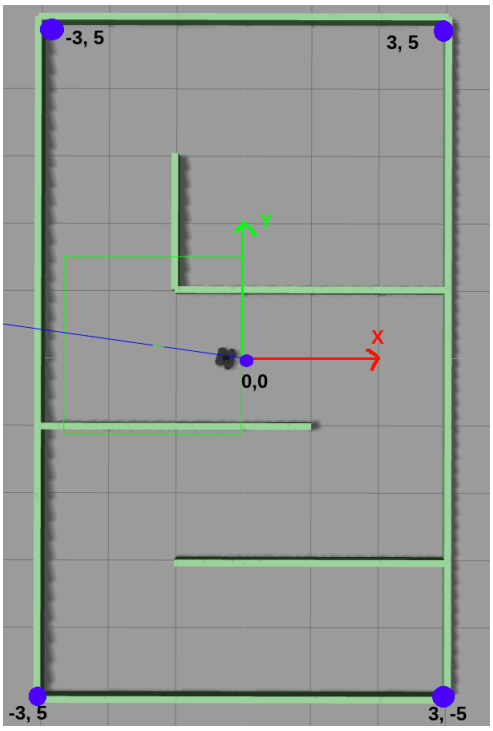


Figure K.1: Simulation of the robot and environment in Gazebo. The blue dots show the respective x, y coordinates

.

Note for easier navigation in Gazebo check the following link, especially the *Mouse Control* section:
`http://gazebosim.org/tutorials?tut=guided_b2&cat=`
To control the robot manually, in a new terminal run:

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

While the focus is maintained on the terminal, by pressing the corresponding keys the robot will move, namely:

- **w**: forward
- **s**: backward
- **a**: left
- **d**: right
- **x**: stop

This way the robot moves by receiving velocity commands, to increase the speed **press the desired command key multiple times**.

To move the robot autonomously, close the terminal window for keyboard control and in another terminal run:

```
$ rosrun got_node go_to_point.py
```

As a result the robot will navigate through several points on the map. More exactly, if looking at the contents of
`~/catkin_ws/src/got_slam/ros/got_node/scripts/go_to_point.py`

```
30 # (x,y) position goal. Robot start position is (2.0, -4.0)
31 goal_points_ = [
32     (-2.0, -4.0),
33     (-2.0, -2.0),
34     (2.0, -2.0),
35     (2.0, 0.0),
36     (-2.0, 0.0),
37     (-2.0, 4.0),
38     (2.0, 4.0),
39     (0.0, 2.0)
40 ]
41
42 rospy.set_param('des_pos_x', goal_points_[0][0])
43 rospy.set_param('des_pos_y', goal_points_[0][1])
```

The script will load up the first set of goal coordinates and move the robot in position. Once at the goal destination the next point is loaded and the robot moves again until it reaches the last point.

The functionality of having a list of x, y coordinates is for demonstration purposes only and not a strict requirement. The movement of the robot is dependent on the ros parameters "*des_pos_x*" and "*des_pos_y*". By example, if the robot is left to run through all of the goal points it will stop, but in this state and even while navigating to another point it is possible to change the goal point just by running the two following commands in a new terminal:

```
$ rosparam set des_pos_x 2.0
$ rosparam set des_pos_y 4.0
```

By doing this we have changed the goal of the robot to the point (2.0, 4.0) and it will start navigating towards it.

It is worth noting that this simulation **doesn't include obstacle avoidance**.

While the robot is moving, it is possible to see in the RViz window that a map is being built. The map is assembled by using the **hector_slam** method, using the Lidar output and the odometry data transmitted by the robot. If the default implementation of `go_to_point.py` is left to complete, the final map will look as in Figure K.2.

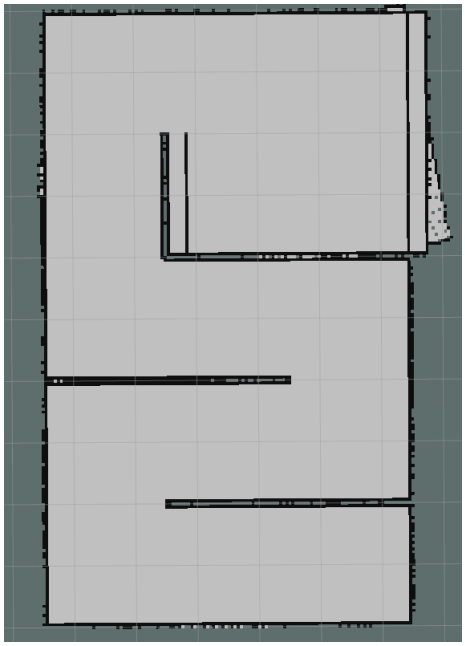


Figure K.2: Visualization of the map built using Lidar output and odometry by using hector_slam SLAM implementation

.

At any point in time, while the Gazebo simulation is running, it's possible to save the built map to a file. To do so run the following in a new terminal:

```
$ rosrun map_server map_saver -f ~/map
```

This will create two-files in the user's home directory, namely: `map.pgm` and `map.yaml`. The *.yaml* file contains some parameters of the map relevant for RViz, but map.pgm is a binary file which can be seen using and image viewer.

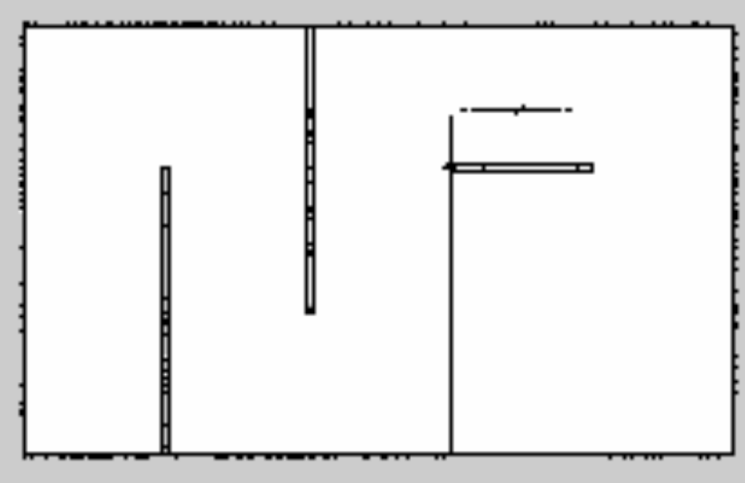


Figure K.3: Saved Binary map file, as seen in an image viewer

.

.
The binary map should closely represent the Gazebo environment, and if the values are adjusted, x, y coordinates from it can be used to give new coordinates to the robot.

K.4 Spawning Simulated Robots into Gazebo Worlds

For simulating the robot this project uses the available packages from turtlebot3[46].

```
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin_make
$ echo "export TURTLEBOT3_MODEL=burger" >> ~/.bashrc
$ source ~/.bashrc
```

It is now possible to launch the simulation environment by running:

```
roslaunch turtlebot3_fake turtlebot3_fake.launch
```

and in a new terminal launch the `joy_teleop` node:

```
$ roslaunch joy_teleop joystick.launch
```

This will result in launching a new `RViz` window displaying the simulated robot which moves as commanded by the joystick as can be seen in Figure 6.2

Simulation using Gazebo

To simulate the robot environment with obstacles the Gazebo software is used. For this the following command is invoked:

```
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

This executes the Gazebo environment as seen in Figure K.4.
given that the joystick has been set in Section K.2 it is possible to launch the joystick node and observe the robot move in in the Gazebo environment. At this point `rviz` is launched to observe the output of the Lidar:

```
$ roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
```

At this point it is possible to initiate SLAM by issuing the following command in a new terminal:

```
roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

At this moment the robot should be fully capable of identifying its position on the map and navigate it autonomously.

K.5 Configuring Gazebo with the GoT Node

The simulation environment is tested under Ubuntu 16.04 with ROS Kinetic installed. Installation of ROS and all other required files can be done by running the following terminal command:

```
$ wget -O - \
https://raw.githubusercontent.com/iarobotics/got_slam/master/install_ros.sh \
| bash
```

The script also creates a new directory `~/catkin_ws/src/got_node` where the source code is stored.
The first step is to execute a demonstration scenario using the following commands in two separate terminals:

```
$ roslaunch got_node demo.launch
```

Select **Yes** for any appearing prompts. As a result several windows appear on the screen, namely:

- Gazebo - the application simulating the robot and its environment, see Fig K.4
- RViz - a vizualization tool. Displays the output from the robot's sensors. see Fig K.5
- PlotJuggler - a plotting tool, see Fig K.6

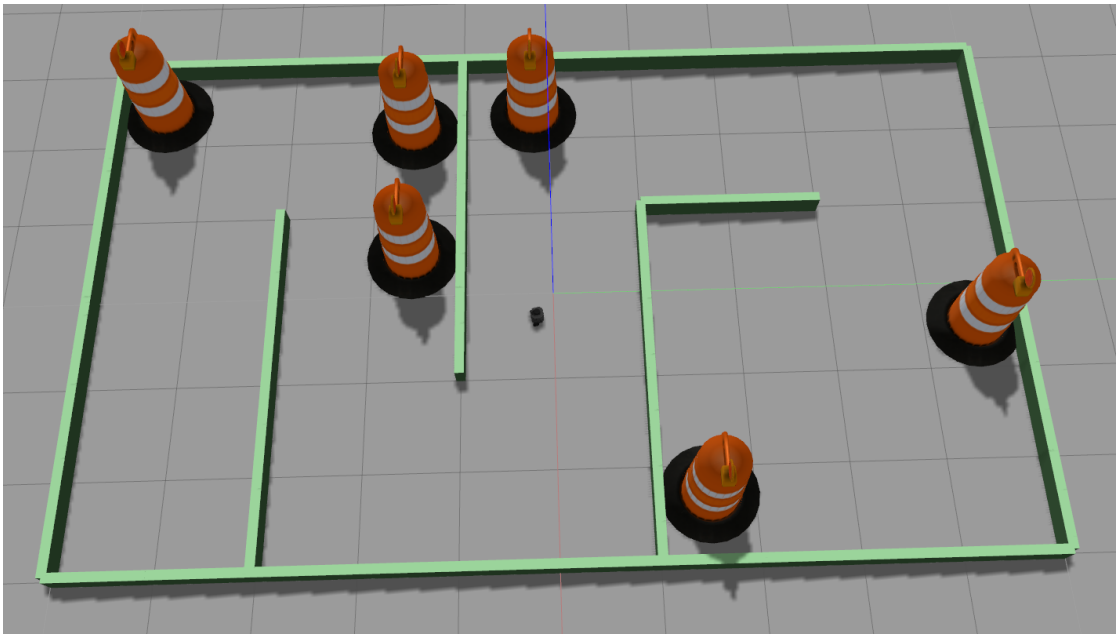


Figure K.4: Gazebo environment with the origin of the robot at 0,0

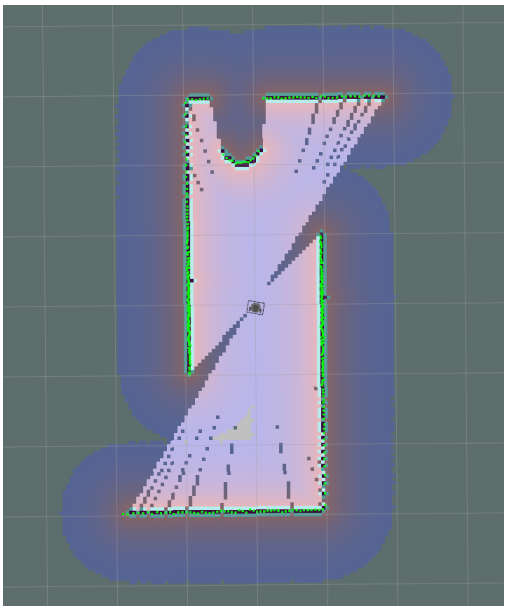


Figure K.5: RViz window showing the current state of the map, the position of the robot on the map and current Lidar reading as green points

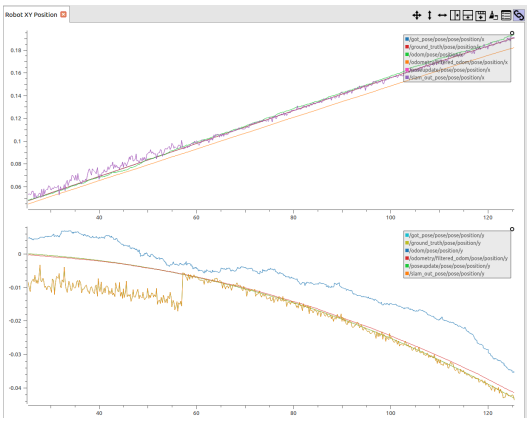


Figure K.6: PlotJuggler plot, in this instance showing the XY position of the robot