UNIVERSITY OF CAMBRIDGE
Department of Engineering

Robotic Exploration of an Indoor Environment for Data Collection

Author Name: **Luis Bustillo Ortiz**
Supervisor: **Dr. Michael Crisp**
Date: **1/06/22**

I hereby declare that, except where specifically indicated, the work submitted herein is my own original work.

Signed _____ *date* **1/06/22**

Robotic Exploration of an Indoor Environment for Data Collection

B-TAC1000-3

Luis Bustillo Ortiz

Supervisor: Dr. Michael Crisp

Department of Engineering
University of Cambridge

This project is submitted for the degree of
Master of Engineering

Churchill College June 2022

Acknowledgements

I would like to acknowledge and express my gratitude towards my supervisor, Dr. Michael Crisp, who made this work possible. His guidance and advice carried me through all the stages of this project. I also wish to show my appreciation to Dr. Shuai Yang for his brilliant insights and suggestions, as well as his help setting-up and carrying out experimental activities.

I would also like to extend my special thanks to my brother, for his continuous support and understanding when undertaking my research and writing my project.

Material from IIB Module 4M20

One of the map files used in simulation testing (Section 3.3) was provided by the module leader for use in the coursework assignments of module 4M20 *Introduction to Robotics*, and has been borrowed here, but there is no work submitted for that module or the project which overlaps. On account of this, I would like to thank Dr. Amanda Prorok and the rest of the module organizers.

Technical Abstract

The goal of this project was to design, program and evaluate a robotic platform which can autonomously explore a known, static environment, avoiding obstacles, while collecting and later processing information about its surroundings through onboard sensing. As the project was conducted under the supervision of the RFID research group, a Software-Defined Radio (SDR) antenna was chosen as the sensor attached to the robot, and radio-frequency (RF) signal strength as the physical phenomenon to be measured. The task, was to develop a platform capable of roaming around an environment following a path computed to maximize coverage, while sampling the RF signal power at multiple points, and storing the values for further analysis.

Current methods to obtain these measurements are time-consuming and involve expensive equipment that is often large & difficult to manoeuvre. Hence, this project sought to automate channel measurements using low-cost robotics. The primary applications of this technology lie in the RFID market, by detecting and mapping the presence of RFID tags for warehousing logistics, inventory management or retail purposes for example. A more general use of the robotic platform, for data collection other than RF signal power might include automation of coverage dependent tasks (e.g cleaning robots), or monitoring the physical properties of a space (e.g factory, power plant etc) to ensure safety or optimize performance.

The structure of the report reflects the order in which the project was completed. First, a review of similar robotic deployments and relevant academic literature is presented. After, the proposed choices for hardware, software and testing methods for navigation and RF measurement to fulfil the specification are detailed. Then, the major project implementations are explained, followed by a discussion on subsequent testing of these subsystems, and an analysis of the results obtained. Finally, the main conclusions for the project are drawn out, and an outlook on future research and project extensions is given.

The system designed consists of a TurtleBot3 Burger robot, equipped with an RTL-SDR receiver antenna, which communicates via the Robot Operating System (ROS) framework with a virtual machine running Ubuntu 20.04, installed on a Host PC. Two different SDR antenna arrangements were tested on the robot, one with a small antenna tucked to the side of the robot, and another with a large antenna mounted on its own layer on top of the robot, partially blocking the LiDAR sensor. The main implementations are the path planning, navigation and RF collection & processing programs, written in Python 3.

The path planning algorithm takes in an input PGM file of the testing environment, and computes an optimal path that maximizes area coverage, minimizes backtracking and avoids obstacles, at a spacing between samples set by the user. The sampling points in the path are fed to the navigation algorithm which estimates the pose of the robot in the environment, using LiDAR sensor readings (a process referred to as Localization), and the robot controller issues appropriate velocity commands to guide the robot from point to point. At each sampling point along the path, the Power Spectral Density (PSD) of the RF field is estimated, from SDR measurements, and used to assess the field strength at the chosen frequency, 915 MHz. Meanwhile, the power values registered are stored alongside their respective locations. The output data files produced by the logging systems are processed to visualize the trajectory followed by the robot, and produce a contour map of the estimated RF field strength.

The development cycle followed throughout the project involved testing the robot controller and path tracking algorithms in a simulated environment prior to testing in the real world, a common practice in industry. For real-world testing, a 1.5 m × 1.5 m arena was built using cardboard boxes, and was tested both with and without obstacles. In addition, to evaluate the accuracy of the RF measurement system, an ADLAM-PLUTO SDR was used as a transmitter to create a field pattern governed by a signal of known, constant properties, greatly simplifying testing.

The main conclusions of the project are summarized below:

- Testing in simulation is a good way to assess whether the controller and navigation algorithms are effective, but in no way guarantees the same level of performance in the real world.
- The path planning algorithm responded well to all the tests, managing to work for different quality map files, map sizes and types (obstacles vs empty), and sample spacing requirements.
- The navigation algorithm, although simplistic, is quite robust, leading the robot safely along the pre-computed route, failing only in very few situations.
- Radiation patterns for antenna in the side position, show it is quite poor even when close to the source. The larger antenna mounted in the top layer is almost omnidirectional when close to the source, but as it moves further away it becomes considerably worse.
- The RF contour and signal decay vs distance plots for the data collected are very different from the theoretical model predictions. There is good agreement close to the transmitter, but the measurements are very noisy, and the contour shows an irregular pattern with a very subtle trend. There are a lot of improvements to be made to the data collection and processing systems, to reach an acceptable industry standard.
- While the measurements recorded have improved significantly with the antenna on top, the navigation is considerably worse due to the LiDAR being blocked. This poses a trade-off between the navigation and RF measurement subsystems for the project in its current state.

Link to project repository on GitHub: `https://github.com/LuisBustillo/IIB_Project`

Contents

Nomenclature

Acronyms / Abbreviations

API	Application Programming Interface
CPP	Coverage Path Planning
DFS	Depth-First Search
DOE	Defined Operational Environment
EM	Electro-Magnetic
FSPL	Free-Space Path Loss model
IMU	Inertial Measurement Unit
LDS	Laser Distance Sensor
LiDAR	Light Detection and Ranging
MCL	Monte Carlo Localization
OG	Occupancy Grid
PID	Proportional, Integral and Derivative
PRM	Probabilistic Road-map
PSD	Power Spectral Density
RFID	Radio Frequency Identification
RF	Radio Frequency
ROS	Robot Operating System
RRT	Rapidly-exploring Random Trees
SBC	Single Board Computer
SDR	Software-Defined Radio
SLAM	Simultaneous Localization and Mapping
STC	Spanning-Tree Coverage algorithm
TOF	Time-Of-Flight
TSP	Travelling Salesman Problem
VM	Virtual Machine

Introduction

1.1 Motivation

The goal of the project was to design, program and evaluate a robotic platform which can autonomously explore a known, static environment [1], avoiding obstacles, while collecting and later processing information about its surroundings, through onboard sensing. The final output shows how the physical phenomenon observed varies across the environment in the form of a contour map.

While the exploration component of the project can be applied to measuring any kind of signal (temperature, humidity, radiation etc) provided the robot is equipped with the right sensing equipment, we have chosen to limit ourselves to collecting radio-frequency (RF) signal strength, as an example of what is achievable using the robotic platform. In addition, this allowed leveraging of Dr. Crisp and his research group's expertise in guiding the outcomes and experiments to test and evaluate the performance of the system, by comparing results to theoretical and empirical models.

A successful project would develop a system capable of the following. Using the provided map, compute a pre-defined path that minimizes back-tracking to cover the whole environment. The robot roams around following the path while sampling the RF electro-magnetic (EM) field across 3 independent axes at multiple locations along the route. After the measurements have been collected, they are calibrated and the resulting data is interpolated to obtain a close estimate of the RF strength contour at the specified frequency. Current methods to obtain these measurements are time-consuming and involve expensive equipment that is often large & difficult to manoeuvre. Hence, this project seeks to automate channel measurements using low-cost robotics.

[1] A known environment is one for which a map exists. The map includes interesting features of the environment (landmarks). It can be a video, 3D scans or even Wi-Fi signal strength [1].

Specifically focusing on RF field measurements, the primary applications of this project lie in the RFID market, currently valued at US$ 12.2 billion and set to grow to US$ 16.23 billion by 2029 [2]. Examples of uses in this space are:

- Warehousing logistics/Inventory management
- Retail

Other applications of the developed technology, in a general scope, might include:

- Automation of coverage dependent tasks (e.g cleaning robots, automated paint spraying robots, window cleaning robots, forest monitoring robots, agricultural robots etc)
- Monitoring the physical properties of a space (e.g factory, power plant etc) to ensure safety or optimize performance
- Aid in search and rescue scenarios by finding clues in the measurements
- Remote exploration and monitoring of a mapped environment in the context of planetary reconnaissance missions (e.g Mars Rover)

1.2 Project Specification

As mentioned previously, this project was undertaken in a research group whose primary focus is development of RFID, thus many of the decisions made focus on applications in diagnosing & characterizing deployments of RFID infrastructure.

In order to successfully read passive RFID tags, the incident RF power must surpass a threshold, thus in wide area deployment, any "dead spots" of low RF power must be identified and rectified for reliable operation. ETSI defines the RFID frequency bands to be 865 – 868 MHz and 915 – 921 MHz [3] corresponding to a wavelength of $\approx$ 30 cm, thus sampling the power every 10 cm (in the worst case) would allow identification of such dead spots. Typical deployments of RFID cover areas of approximately 10 m $\times$ 10 m, however if the system could be shown to be accurate over a 2 m $\times$ 2 m range it would serve as a proof of concept. This range is outside the reach of robotic arms, and the desired accuracy is too high for GPS-positioned or indoor drone usage, thus a wheel-based solution is the most sensible. Moreover, the solution should strive for minimal user intervention during setup & running, in order to provide clear benefits over costly & time-consuming manual approaches. The threshold power for a tag to turn on varies due to circuit design, but is usually in the range of $\approx$ -20 dBm for a basic tag, and up to $\approx$ -10 dBm for higher power applications such as wireless sensing. Combining all of the above desired properties yields the specification in Table 1.1, which is a slightly modified version of the specification for David Swarbrick's project [4], reflecting the differences in approach and outcomes. More on this in Section 2.

Table 1.1 Specification of quantitative requirements for measurements associated with robotic system to be developed

Measurement	Property	Target	Acceptable range
Robot position	Accuracy	$\sim$ cm	$\pm$ 5cm
Robot position	Range	10m x 10m	2m x 2m
RF signal	Bandwidth	0-60 GHz	865-921 MHz
RF signal	Power	$\sim$10dBm – -70dBm	0dBm – -30dBm
RF signal	Power Accuracy	$\sim$0.1dBm	<1dBm

1.3 Project and Report Structure

In order to successfully satisfy the aim, the project was broken down into several milestones:

1. Undertake a literature review, to learn about existing solutions in industry and academia.
2. Carry out a feasibility study of mapping, localization and navigation methods/algorithms to assess which aspects of the project to focus development on.
3. Design, implementation and testing of path planning and navigation algorithms.
4. Integration of Software-Defined Radio (SDR) antenna for measurement of RF signal strength, and development of collection and processing programs for samples.
5. Testing and evaluation of final system.

The structure of this report follows a similar progression to the above objectives. First, a review of similar robotic deployments and relevant academic literature is presented. In Section 3, System Design, the proposed choices for hardware, software and methods for navigation and RF measurement to fulfil the specification are detailed. Then, in Section 4, the major project implementations are explained followed by a discussion on subsequent testing of these subsystems, and an analysis of the results obtained. After, an error evaluation for the systems developed is carried out, in Section 5. Finally, in Section 6, the main conclusions for the project are drawn out, and an outlook on future research and project extensions is given.

Literature Review

The first step of the project was to look at previous research efforts in the fields of autonomous coverage, path planning, navigation and RF field measurement, from both academia and industry. This enabled a feasibility study to be carried out, which determined whether the goals set out for the project were too ambitious, or not challenging enough. This ultimately led to the specification in Section 1.2.

RF Field Measurements with Mobile Robots

The following papers describe the use of mobile robots to measure an aspect of the RF field in an indoor environment employing different forms of localization mechanisms through the use of different sensors (odometry, cameras, IMUs, LiDAR). This enabled a comparison of several pose estimation methods that informed the project proposition. Many of the modern perception, mapping and localization techniques stem from Thrun's *Probabilistic Robotics* [5], where the foundation for mobile robotics algorithms is contained.

A Pioneer 2 robot was equipped with a laser range-finder (LDS) to create a map of an indoor space using a variant of SLAM (FastSLAM) by learning the geometrical structure of the environment through TOF measurements [6]. An RFID antenna attached to the robot detected the presence of RFID tags in the area, and given the map and the maximum likelihood path of the robot computed by the FastSLAM algorithm, the locations of the RFID tags were estimated through recursive Bayesian filtering in a technique known as Monte Carlo Localization (MCL) [7]. Another example showed how LiDAR and indoor GPS were combined (a process known as sensor fusion) to increase accuracy of robot localization in a simulated environment [8].

A novel wireless power transfer system, "Energy-Ball" [9], was tested using a small robot (Pioneer-P3DX) mounted with measuring apparatus and manually driven around a 20×20 m^2 test area. At each test position, the robot's location was determined using an upwards facing camera scanning a Fiducial marker (Chilitags [10]), and the power measured on a 915 MHz channel. To adequately localize the robot, 460 Chilitags were placed on the ceiling with 1 metre spacing.

Two Pioneer P3-AT mobile robots were used to characterize wireless channels for networked robotic systems [11]. Their payloads were IEEE 802.11 g (WLAN) antennas, and they were localized using their gyroscopes and wheel encoders, achieving an error of 2.5 cm every 1 m of straight line movement.

A small unmanned aerial vehicle ("drone") has been deployed as a spectrum monitoring tool [12]. The drone carried a Raspberry Pi attached to a RTL2832U-based SDR which is able to measure frequencies between 25–1700 MHz, and streamed relevant data over a wireless network connection to a master computer. Its location and orientation were estimated using indoor GPS and an electronic compass, and the SDR computer package employed was GNU Radio.

Planning and Navigation Algorithms

The next set of papers detail path-planning and navigation algorithms used in discrete (graph traversal) and continuous environments. LaValle's *Planning Algorithms* [13] goes into depth on the elementary theory and fundamentals of path planning, covering the essential concepts of the combinatorial and sampling-based approaches discussed next. In particular, Rapidly Exploring Random Trees (RRTs) and Probabilistic Road-maps (PRMs), which are two algorithms used for point-to-point graph navigation, designed to minimize distance travelled.

The analysis in [14] hinges on novel connections between stochastic sampling-based path planning algorithms and the theory of random geometric graphs, to explore the computational complexity and asymptotic behaviour of traditional RRTs and PRMs. Sampling-based motion planning with automatically derived extension heuristics is also explored in [15, 16], resulting in optimized versions of the RRT and PRM algorithms respectively, which boast lower run-times, allowing them to be run continuously during robot operation for the best performance (termed "online" operation).

The achievements of the last two decades on area coverage are reviewed in [17]. An analysis and summary of the best Coverage Path-Planning (CPP) algorithms is given, highlighting the advantages and disadvantages. This informed the options available for solving the CPP problem, precisely the smooth-STC model [18]. It is a type of Spanning-Tree Coverage traversal method which can identify an optimal path that, avoids all obstacles, prevents (or at least minimizes) backtracking, and maximizes the coverage in any defined operational environment (DOE). This algorithm was shown to be optimal if run offline on a readily available map.

A solution for path-finding and navigation of indoor robot cleaners is proposed in [19]. The proposed program plans a path from a priori cellular decomposition of the work environment into an occupancy grid. The planned path achieves complete coverage on the map and reduces duplicate coverage. The solution is implemented inside the ROS framework [20], and is validated within Gazebo simulator [21]. This paper was especially helpful as it exemplified how to structure a robotics project using ROS, and many of the concepts and objectives were directly transferrable to this project.

Software Platforms

As the project is mainly computational in nature, a review of the main software tools and development pipelines used in robotics research in industry and academia, was in order. Completion of the AWS Robomaker badge series [22] provided a good foundation and understanding of ROS [23], and how to use it to build robotics projects effectively. ROS is an open-source framework that consists of software libraries and tools that allow users to build general, multi-domain robot applications effectively and at scale. It is well documented and has an active global community For more details on how ROS works, and how it was used in this project refer to Appendix B. The series also explored the popular robotics simulation tool, Gazebo [21], which a realistic simulator with advanced 3D graphics, precise physics and detailed sensor and noise models, that can easily integrate with ROS. These two platforms were at the centre of developing the systems implemented in this project.

Previous IIB Project

David Swarbrick carried out a IIB project with an overlapping set of goals, under the supervision of the same research group, in 2020 [4]. The foundation for the research ideas was laid out, but a fundamentally different approach was employed, using different sensors and software development applications and pipelines, focusing primarily on RF field measurements and characterization, using a teleoperated robotic platform. This is as opposed to programming a robot to autonomously cover an area optimally by accurately localizing through LiDAR readings and not odometry, to serve as a versatile data collection tool, which are the outcomes of this project.

System Design

3.1 Overview

The diagram in Figure 3.1 shows the final design of the proposed system. A user provides a map of the environment they wish to collect data over, in the form of a PGM file (in testing, the map file was generated using SLAM, see Section 3.3). The path planning algorithm processes the map, and computes an optimal path that maximizes area coverage, avoiding obstacles, at a spacing between samples set by the user. The sampling points in the path are fed to the navigation algorithm which estimates the pose (position and orientation) of the robot in the environment, by comparing the LiDAR sensor readings and the map file, and the robot controller issues appropriate velocity commands to guide the robot from point to point. At each sampling point along the path, the Power Spectral Density (PSD) of the RF field is estimated, from measurements performed by the SDR attached to the robot, and used to assess the field strength for the desired frequency, chosen to be 915 MHz. The power values registered are stored alongside their respective locations in the environment. The output data files produced by the logging systems are processed to visualize the trajectory followed by the robot, and produce a contour map of the estimated RF field strength in the environment.

A decision was made to collect data in specific locations rather than continuously, as the program written to display the results (using python library *matplotlib.pyplot* [24]) has a quadratic time complexity ($O(n^2)$), which means that the run-time scales with the volume of data created squared, and due to the limited computing power available this proved unacceptable. The resolution of the sampling (i.e. spacing between samples) is fed as an input to the path-planning algorithm to retain accuracy in the solution.

The final design presented here was developed over time, from an initial starting point of considering “path planning”, “navigation” and “field measurement” as the major challenges facing a completed system. System development was structured around these issues, with multiple alternatives explored for each, eventually resulting in the final design. The decisions made regarding each subsystem will be detailed in Section 4.

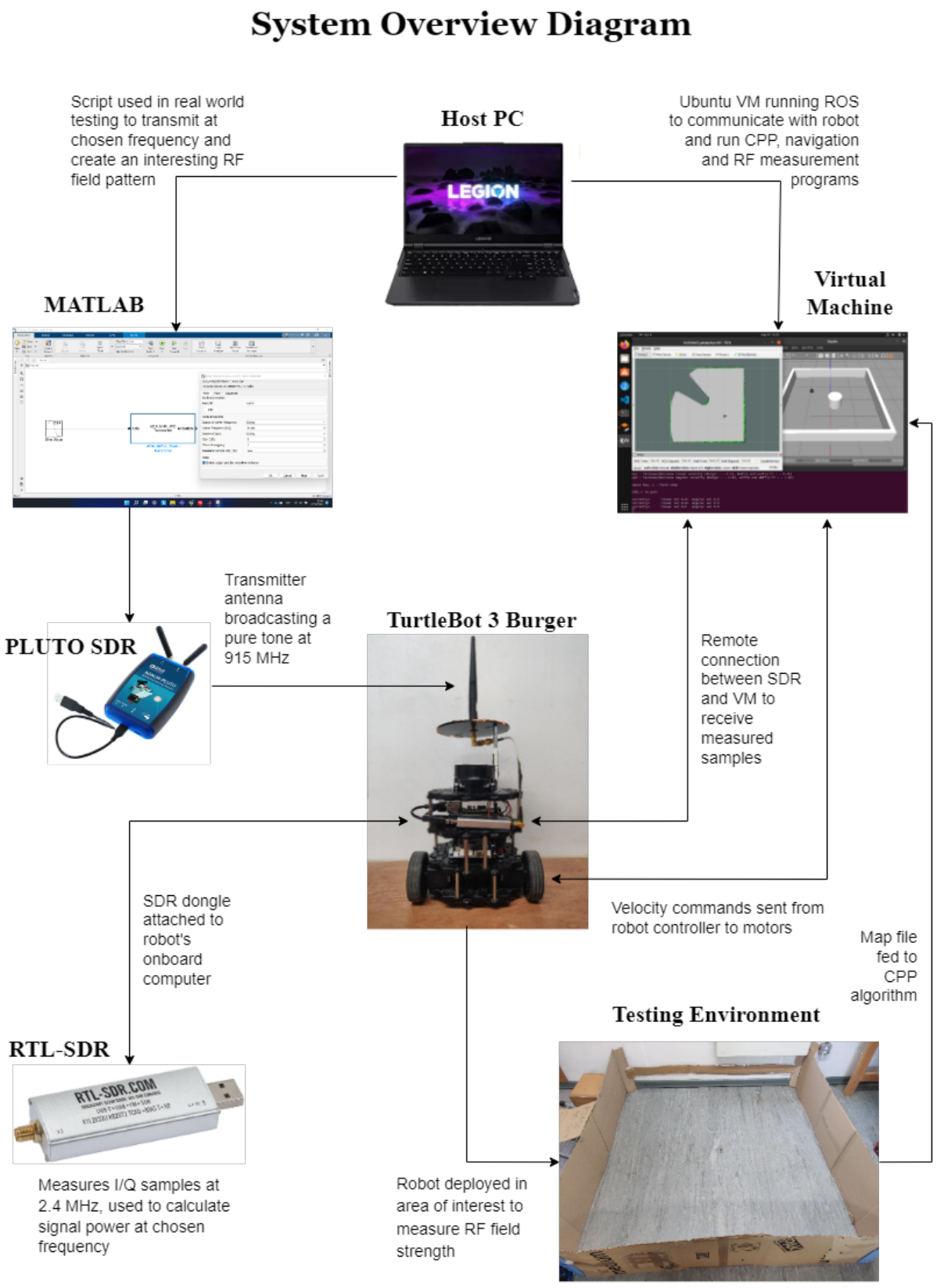


Fig. 3.1 Diagram showing how all the elements in the final design are connected together.

3.2 Robot

This section will discuss the design choices that were made regarding the robot, encompassing hardware and software elements.

3.2.1 Hardware

A TurtleBot3 Burger made by ROBOTIS [25] was chosen as the robotic platform for this project. It is a relatively low-cost, widely available commonly used research platform, with a detailed documentation and setup manual, and extensive open-source community support. It is a differential drive robot, which contains an OpenCR open-source motor control board, interfaced with an on-board Raspberry Pi 3 single board computer (SBC), responsible for running all the programs and processing requests sent by the Host PC. Also available on the TurtleBot3, is a LiDAR rangefinder [26], which allows the usage of Simultaneous Localization and Mapping (SLAM) algorithms as a position measurement option for the robot relative to its environment. Odometry measurements are recorded by an IMU embedded in the OpenCR board. The battery life of the TurtleBot is reported to be $\sim$ 2.5 hrs. A labelled schematic of the TurtleBot is shown in Figure 3.2.

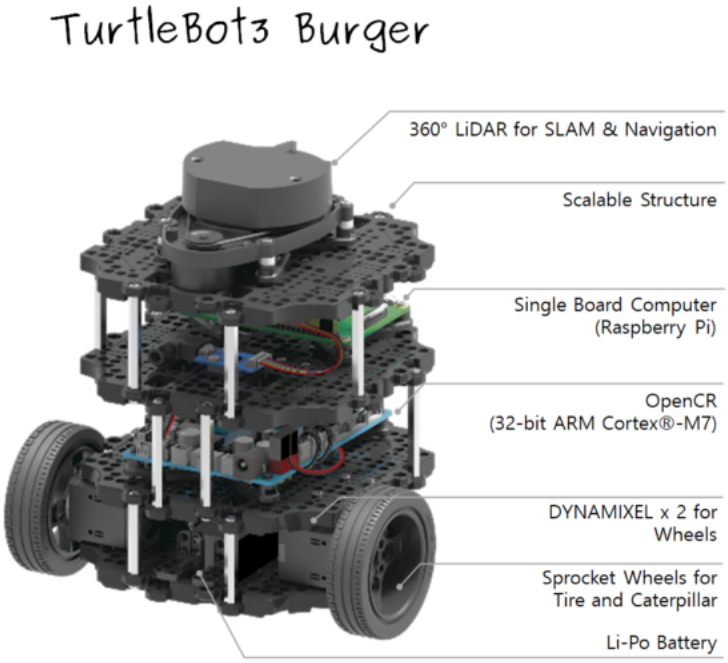


Fig. 3.2 Diagram showing a view of the Turtlebot3 burger with the main components labelled. Courtesy of ROBOTIS [25].

3.2.2 Software

The Host PC consisted of a Virtual Machine (VM) running Ubuntu 20.04, which is the latest version at the time of writing, through Oracle's VirtualBox 6.1.3 installed on my personal laptop (Lenovo Legion 5). A Linux based OS is required to communicate with the TurtleBot via the Robot Operating System (ROS) framework, which is widely used in industry. ROS is a collection of open-source libraries written in C++, Python 2 & 3 and Lisp which provide a common framework for writing software to control robots [20]. For more detail on how ROS was used in this project, see Appendix B. ROS Noetic was installed on the Host PC. The TurtleBot's Raspberry Pi 3 was fitted with Ubuntu 16.04 (along with ROS Kinetic) and the necessary drivers for the OpenCR & LiDAR, by following the start-up guide. ROBOTIS provide packages to control the TurtleBot under ROS, [27, 28] and these were used as a starting point for the project and some of its implementations. A key aspect of the project was to test the navigation algorithms in simulation, and this was achieved using Gazebo simulator v11 [21], which is the default simulation package for ROS. The majority of the code files are written in Python 3, and all the relevant files needed to run the various tests and experiments discussed in this report can be found in the GitHub repository for the project [29]. A picture of the screen setup adopted when running experiments, showing several software programs mentioned, can be seen in Figure 3.3.

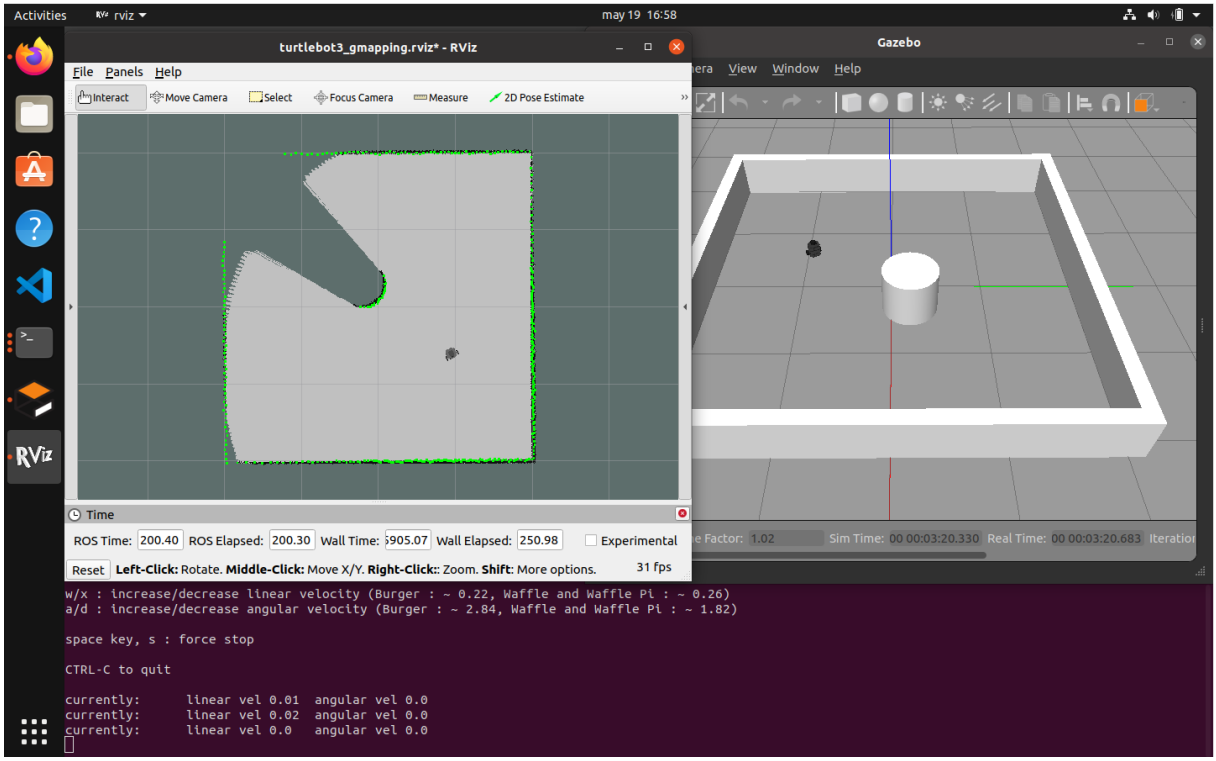


Fig. 3.3 Picture of VM setup when running mapping step. RViz visualizer in the top left, Gazebo simulator in the top right and the command-line at the bottom.

3.3 Testing Methods

The development cycle followed throughout the project involved testing the robot controller and path tracking algorithms in a simulated environment prior to testing in the real world, as setting up the communication between the robot and the Host PC is time-consuming, and not worth the effort unless we can guarantee that the program works as expected in simulation. This methodology is a common practice in industry, where dynamic models of a problem are first built and tested to ensure acceptable results are obtained, before committing to a real-world test, which is often expensive. The specific instructions and commands needed to run the system designed can be found in Appendix C.

Mapping

As mentioned before, the path-planner implemented in this project (4.1) uses a map file of the area of interest to find an optimal path that solves the CPP problem. Mapping could be achieved using different sensing equipment like an overhead camera, 2D or 3D LiDAR and 2D sonar for example. However, the easiest way was to leverage the LiDAR on the TurtleBot to generate a 2D grid map of square discretized cells which represents the topology of the world, making inferences about which cells are occupied. Such a grid is called an Occupancy Grid (OG). The result is a PGM file [1] where each cell in the grid is "colour-coded" by a number representation. The colours of the map correspond to the robot's knowledge of its environment: gray – unexplored/unknown regions, white – clear space the robot may occupy, black – solid barriers/obstacles. An example of such a map is shown in Figure 3.4.

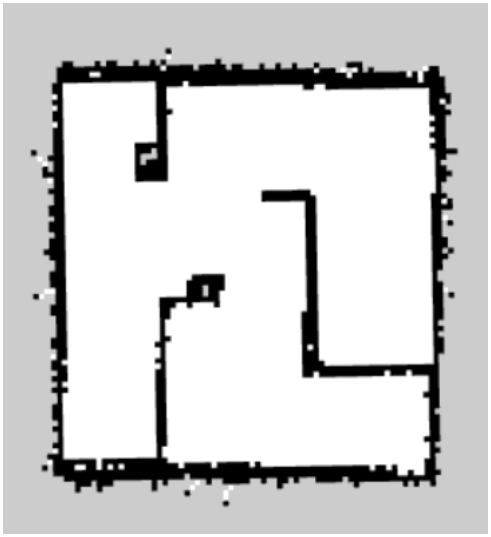


Fig. 3.4 Example of the map file generated by the SLAM algorithm. Courtesy of ROBOTIS [25].

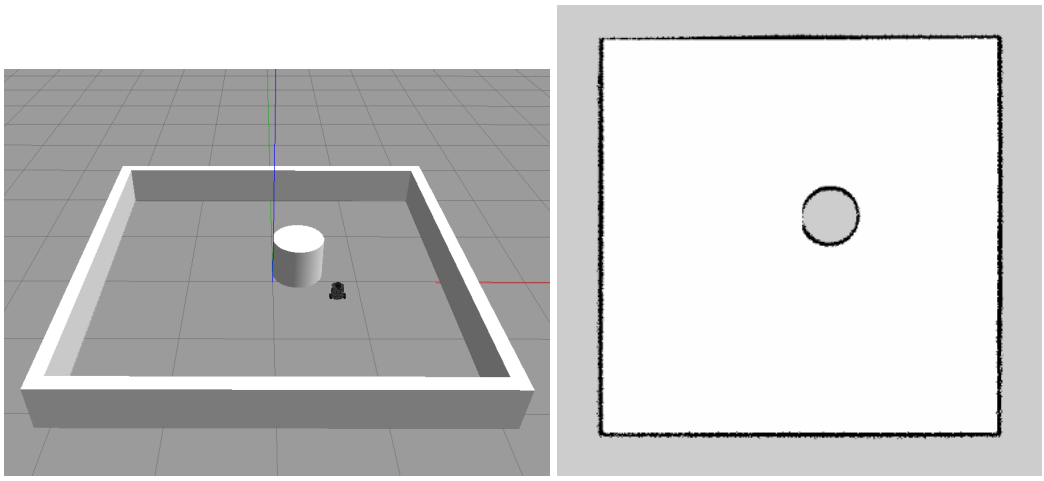
[1] A PGM file format is a grayscale image file saved in the portable gray map (PGM) format and encoded with one or two bytes (8 or 16 bits) per pixel. It contains header information and a grid of numbers representing different shades of gray, from black to white

The SLAM package provided by ROBOTIS [28] was used for this stage, as it has been optimized for the TurtleBot robot. It is based on MCL, and relies on several algorithms well documented in [5]. Essentially, the robot can be driven around manually (or roam autonomously with a simple rule-based controller which avoids collisions) while the LiDAR sends out laser pulses and measures the TOF of the received reflections to calculate the distance to nearby objects, by spinning through 360°, hence mapping the walls and obstacles in the environment. At the same time, a particle filter provides an estimation of the posterior probability function for the pose of the robot, taking into account the kinematic motion model (differential drive in this case) and past motion. These two stages work in unison to build a picture of the environment by spotting external references that tell the robot whether it has returned to a place it's seen before, allowing it to estimate its accrued drift, and achieve so-called loop-closure and thus localization. An image of the SLAM program in action is shown in the top left of Figure 3.3. Developing a mapping approach was not a primary aim of this project, but rather a necessary precursor to the path planning algorithm.

3.3.1 Simulation

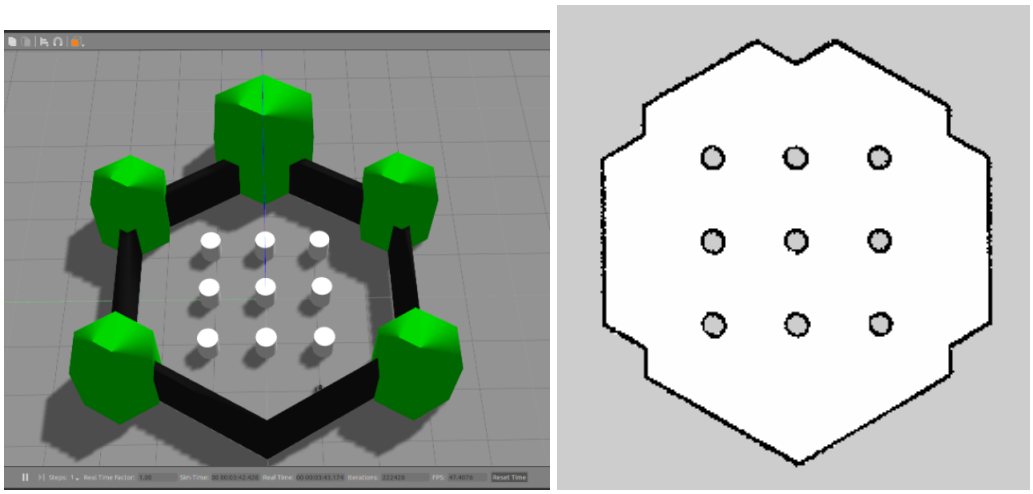
The first step of testing in simulation involves loading the desired testing environment in Gazebo [21], and creating a map file of the world as discussed in the previous section. Then, the sampling spacing is selected and the path-planner program is run to give the CPP trajectory as a list of sampling points for the robot to visit. After this, the main algorithm which guides the robot from point to point, bringing together localization and navigation, is executed. Unfortunately, it is not possible to test the RF measurement and data collection part of the algorithm in simulation, as the transmitter and receiver antennas cannot be modelled in Gazebo.

Two simulated worlds were used to test the robot controller. The first of these was created for use in IIB module 4M20's coursework assignments, and is 4 m × 4 m. The second one was created by ROBOTIS [28] and is approximately 6 m × 6 m. These simulated worlds are shown in Figures 3.5 and 3.6, respectively.



(a) (b)

Fig. 3.5 Render of the first simulated environment (Fig.3.5a) and its corresponding map generated by SLAM (Fig. 3.5b).



(a) (b)

Fig. 3.6 Render of the second simulated environment (Fig.3.6a) and its corresponding map generated by SLAM (Fig. 3.6b).

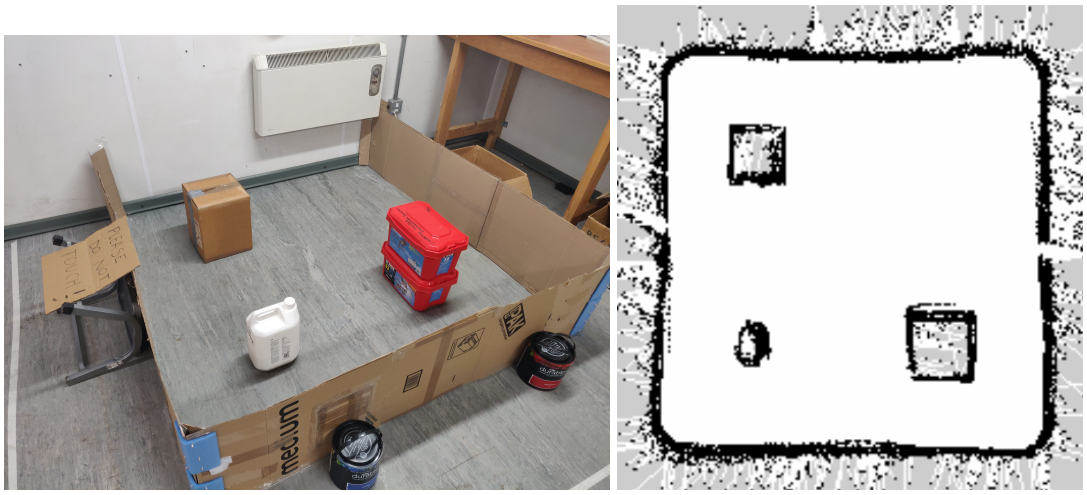
3.3.2 Real-world

Testing in the real-world is relatively more complicated, including changes to the steps above and some additional ones. Firstly, we have to connect to the SBC in the real TurtleBot (via the Secure Shell or SSH), and initiate the bring-up instructions. The mapping and path planning stages can then be launched. RF field measurements can now be taken, so we have to connect to the SDR and turn it on to start taking samples. In the testing setup used, a transmitter (Tx) was employed to generate an interesting RF pattern at the desired frequency, in the area of interest, so the script to start transmitting samples was run at this point. See the next Section (3.4) for more details. Finally, the navigation algorithm is executed, and the data is logged for further analysis.

A simple testing arena in which to conduct the experiments was built out of cardboard boxes, and consisted of 4 solid walls/barriers having dimensions of approximately 1.5 m × 1.5 m. It was tested in 2 different configurations, with and without obstacles, shown in Figures 3.7 and 3.8 respectively.

(a) **(b)**

Fig. 3.7 Picture of the empty testing arena (Fig.3.7a) and its corresponding map generated by SLAM (Fig. 3.7b).



(a) **(b)**

Fig. 3.8 Picture of the testing arena filled with obstacles (Fig.3.8a) and its corresponding map generated by SLAM (Fig. 3.8b).

The poorer resolution in the SLAM generated maps compared to those in the previous section (3.3.1), is partly due to real-world noise and sensor inaccuracies, and partly to two metal stand-offs, 180° apart, which block the LiDAR beam. The stand-offs support the extra layer added to house the SDR antenna.

3.4 RF Field Measurement

This section focuses on the two devices that made RF field strength measurement and subsequent analysis possible, the SDR attached to the robot's SBC, and the transmitter that is used to generate an RF field pattern for the SDR to detect and quantify, in testing.

3.4.1 Software Defined Radio (SDR)

The requirements of the measurement device were that it must be capable of power measurements across the 865–921 MHz frequency bands and can be interfaced to the SBC on the robot. Due to the low cost, compactness, high flexibility and ease of integration with the TurtleBot's Raspberry Pi, an RTL2832U based SDR was selected as the measurement device. In addition, it includes a supporting Python API [30] which stems from the *librtlsdr* library, making it simple to use built-in functions to write high-level programs. The installation steps in [31] were followed, setting up the Host PC, VM and SBC to interface with the SDR. Several tests were run with the dongle directly plugged into the computer, to ensure the device worked properly, before attaching it to the SBC via a USB extender cable. Two different antenna configurations were trialled on the robot (see Section 4.3). An image of the RTL-SDR device is shown in Figure 3.9.

Fig. 3.9 Picture of RTL-SDR dongle used in project. Taken from `https://www.rtl-sdr.com/`.

Firstly, a program was written to read a stream of I/Q samples measured by the SDR, and plot a PSD of the signal from which to estimate the power value at a desired frequency (More detail in Section 4.3). The next step was to find the calibration factor for the SDR, to ensure accurate measurements of the RF power in the area of interest were taken. This was done by connecting the output of a signal generator to the input of the SDR dongle (via a 50 Ω coaxial cable) in the RFID lab, and comparing the received signal power to the known transmitted power, for transmitted powers in the range -20 dB – -80 dB. The results are summarized in Figure 3.10, for 2 values of SDR "software" gain [2]. The graphs show a straight line through the measurements having a slope of 1 in both cases (almost perfect fit from the R^2 values), confirming that both power measurements are in dBW.

[2]This refers to changing gain parameter in the code that configures the SDR as a receiver, not the antenna gain.

Looking at the line for a gain of 10, the zero crossing is at -50dB. This gives us a calibration equation relating the measured power, P_{SDR}^{dB}, to the real received power, P_r^{dB} :

$$P_r^{dB} = P_{SDR}^{dB} - 50 \tag{3.1}$$

Lab testing also showed the SDR is capable of picking out the RF peak at 915 MHz (pure tone) from noise, up to a transmitted power of -70 dBW = -40dBm for a gain of 10, comfortably satisfying the sensitivity requirements in Table 1.1.

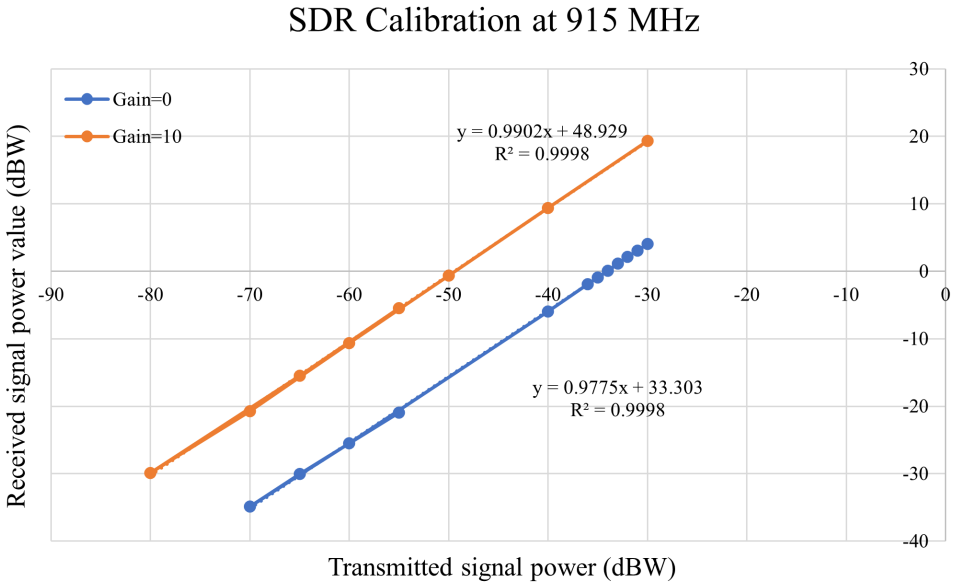


Fig. 3.10 Graph of received signal power by SDR vs transmitted power from a signal generator at 915 MHz in dB, for 2 different antenna gains.

3.4.2 Testing Transmitter

The final piece of the system used in the project was an ADALM-PLUTO SDR, shown in Figure 3.11. This device was selected due to its ease of configuration and integration with the software platforms chosen for the project (being backed by a large online support community), its dedicated transmit channel capable of generating signals well above and below the range of interest, and its availability, with the RFID research group being able to lend one out to me for the duration of the project. The PLUTO SDR was considered as a candidate to be mounted on the TurtleBot as a receiver instead of the RTL-SDR, however, due to its dimensions and overall size, it wasn't able to fit compactly within the body of the robot and was thus discarded.

In a real scenario, the robot would be deployed in the area of interest, and the existing RF field would be sampled, caused by whatever devices and processes are present. As a means to evaluate the accuracy of the RF measurement system, the PLUTO SDR was used as a transmitter to create a field pattern governed by a signal of known, constant properties. This greatly simplified testing, as it enabled the correctness of the RF data collection system to be assessed, in an otherwise predominantly noisy (and completely unknown) environment.

A MATLAB script for the PLUTO was written, to send out a pure tune (constant amplitude carrier wave) at 915 MHz continuously. Spectrum analyser testing revealed that transmitted power is 1.91 dBm when running the script. It was then placed at a corner of the testing arena at the same height as the antenna on the robot's SDR (to minimize reflections) and in direct line of sight with the robot (i.e. was never obscured by obstacles). It was also placed at least a full wavelength away ($\sim$ 30 cm) to minimize near-field effects [32].

Fig. 3.11 Picture of ADALM-PLUTO SDR used in the project. Taken from `https://www.analog.com/`.

Implementations and Analysis

This section describes the path planning, navigation and data collection & processing programs, which are the three main systems developed during the project. It looks at the phases of development each system went through before reaching its final iteration, explains how the algorithms designed work and how they were implemented into code, and provides an analysis of the results obtained after testing each of the final systems.

4.1 Path Planning

The first development focused on using the PGM environment file generated in the mapping stage to find an optimal path that solves the CPP problem in the testing environment. This path must avoid collisions with obstacles/walls and maximize area coverage in order to achieve the highest possible resolution and accuracy for the RF field map. There have been several approaches explored in this domain, as mentioned in the Literature Review (Section 2), including randomized trajectories, and optimal solutions. The main distinction revolves around whether the algorithm is offline (the trajectory is computed beforehand, when the map is readily available) or online (the trajectory is computed on the go relying on information from sensors). This project will exclusively focus on offline methods, as these avoid errors in sensing, and lead to optimal trajectories, minimizing backtracks and maximizing coverage (provided the map is reasonably high quality). It will also deal with static environments only (where obstacles are stationary), to simplify the constraints.

Several methods were trialled before landing on a successful approach. They all started by randomly initializing N sampling points (nodes) in the free area, and joining adjacent nodes to form a graph. Only nodes less than 15 cm apart were joined together in a straight line, becoming neighbours. The procedures tested were:

- Framing the problem as a Travelling Salesman Problem (TSP) [33] which seeks to visit all nodes in a graph minimizing distance travelled, and implementing a relevant algorithm from the literature.

- Using the graph traversal algorithm, Depth-First Search (DFS) [34] to find a trajectory that visits all the nodes from the starting point.
- Using a PRM to navigate from one node to another, and applying this technique for all nodes in the graph until every node has been visited at least once.
- Navigating from node to node by using an RRT, based on proximity to nearby nodes.

While these approaches worked, a rough implementation in python proved that the trajectories are sporadic, inefficient and can run into obstacles in cluttered environments. One such example is shown in Figure 4.1, and corresponds to the method utilizing DFS. Hence, after carefully reviewing the literature to see what other solutions existed, the algorithm chosen was the Spanning Tree Coverage algorithm (STC) [18], which employs a fundamentally different setup.

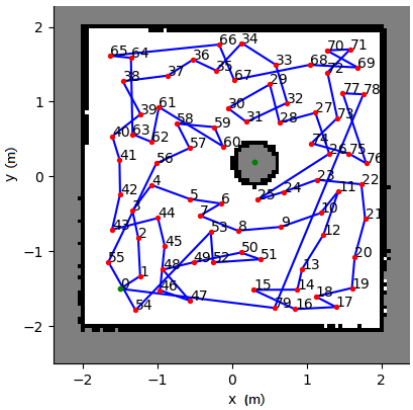
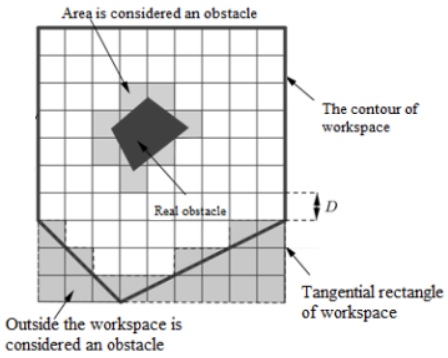


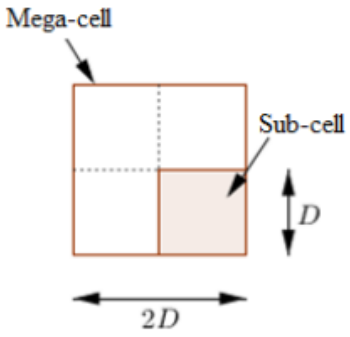
Fig. 4.1 CPP path generated by DFS python implementation. Nodes are shown in red, and trajectory in blue, with each node labelled according to its visiting order.

STC Algorithm

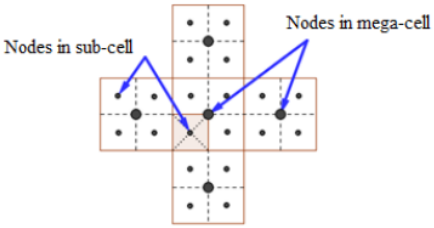
The algorithm starts by decomposing the working environment, referred to as the Defined Operational Environment (DOE), into a grid of cells, where the size of each cell corresponds to the diameter, D, of the robot as indicated in Figure 4.2a. This is the Occupancy Grid (OG). Each individual cell is referred to as a sub-cell, and is labelled as either occupied or free according to whether an obstacle lies inside it. Four adjacent sub-cells are grouped into a mega-cell, leading to N mega-cells 2D × 2D in size, M_1, M_2, ... M_N as shown in Figure 4.2b. Each mega-cell is characterized by a node located centrally in the cell, and the same configuration applies to each sub-cell, as can be seen from Figure 4.2c. Where the edge between two adjacent mega-cells is completely free, a connected edge between the corresponding nodes on the OG can be created, and if the opposite is true, then the two nodes cannot be connected through this edge. Figure 4.2d shows the connections of adjacent mega-cells. Node N_c can be connected with nodes N_e, N_s and N_w because the common edges between them (B_e, B_s and B_w) are free. Graph traversal algorithms like DFS can be used to build a spanning tree by connecting adjacent nodes that follow these rules.



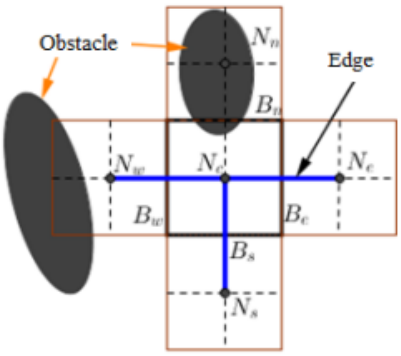
(a) Decomposing DOE into sub-cells



(b) Grouping 4 sub-cells into a mega-cell



(c) Placing nodes in each sub-cell and mega-cell



(d) Edges between adjacent mega-cells

Fig. 4.2 Collection of Diagrams explaining the division of a DOE into sub-cells and subsequently mega-cells to illustrate the STC algorithm. Taken from [18].

Algorithm 1 shows the offline STC algorithm variant implemented in this project, as described in research paper [18]. The OG and starting cell position, s, of mega-cell M_s (starting point for the robot to traverse the DOE), are the inputs to the algorithm. A spanning tree is created in line 1 using the DFS algorithm. Depending on the characteristics of each mega-cell in relation to obstacle obstruction, the connection edge between nodes is different, as described previously. The path of the robot in line 2 starts from sub-cell s. The robot performs the right-side spanning tree (for nodes in sub-cells) to move around the spanning tree for nodes in mega-cells, and ensure complete coverage (line 16). Whenever the centre of the robot is located on the boundary of the C-Space [1] the robot will move on the boundary from right to left and return to the previous mega-cell (line 21). The algorithm stops when the robot returns to the original mega-cell M_s. The final result is illustrated in Figure 4.3. Various GitHub repositories implementing STC algorithms were consulted [35, 36], and parts of their solutions were used as the starting point for the version of the STC program written for this project, explained in Algorithm 1. The existing code had to be adapted, as it implemented a less advanced version of STC, where if a mega-cell is occupied by part of an obstacle, then the entire mega-cell is also seen as an obstruction occupied entirely by an obstacle. This leads a robot to consider the entire cell as an obstruction and fail to cover the cell area, hence, failing to cover the entirety of the DOE.

[1] The configuration space, or "C-Space", of a robot is the set of all possible positions a robot may occupy in the DOE.

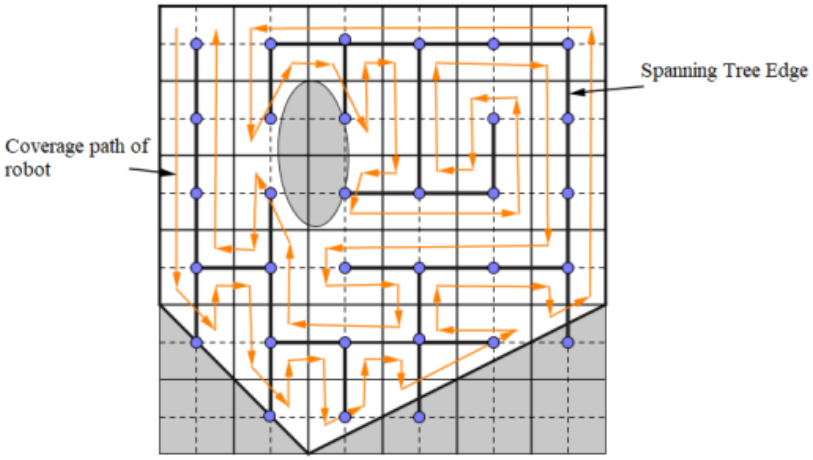


Fig. 4.3 Diagram of spanning-tree and final robot path generated by STC Algorithm. Taken from [18].

Algorithm 1: Spanning Tree Coverage Algorithm

Input: Grid G includes N mega-cells $M_1, M_2, ...M_N$ and a starting cell s of MegaCell M_s
Output: A coverage path P that cover all reachable cells in G from s

$T = Create\_Spanning\_Tree(G, M_s)$
$P = Coverage\_Step(s, T)$
return P
Function `Create_Spanning_Tree(`G, M_s`)`**:**
    T = Tree consisting only of the MegaCell M_s
    $visit(M_s)$:
    for *each MegaCell W adjacent to* M_s *and not yet in* T **do**
        add W and edge $\{M_s, W\}$ to T
        $visit(W)$
    end
Function `Coverage_Step(`s, T`)`**:**
    $P \leftarrow \varnothing$
    current.Cell ←s
    current.MegaCell ←M_s
    while $\exists\, M_i \in G$ *that* $\exists$ *edge* $\{M_i$, *current.MegaCell*$\} \in T$ **do**
        if $\exists\, M_j \in G$ *that* $\exists$ *edge* $\{$*current.MegaCell*, $M_j\} \in T$ **then**
            add path from *current*.Cell to *next*.Cell, to set P
            current.Cell ← *next*.Cell
            current.MegaCell ← MegaCell(*next*.Cell)
        else
            add path from *current*.Cell to Cell(*last*.MegaCell), to set P
            current.MegaCell ← *last*.MegaCell
            current.Cell ← Cell(*last*.MegaCell)
        end
    end

CPP Algorithm

The CPP algorithm takes in the raw PGM map file generated by SLAM, and outputs the set of points that make the optimal area coverage path. At the heart of this program is the STC algorithm discussed in the previous section, but in order to be compatible with the hardware and software elements present in this project, a few additional steps had to be added before and after the STC function call.

Each pixel in the PGM map file corresponds to a 1 cm × 1 cm square of the actual area (as the LiDAR resolution in SLAM was chosen to be 1 cm for the highest accuracy). This means the DOE has already been decomposed into 1 cm × 1 cm cells. We define an Occupancy Grid class that slices the map to contain only the information in the area of interest, and converts the PGM file into a matrix where each cell corresponds to a pixel, and the value of the cell is assigned to 1 if it is occupied, and 0 if it is free. The robot can be approximated as a square of side length 16 cm (from measuring its diameter, D). Using this value of D, the obstacles (i.e. occupied cells) are inflated by an enlarged robot radius (D/2) plus 1 cm to account for possible errors in localization. This defines the C-space for the robot. The OG class also contains methods to find a mapping from an (x, y) position in the actual map to a cell in the matrix, and an inverse mapping from a cell in the matrix to a real (x, y) position in the actual map. This OG alongside the starting pose of the robot (a vector of [x, y, θ] measured by the localization program as per Section 4.2) and the desired spatial sampling are fed to the CPP algorithm as input parameters.

After the map file has been pre-processed into an OG, as explained above, the CPP algorithm can be used. In order to achieve variable spatial sampling, the OG is resized by using the *cv2.resize* image processing function from the OpenCV Python library [37]. Resizing involves using an interpolation method, to effectively combine the matrix cells together, scaling down the matrix by the spatial sampling chosen. For instance, if a spacing of 10 cm is selected, this means 10 × 10 matrix cells corresponding to 1 cm × 1 cm squares in the actual map, are grouped together to form a single cell in the new scaled-down matrix. There is no limitation on the value of spacing that can be chosen, but any value less than 1 cm causes the OG to be scaled up rather than down. When the STC function is now called, the points in the path will be 10 cm apart as desired. Several interpolation routines were tested, with "nearest integer interpolation" being selected as it was the most conservative. After the resizing step, the values of the OG are checked, and anything other than a 0 (free space) is assigned a value of 1 (occupied space). This is because during resizing, the value of the resulting cells will be a function of the individual grouped cells' values (the function varies according to interpolation method), and any value different from zero implies at least one of the individual cells that was grouped, was occupied.

At this point, the function implementing STC on the matrix is called, resulting in a path of points covering the area, starting with the cell closest to the starting pose. These points refer to cells in the matrix, so the final step is to use the inverse mapping method in the OG class to convert them into coordinate locations in the actual map. The output is a YAML file [2] containing the (x, y) coordinate and robot orientation for each point in the CPP path, to be used in the navigation algorithm, with the starting pose of the robot prepended to this list. A visual tool was also written to display the path overlaid onto the map file, for debugging and validation purposes.

Results

Figure 4.5 shows examples of the CPP algorithm being used to find trajectories in different environments, including a simulated world and the real testing arena, both with and without obstacles. This proves that the CPP program still works well even when the map files generated are significantly noisier, as in the case of Figures 4.5b and 4.5c. Moreover, as can be seen from Figures 4.5a and 4.5c, the algorithm also functions properly in environments presenting obstacles lying inside the DOE, in addition to walls/barriers, where the gap around the obstacles is due to the robot's physical size. Finally, as is clear from the difference in dimensions of the simulated world and the testing arena, the CPP script works well for map files of any size.

Figure 4.6 shows examples of the CPP algorithm being used to generate paths with varying spatial sampling for the real empty arena. It shows how it is able to generate CPP paths for 3 different sample spacings, with each sampling point along the path being separated by the spacing chosen, and the total number of sampling points scaling in proportion. It is not surprising to note that for smaller spacing, the algorithm takes longer to run, as the graph of nodes increases in size, and it becomes harder to compute a path. A graph of runtime in milliseconds against spatial spacing for the real empty map of Figure 3.7 is shown in Figure 4.4. This phenomenon is also true for larger maps, as is expected.

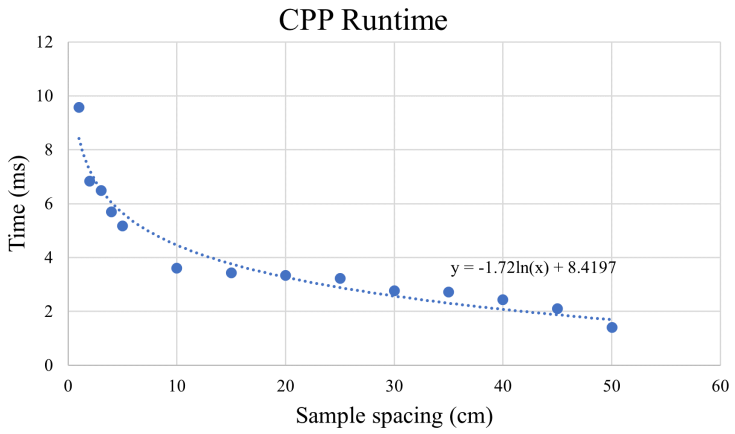
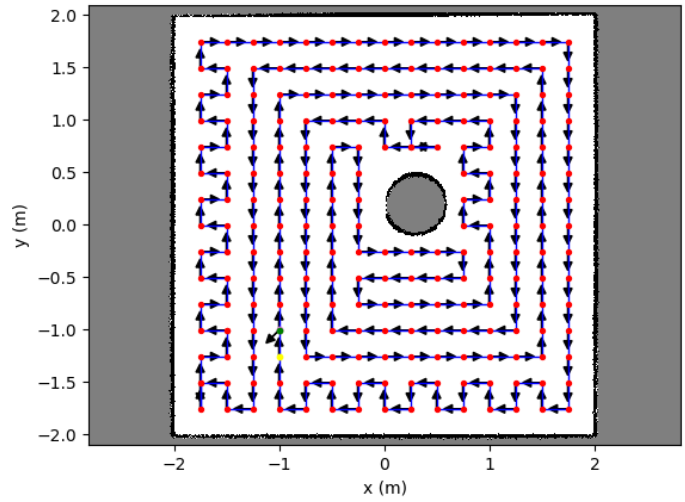
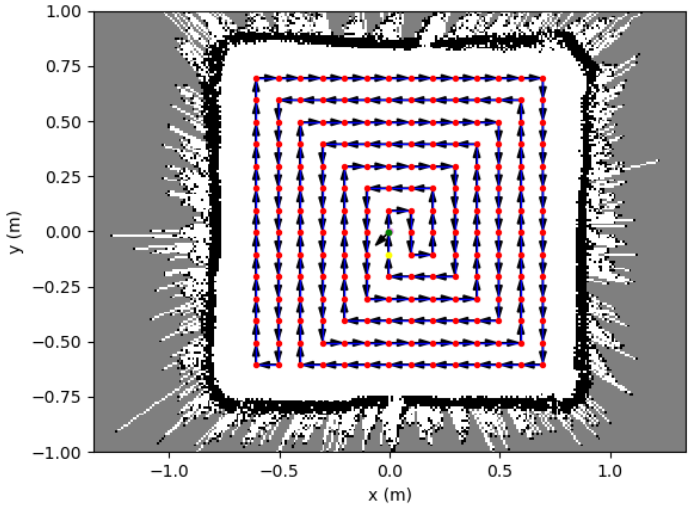


Fig. 4.4 Graph of CPP algorithm runtime vs sample spacing for the empty testing arena environment.

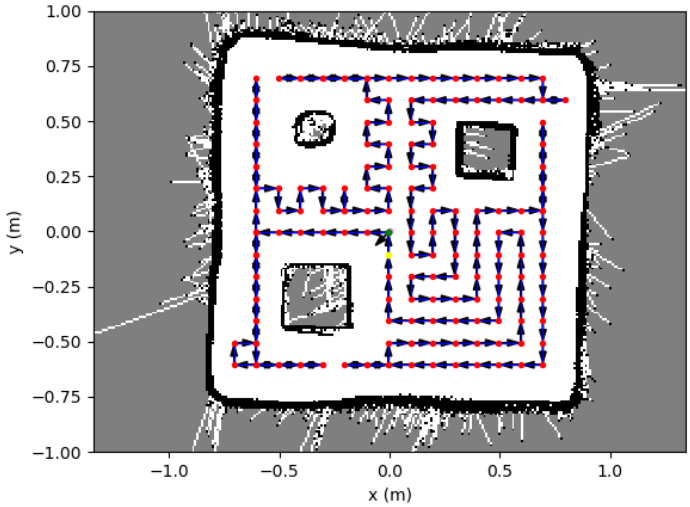
[2] A YAML file is a text document that contains data formatted using YAML, a human-readable data format used for data serialization. It is used for reading and writing data independent of a specific programming language.



(a) Simulated map in Figure 3.5

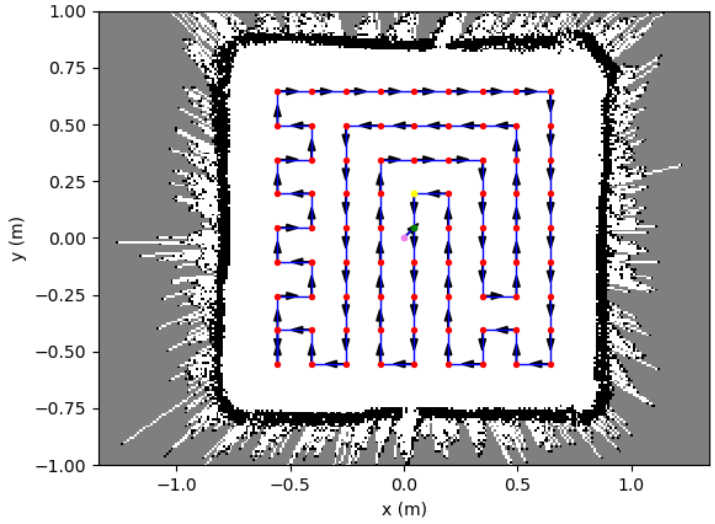


(b) Real empty map in Figure 3.7

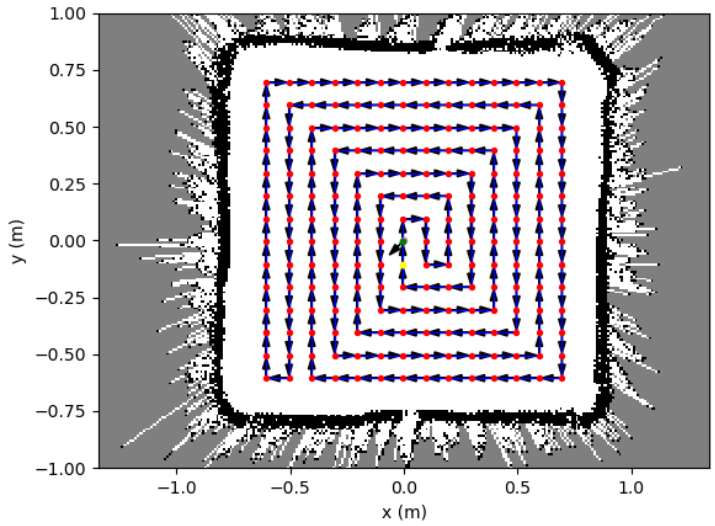


(c) Real map with obstacles in Figure 3.8

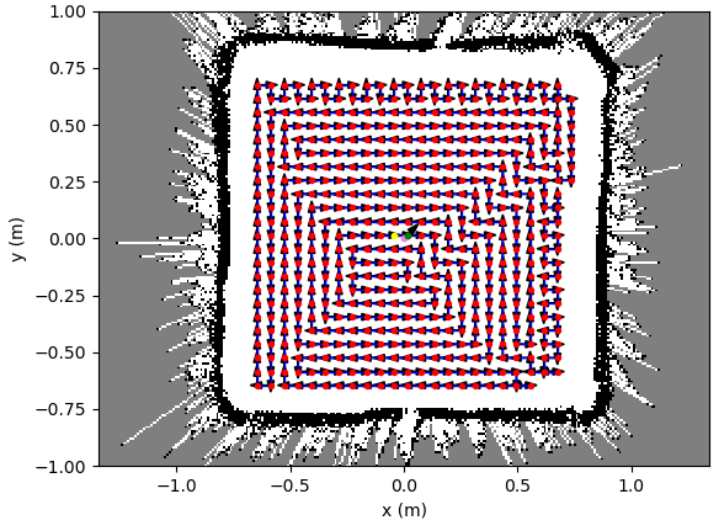
Fig. 4.5 Paths generated by CPP algorithm for environments discussed in Section 3.3. Sampling points in red, path in blue, and black arrows showing the direction along the path. Starting pose in violet, first and last points on the path are shown in green and yellow respectively.



(a) 15 cm spacing



(b) 10 cm spacing



(c) 5 cm spacing

Fig. 4.6 Paths generated by CPP algorithm for the empty arena in 3.7 with varying sample spacing. The colour coding convention is the same as in Figure 4.5.

4.2 Path Tracking

After computing the sampling points that make up the coverage path, the next task is to design a program that calculates the translational and rotational velocity components required to guide the robot from point to point along the path, ensuring the trajectory matches as closely as possible the CPP path, to avoid colliding with obstacles. This comprises the localization, controller and navigation systems, which are the focus of this section.

Localization

Localization refers to a robot's ability to establish its own position and orientation within the frame of reference defined for its environment. It's one of the most fundamental competencies required by an autonomous robot, as the knowledge of the robot's own location is an essential precursor to making decisions about future actions. There are many localization techniques available, depending on the sensing equipment accessible to the robot [38]. The TurtleBot robot is equipped with an IMU, which can be used for odometry, and a LiDAR sensor. This opened up several alternatives to estimate the pose of the robot, using each sensor available.

Firstly, a dead-reckoning approach was prototyped, where the sampling points in the CPP path were converted into a set of instructions telling the robot to move forwards/backwards for a certain distance, or turn clockwise/anticlockwise by a specific angle, bypassing pose estimation. Unsurprisingly, this approach worked poorly, as the robot was unaware of its position and surroundings, and sensor noise caused distances and angles to be over/underestimated. These errors accumulated over time, causing the robot to follow a very different trajectory from that desired, even in simulation. In real-world testing, friction and other similar effects added even further uncertainty. The next iteration used odometry measurements, combined with map data. This technique worked well in simulation, but only slightly better than dead-reckoning in a real environment. This is because the IMU was very noisy and accrued drift over time, which led to significant errors in the location estimate of the robot. The final method utilized the LiDAR sensor alongside a particle filter and map information. The particle filter was used to represent the distribution of likely states, with each particle representing a possible state, (hypothesis of where the robot is). The system starts with a uniform random distribution of particles over the configuration space, meaning the robot has no information about where it is and assumes it is equally likely to be at any point in space. Whenever the robot moves, it shifts the particles to predict its new state after the movement. Using distance data from the LiDAR sensor, the particles are resampled based on recursive Bayesian estimation (how well the actual LiDAR data correlates with the predicted state). Ultimately, the particles converge towards the actual position of the robot. Another approach would have been to combine data from odometry and LiDAR to obtain a better estimate of the robot's position. This will be discussed in Section 6.2

Kinematic Motion model

Before designing the controller, the kinematics of the robot need to be understood. The Turtlebot3 burger robot is a differential-drive robot whose movement is based on two separately driven wheels placed on either side of the robot body. It can thus change its direction by varying the relative rate of rotation of its wheels, and hence does not require an additional steering motion. To balance the robot, the TurtleBot includes an additional caster wheel. Figure 4.7 shows a diagram of the motion model for the TurtleBot robot. The sign-convention and notation shown will be assumed for the remainder of the report. Note that the forward translational velocity is u, and the positive angular velocity, ω is in the counter-clockwise direction.

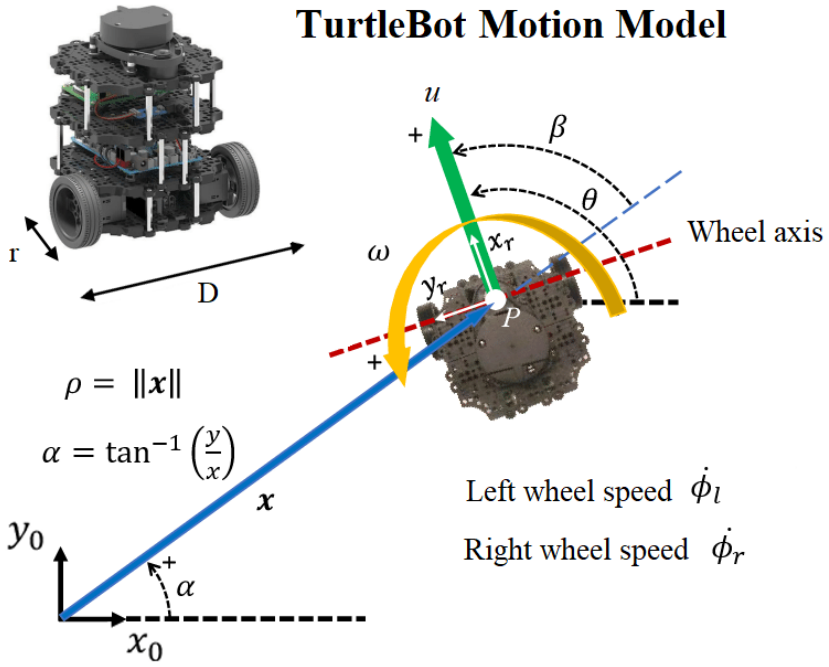


Fig. 4.7 Motion model for a differential drive robot

Clearly, from the diagram in Figure 4.7, it follows that:

$$u = \frac{r\dot{\phi}_r}{2} + \frac{r\dot{\phi}_l}{2} \tag{4.1}$$

$$\omega = \frac{r\dot{\phi}_r}{D} - \frac{r\dot{\phi}_l}{D} \tag{4.2}$$

Where $\dot{x_r} = u$, $\dot{y_r} = 0$ and $\dot{\theta} = \omega$

With respect to the global coordinate frame defined by x_0, y_0 this gives:

$$\begin{bmatrix} \dot{x_0} \\ \dot{y_0} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x_r} \\ \dot{y_r} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} u\cos\theta \\ u\sin\theta \\ \omega \end{bmatrix} \tag{4.3}$$

Robot Controller

The controller is a critical part of the navigation algorithm, as it keeps the robot on course by correcting for sensor noise and real-world disturbances like friction. The code snippet in Listing 1 shows the *Go_to_point()* function, which is responsible for implementing the controller in the code. The function take in the coordinates of the sampling point as a tuple, [x, y], and the orientation of the robot heading into the next point. This controller is based upon the design provided by ROBOTIS [28] for the TurtleBot specifically, and the program developed in [39], which realizes a generic open-loop proportional controller. Various major modifications were carried out:

1. The localization method was changed from using odometry measurements, to LiDAR data, adding a particle filter as explained in 4.2, for enhanced state estimation. The function *slam.update()*, was written to perform localization, and returns the estimated pose of the robot as a tuple, [x, y, θ] when it is called.

2. The open-loop design was replaced by a closed-loop controller with negative feedback, to increase robustness. This is achieved through a while loop, which runs until the robot is within 2.5 cm of the sampling point, as measured by the localization system. This value was a trade-off between accuracy and performance, with larger values leading to poor sampling spacing, and smaller values placing too high a demand on the hardware. It also satisfies the requirement in Table 1.1.

3. The robot poses are regularly logged in a list, so that the trajectory can be visualized in the post-processing stage.

In proportional control, the forwards and angular speeds of the robot are proportional to the difference (error) between the desired/reference and current values. This takes place on lines 35 and 38 in the code, respectively. This controller has two proportional gains, one for the translational speed and one for the rotational speed, which were tuned by a mixture of trial and error, and a set of empirical rules contained in [40]. An investigation was carried out into Proportional, Integral and Derivative (PID) control [41], with a PID controller script being written and tested. The controller gains were tuned following the Ziegler-Nichols [42] guidelines, but this proved significantly harder, as 6 gain parameters (k_p, k_d, k_i for u and ω) had to be tuned for successful operation, as opposed to just 2 (k_p only) in proportional control.

```python
def Go_to_point(self, point, angle):
    (goal_x, goal_y, goal_z) = (point['x'], point['y'], angle)
    # Use localization to find (x, y) position and orientation θ
    self.slam.update()
    self.position = Point(self.slam.pose[X], self.slam.pose[Y], 0)
    rotation = self.slam.pose[YAW]
    # Add robot position to list to construct trajectory at the end
    self.positions.append((self.position.x, self.position.y))
    # Set proportional gains
    linear_speed = 0.03             # Forward speed gain
    angular_speed = 0.3             # Rotational speed gain

    goal_distance = sqrt(pow(goal_x - self.position.x, 2) + pow(goal_y - self.position.y, 2))
    distance = goal_distance
    old_distance = distance

    while distance > 0.025:

        self.slam.update()
        self.position = Point(self.slam.pose[X], self.slam.pose[Y], 0)
        rotation = self.slam.pose[YAW]
        # Current position
        x_start = self.position.x
        y_start = self.position.y
        # Find difference between desired and current value (errors)
        path_angle = atan2(goal_y - y_start, goal_x- x_start)
        distance = sqrt(pow((goal_x - x_start), 2) + pow((goal_y - y_start), 2))
        z_dist = min(abs(rotation - path_angle), abs(2*pi - abs(rotation - path_angle)))

        # Detect and halt open loop behaviour
        if distance > old_distance + 0.1:
            break
        # Proportional control for linear speed
        self.move_cmd.linear.x = min(linear_speed * distance ** 1.1 + 0.01, linear_speed)
        # proportional control for heading
        if path_angle >= 0:
            if rotation <= path_angle and rotation >= path_angle - pi:
                self.move_cmd.angular.z = angular_speed * z_dist / (abs(self.move_cmd.linear.x) + 0.65)
            else:
                self.move_cmd.angular.z = -1 * angular_speed * z_dist / (abs(self.move_cmd.linear.x) + 0.65)
        else:
            if rotation <= path_angle + pi and rotation > path_angle:
                self.move_cmd.angular.z = -1 * angular_speed * z_dist / (abs(self.move_cmd.linear.x) + 0.65)
            else:
                self.move_cmd.angular.z = angular_speed * z_dist / (abs(self.move_cmd.linear.x) + 0.65)

        old_distance = distance
        # Send u, ω to OpenCR motor control board on robot
        self.cmd_vel.publish(self.move_cmd)

    self.slam.update()
    self.position = Point(self.slam.pose[X], self.slam.pose[Y], 0)
    rotation = self.slam.pose[YAW]
    self.positions.append((self.position.x, self.position.y))
    # Send stop message
    self.cmd_vel.publish(Twist())
    self.r.sleep()
```

Listing 1 Robot Controller Function

Main Navigation Loop

All the elements discussed previously play a big role in the main navigation loop, as we shall see in this section. This is the program that is run to start the testing sequence, and the essence of it is captured in the following code snippet, Listing 2. Firstly, the program loads the YAML file containing the coordinates of the sampling points in the CPP path. Then, several key elements are initialized, including establishing a link with ROS, setting up and configuring the SDR on the robot for the test, and creating a data structure to store the measurements taken during the experiment. Next, a *for* loop iterates through all the samples in the CPP path, and for each sample, a timer is started, which gives the robot 30 seconds (determined experimentally, as a function of controller parameters) to navigate from the current point to the next point, using the controller function, *Go_to_point()* explained in the previous section. If the robot finds the sampling point, it stops, makes a note of its location [3], and calls the functions to measure the RF field strength at the chosen frequency of 915 MHz and calculate the signal power (More detail in Section 4.3), storing it in the variable created at the start of the program for this purpose. Should the robot not find the sampling point in time, it simply moves on to the next one. This method stops erratic behaviours if the robot cannot find a point, while minimizing runtime. Finally, when the robot completes the route, the RF and trajectory data files are saved for analysis. The final block of code (line 40) is a failsafe, which stores the data collected up to that point, in case an unexpected issue occurs during the experiment, like the robot-computer connection being interrupted, or the robot running out of battery.

```
1  with open("/home/luis/catkin_ws/src/IIB_Project/part2/python/route.yaml", 'r') as stream:
2          cpp_samples = yaml.safe_load(stream)    # Load sampling point YAML file generated by CPP algorithm
3  try:
4      # Initialize ROS
5      rospy.init_node('follow_route', anonymous=False)
6      navigator = Nav.GoToPose()  # Set-up robot controller Class
7      # Initialize SDR
8      sdr = RtlSdrTcpClient(hostname='192.168.228.210', port=55366)
9      SDR.configure_device(sdr, center_freq=914.6e6)
10     # Create list to store RF power measurements
11     data = []
12
13     for sample in cpp_samples:
14         # Ensure ROS communication is working
15         if rospy.is_shutdown():
16             break
17         # Set 30s timer for robot to reach next point
18         signal.signal(signal.SIGALRM, handler)
19         signal.alarm(30)
20         # Navigation Loop
21         try:
22             navigator.Go_to_point(sample['position'], sample['rotation'])   # Call robot controller function
23             # Stop to take measurement at sample point
24             time.sleep(1)
25             # Get actual location in map
26             sdr_pose = navigator.return_pose()
```

[3] This step is necessary, as the actual sampling location will differ slightly from the CPP sampling location due to the 2.5 cm allowable tolerance in the controller.

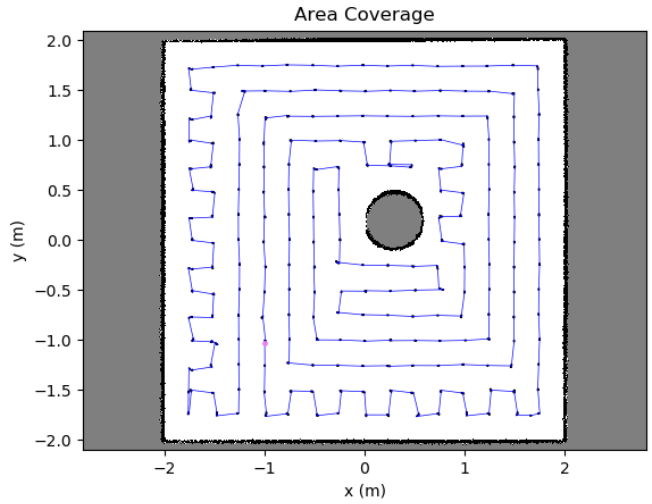
```
            # Measure and calculate RF signal power
            rf_samples = SDR.receive_samples(sdr)
            max_power, _ = SDR.get_power_from_PSD(rf_samples, sdr, freq=915e6, plot=False)
            # Save data in list
            data.append(np.array([sdr_pose[X], sdr_pose[Y], max_power]))

        # Move on to next point if timer exceeds 30s
        except Exception as exc:
            print(exc)
    # Store RF and trajectory data as YAML files for post-processing
    generate_rf_data(data)
    navigator.generate_path()
# Generate RF and trajectory data as failsafe
except rospy.ROSInterruptException:
    rospy.loginfo("Ctrl-C caught. Quitting")
    generate_rf_data(data)
    navigator.generate_path()
```

Listing 2 Navigation Algorithm

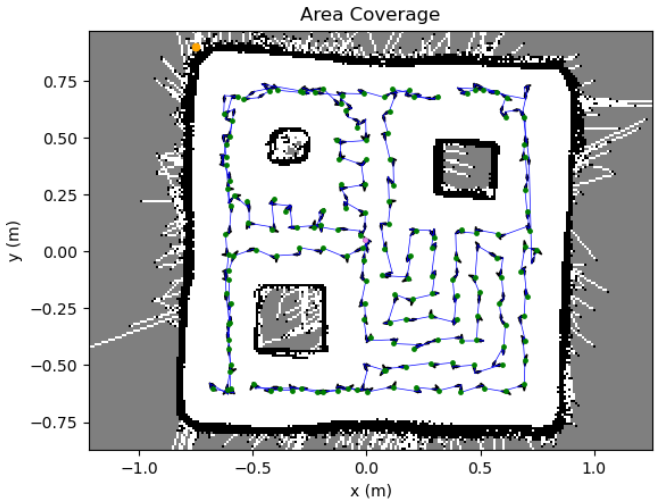
Results

Figure 4.8, displays the actual trajectories traced out by the robot while attempting to follow the CPP path, for both simulated and real-world environments, utilizing the systems explained in this section. Comparing Figure 4.8a to the other two, it is evident that the navigation program works considerably better in simulation than in real life, as would be expected, due to the simplifying constraints of a simulated world. The robot traces out straight lines and matches the CPP path very closely. Figures 4.8b and 4.8c show the results of real world testing in the same environment. The only difference is the placement of the SDR antenna, which in the case of the experiment in Figure 4.8c partially obscures the LiDAR beam at two angles. These two configurations are shown in Figure 4.10. Whereas the trajectory in Figure 4.8b is not as close to the ideal case as in simulation, the overall trajectory follows the desired path reasonably well taking into account the more demanding conditions of the real-world, with all the features (turns and backtracks) clearly recognizable in the figure. This is not the case for the setup in Figure 4.8c, where the robot presents increased difficulty in executing consecutive 90° turns, instead resulting in a zigzag motion, with straight lines being followed reasonably well. This can be attributed to errors in pose estimation (see Section 5.3), as the localization program has fewer data available, resulting in poorer accuracy. Consequently, the robot struggles to find points on the first try, and so has to loop around exploring its immediate vicinity, hence the irregular patterns.

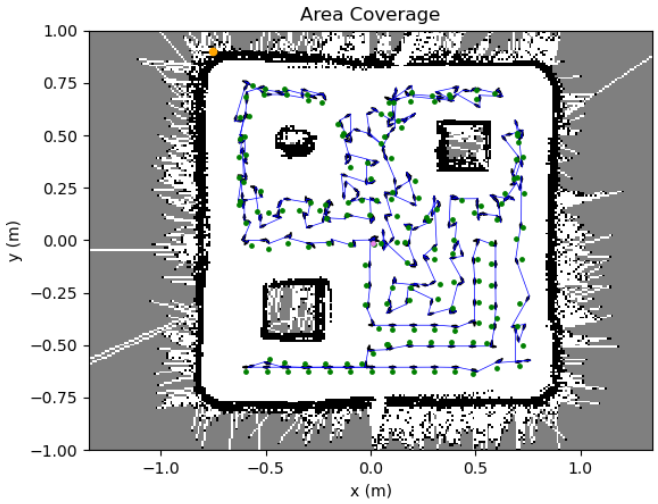
The important draw, however, is that due to the tolerances and safety measures added in each sub-system, while the trajectories aren't perfect, the robot still manages to visit almost every point in the CPP path collecting enough measurements to map out the RF field to the desired degree of spatial accuracy, while always avoiding obstacles.



(a) Simulated environment from Figure 3.5



(b) Real testing arena from Figure 3.8



(c) Real testing arena from Figure 3.8 with LiDAR partially obscured by extra antenna layer

Fig. 4.8 Actual robot trajectories when attempting to follow the route generated by CPP algorithm in different environments and conditions. Trajectories are shown in blue, with black arrows indicating the direction of motion. For Figures 4.8b and 4.8c, the green points indicate where the RF signal was sampled, and the orange point refers to the position of the PLUTO SDR transmitter.

4.3 Data Collection

This section explores all aspects of measuring the RF field strength in the area of interest, looking at the functions designed to capture and store the data, the different SDR antenna configurations tested, and the post-processing of the RF data which is subsequently analysed and compared to simulated data based on theoretical models.

RF Power Measurement

The focus here is to break down the functions associated with RF power measurement that appear in the code snippet for the navigation loop in Section 4.2, which are executed during every test run. Each function will be considered in turn, concentrating on how input parameters are used to achieve the desired outcome. These functions were developed with the help of the PySDR online resource [43], and the *pyrtlsdr* library [30].

- *RtlSdrTcPClient(hostname, port)*: Establishes a remote connection between the SDR dongle attached to the TurtleBot SBC, and the VM running the code. This enables reading from/writing to the SDR as if it were physically connected to the VM. The arguments are TurtleBot IP address, and the port to which the SDR is attached in the SBC, respectively.
- *SDR.configure_device(sdr, center_freq)*: Sets the sample rate at 2.4 Msps, the centre frequency at 914.6 MHz, the frequency correction at 60 PPM, and "software" gain of the SDR to 10. The centre frequency chosen is slightly different from the desired frequency of 915 MHz, to avoid the DC offset spike[4] which would alter the real received signal power measurement.
- *SDR.receive_samples(sdr)*: Performs measurements, reading 256x1024 I/Q samples in the range [913.4, 915.8] MHz, and storing them in a cyclic buffer.
- *SDR.get_power_from_PSD(rf_samples, sdr, freq)*: Uses *matplotlib.pyplot* [24] to estimate and plot the PSD from the samples collected, and uses it to find the un-calibrated signal power at the chosen frequency of 915 MHz. Figure 4.9 shows an example of such a plot.
- *generate_rf_data(data)*: Stores the logged power measurements, and their respective locations, in a YAML file for further analysis.

[4]The DC-offset is caused by leakage of the local oscillator (LO) into the down converter input.

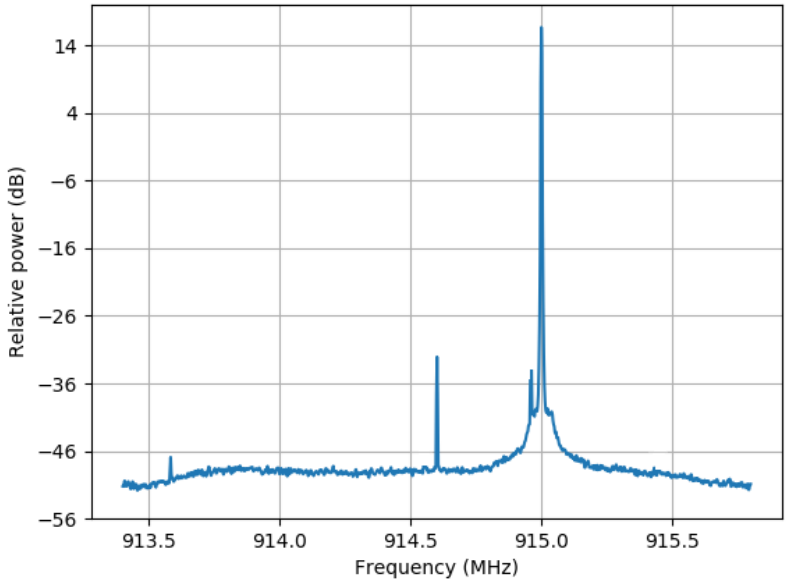
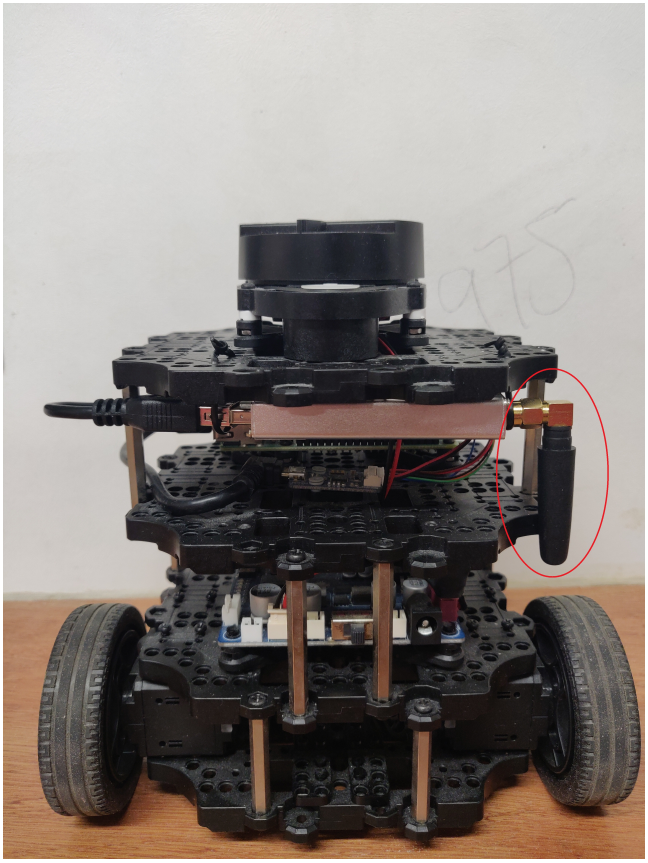


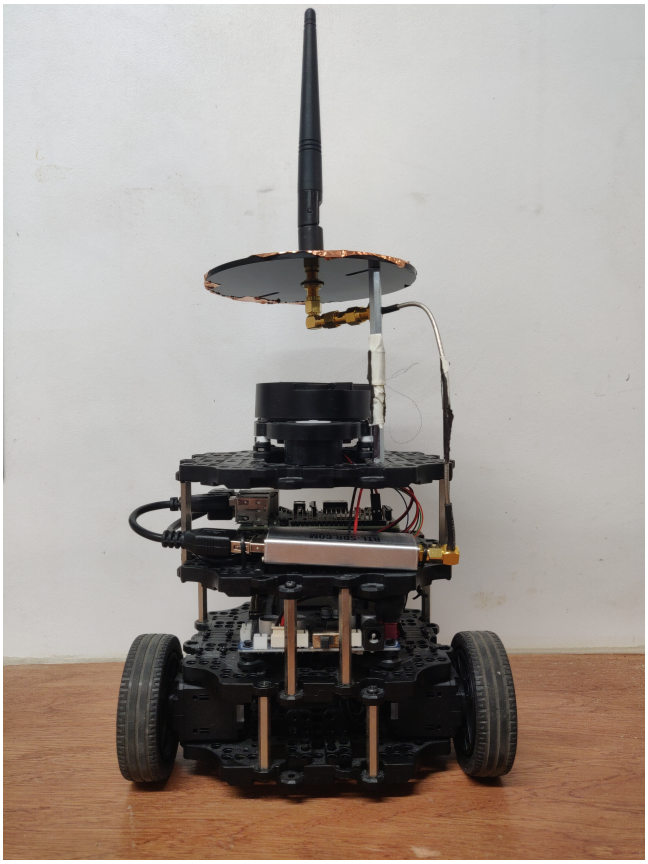
Fig. 4.9 Graph of PSD estimated from I/Q samples collected by SDR, showing a peak at the transmitting frequency of 915 MHz. Note also the DC spike at 914.6 MHz.

SDR Antenna Configurations

As has been alluded to previously in the report, 2 different SDR antenna arrangements were tested on the robot. Photographs of the two configurations are shown in Figure 4.10. The setup in Figure 4.10a shows the SDR dongle attached to the SBC in the third layer, held in place with cable ties, and a small antenna connected directly to the dongle, tucked to the side, so as to not increase the average diameter, D. The arrangement in Figure 4.10b uses a coaxial extender cable, that connects the SDR to a larger antenna mounted on a fifth layer directly above the LiDAR, with the antenna facing upwards. This extra layer is held up by metal stand-offs, which along with the extender cable partially block the LiDAR beam.



(a) Antenna tucked to the side (circled)

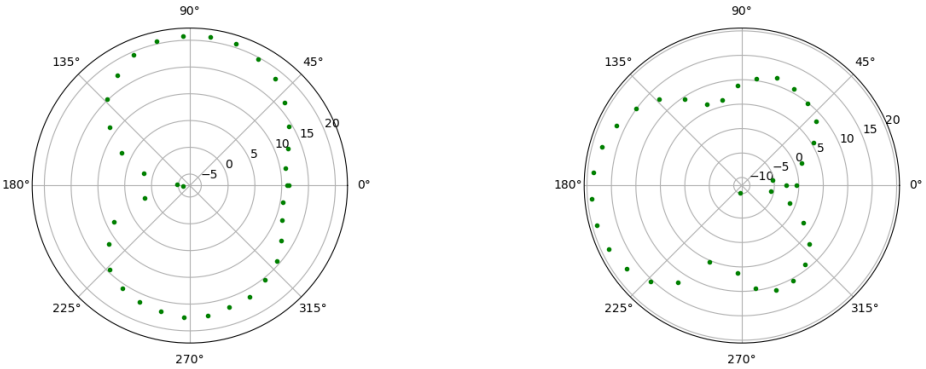


(b) Antenna placed in extra layer on top

Fig. 4.10 Photographs of the 2 different antenna configurations tested in the project

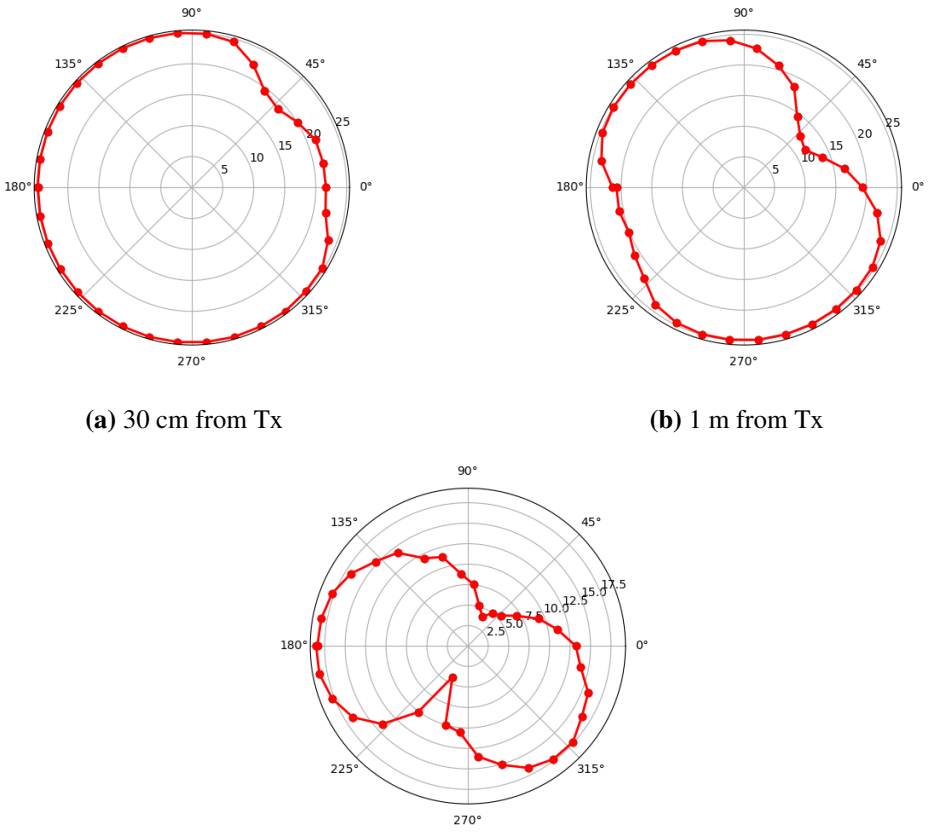
Figures 4.11 and 4.12 show polar plots of antenna radiation patterns at various distances away from the PLUTO SDR transmitter, for each of the antenna configurations. When the RF samples are measured, the SDR antenna is assumed to be omnidirectional, so these results were collected to test the validity of that claim. The signal power values (dBW) shown are before the calibration step in equation 3.1. These plots were generated by placing the TurtleBot in a direct Line-of-Sight (LOS) path between the transmitter and the SDR antenna, and taking a measurement every 10° while the robot rotates on the spot through a full revolution. Testing was carried out in the RFID lab, to minimize reflections, and both antennas were placed at approximately the same height.

Looking closely at Figure 4.11, it is clear that the antenna in the side configuration is not omnidirectional even when the robot is very close to the transmitter, and the radiation pattern becomes more directive as the distance between transmitter and receiver increases. Conversely, the polar plots in Figure 4.12, show that close to the transmitter, the antenna behaves almost like an omnidirectional antenna, and conserves this pattern up to about 1 m. However, again, as the robot is placed further away, the antenna becomes more directive. These observations can be explained because the antennas were placed quite close to the ground ($\sim$ 30 cm) and ground reflections have a strong influence on the measured signal power, becoming more dominant with distance from the transmitter. Although the antenna configuration shown in Figure 4.10a results in a more compact robot arrangement, the antenna is considerably detuned by the robot body, which contains reflective materials such as aluminium stand-offs. The results prove that the configuration in Figure 4.10b is significantly better than placing the antenna to the side, for the purposes of RF field measurement, as the gain can be approximated as being constant all around, which translates into not having to adjust the received signal power based on the orientation of the robot to the transmitter (which won't always be known), justifying our initial assumption.



(a) 30 cm from Tx **(b)** 1 m from Tx

Fig. 4.11 Polar plots showing the radiation pattern for the antenna configuration in Figure 4.10a at two different distances from the PLUTO SDR transmitter.



(a) 30 cm from Tx

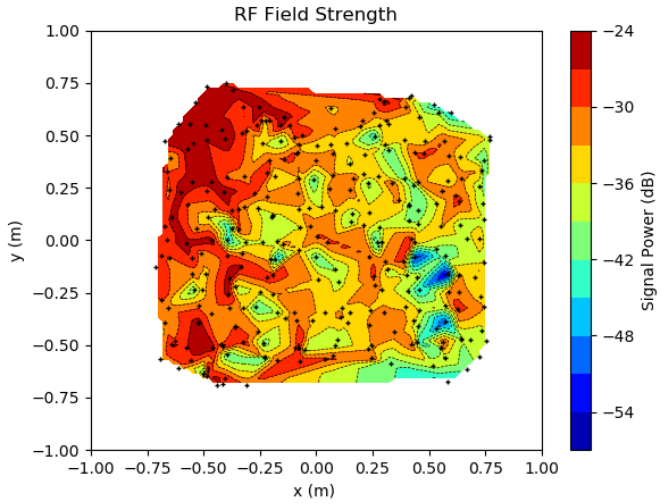
(b) 1 m from Tx

(c) 2 m from Tx

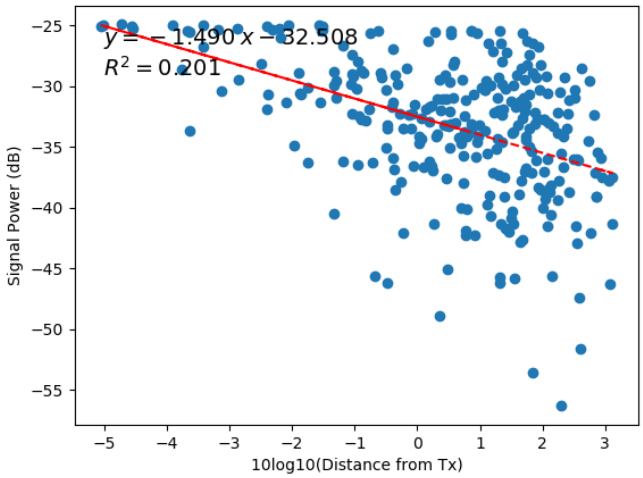
Fig. 4.12 Polar plots showing the radiation pattern for the antenna configuration in Figure 4.10b at three different distances from the PLUTO SDR transmitter.

Results

Figure 4.13 shows a contour plot and a signal decay vs distance graph, for the configuration with the antenna mounted on top, as the results from the polar plots show it leads to better performance. The results for an empty environment are shown (Figure 3.7), as it allows for the highest sampling rate and greater coverage, leading to the most accurate results that can be obtained using the hardware combination chosen for this project. The transmitter antenna was situated in the upper left-hand corner of the testing arena, at (-0.8, 0.8), and was assumed to be omnidirectional. The data plotted is the RF power data logged during the navigation step in the test run, calibrated using equation 3.1. The contour plot in Figure 4.13a is created by plotting the signal power measurements at the locations where they were taken (denoted by black crosses) and linearly interpolating along the x-y grid. The decay plot, in Figure 4.13b, shows a scatter graph of the RF power measurements (dB) plotted against $\log_{10}$(distance from transmitter). The coefficient of determination shows relatively low agreement between the data points, and a line of best has been draw through the data (red trace) with a gradient of approximately -1.5.



(a) Contour plot of RF field strength



(b) RF power vs distance decay plot

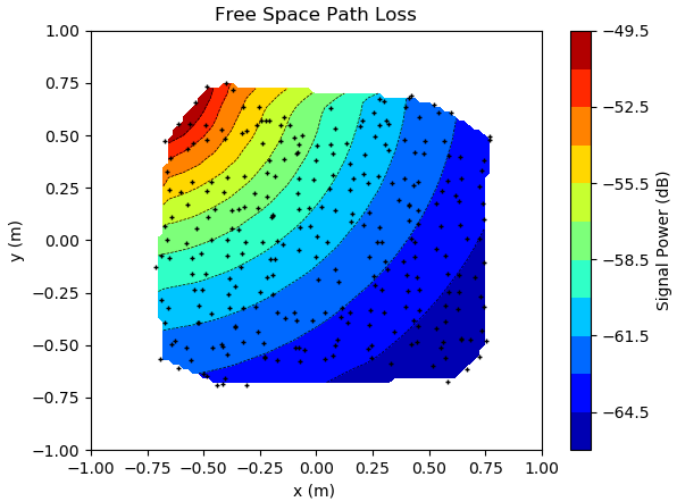
Fig. 4.13 Plots of real RF power results (black crosses) measured by SDR mounted on the robot.

Figure 4.14 shows the RF power contour and decay plot, where the values for signal power have been calculated following the Free-Space Path Loss (FSPL) model [44], keeping the locations of the sampling points constant. The FSPL model predicts the attenuation of radio energy between the feed points of two antennas that results from the combination of the receiving antenna's capture area plus the obstacle-free, line-of-sight path through free space (air). For distance, d in meters, and frequency, f, in MHz, the FSPL in dB, and hence power received, P_r^{dB} are given by:

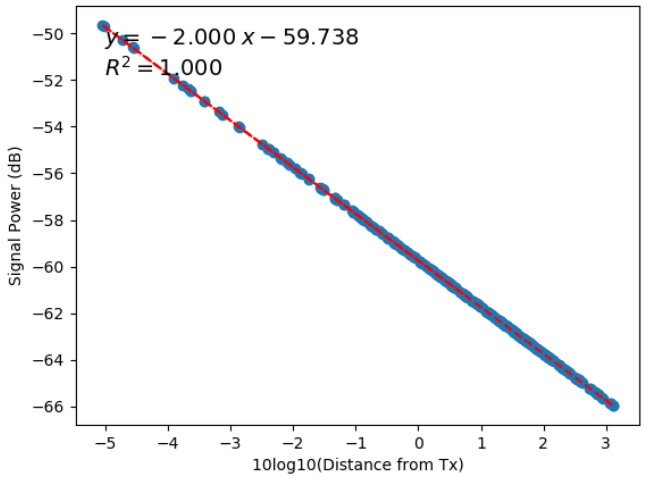
$$FSPL_{dB} = 20\log_{10}(d) + 20\log_{10}(f) - 27.55 \tag{4.4}$$

$$P_r^{dB} = P_t^{dB} - FSPL_{dB} = P_t^{dB} - 20\log_{10}(d) - 20\log_{10}(f) + 27.55 \tag{4.5}$$

The FSPL increases with the square of distance between the antennas, because the radio waves spread out by the inverse square law, and this is captured in the contour plot of Figure 4.14a, clearly showing circular contours of decreasing signal power away from the transmitter which is modelled at position (-0.8, -0.8) as well. The real contour plot in Figure 4.13a resembles this behaviour for measurements close to the source, but doesn't decay following an inverse law. In fact, the plot shows measurements decaying, as d increases, but they appear to be superimposed with noise/fading, causing an irregular pattern. This can be further seen by comparing the signal power vs distance decay plots, where the FSPL model shows a gradient of -2, as expected, with all the points lying on the straight line, whereas for the real case, there is a downwards trend with a similar slope, yet the agreement between the data points is significantly lower. The power measurements themselves are also very different, with the predicted results being significantly lower close to the source, due to ground reflections being absent in the FSPL model.



(a) Contour plot of RF field strength



(b) RF power vs distance decay plot

Fig. 4.14 Plots of simulated RF power results obtained using the FSPL mode

Error Evaluation and Analysis

This section explores some extra minor features of the system, and details several tests that were conducted to characterize errors in the localization and navigation subsystems, presenting and discussing the findings.

5.1 Area Coverage

In addition to plotting the trajectory of the robot on the map to assess the performance of the navigation algorithm, the system can also estimate the percentage of the area the robot covered during the test run, by analysing the logged position data. The first step is to estimate the amount of free area, by counting the free cells in the OG, and multiplying them by the real area of each cell, 1 cm^2 for the highest LiDAR resolution. The next step involves modelling the TurtleBot as a square of side length D, where D is 16 cm, (as per Section 4.1) and using the positions visited by the robot in the path log file, make a note of the cells in the OG that have been visited, making sure to only count them once. Finally, we multiply the number of covered cells by the cell resolution, and divide by the free area to obtain an estimate of coverage.

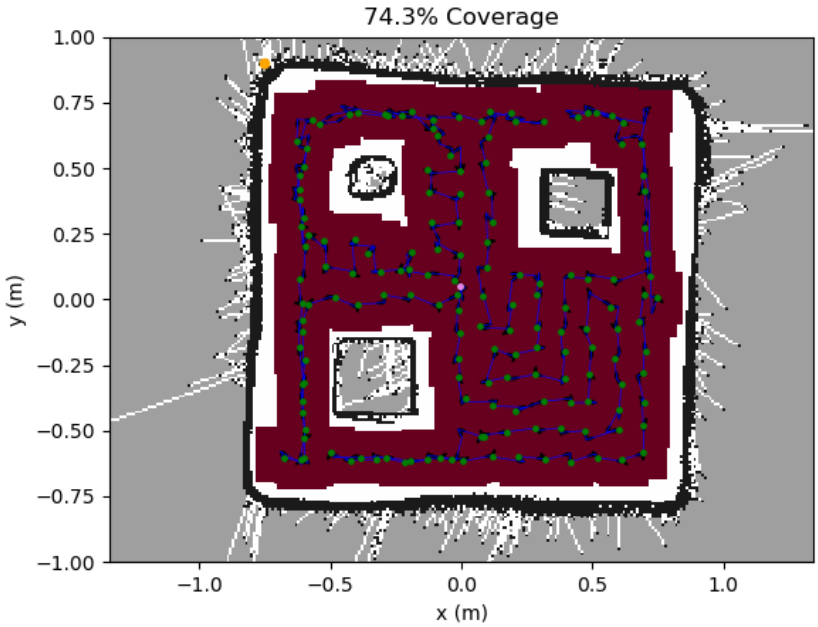
The first variant in Figure 5.1a, shows the covered area in red, while the second version in Figure 5.1b, highlights the cells in the map which have been visited more than once, by shading them in a gradually darker red in proportion to how many times the robot has visited them. This is useful, to spot how many times the robot backtracks, or it could be a sign of the robot not being able to find a sampling point in the arena. The latter results in the robot circling around where it thinks the point should be, and points to a failure in the controller. The results show a coverage of 74.3%, and while this could certainly be improved by upgrading the controller and CPP algorithm, it shows that the robot was conservative in its approach, always keeping a safe distance between the obstacles and the barriers of the arena. It is important to bear in mind that obstacles are enlarged in the CPP stage for this very purpose.

5.2 LiDAR Integration Error

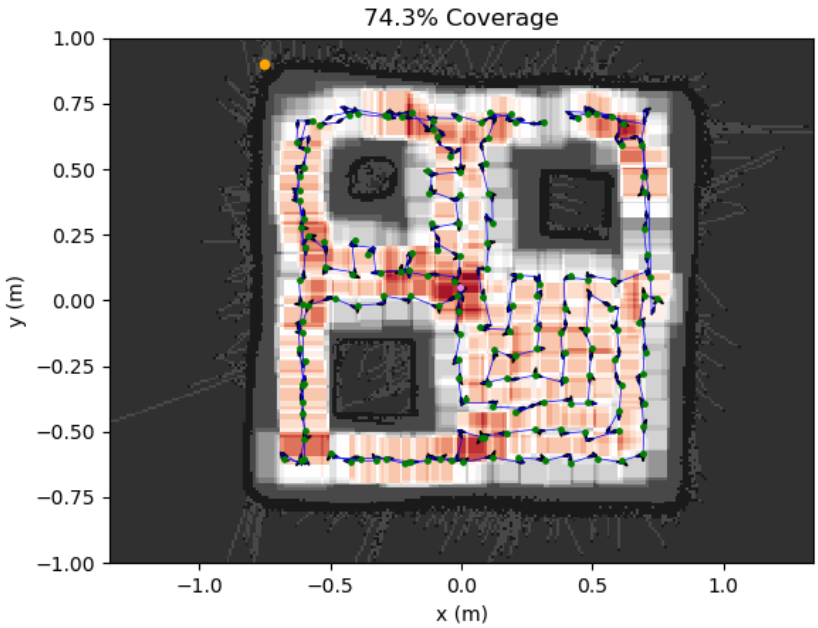
If a LiDAR integration error exists, this would clearly present a problem for long coverage paths, as the estimate of the robot's position would be less and less accurate as the length of the path increased, due to errors adding up, eventually causing it to diverge from the desired trajectory and potentially collide with obstacles in the environment. In this test, the robot was tasked with following a square shaped path of side length 50 cm, in the empty arena, and the final position was compared to the starting position (origin) of the path, to evaluate if the LiDAR position estimate error builds up as the distance travelled by the robot increases. The position error was recorded as the distance between the start and end positions of the robot, measured with a ruler. The results are shown in Figure 5.2, alongside the controller tolerance of 2.5 cm, explained in Section 4.2. The graph shows no apparent correlation between the position errors and the number of loops traversed, with the errors appearing randomly distributed about the controller tolerance value, as expected. Within the limits of experimental accuracy, this data supports that the LiDAR sensor has negligible integration error.

5.3 Pose Estimation Error

This section focuses on how well the robot can estimate its pose in the environment, with the addition of an extra layer for the antenna, resulting in partial blockage of the LiDAR beam. The maximum range of the LiDAR beam is 3.5 m, but in the ranges trialled during this project, (maximum $\sim$ 1.5 m), the distance accuracy uncertainty is between 3 – 5 % and the distance precision uncertainty between 2 – 3.5 %, according to the manufacturer [26]. The angular resolution is 1°. Using the antenna configuration in Figure 4.10b, has led to poorer path tracking as seen in previous sections, as a result of fewer data accessible to the localization system. In an effort to circumvent the side effects of better RF signal measurements with the antenna on top of the LiDAR, some contingency measures were taken, like painting the reflective metal stand-offs that support the extra layer in black or covering them up with tape to minimize reflections. The results of the latter procedure are shown in Figure 5.3. As can be seen from the figure, the localization system doesn't show any signs of improvement after the changes were made, unfortunately.



(a) Area covered by robot shown in red



(b) Area covered shaded proportionally to how many times the robot visited it. From white to red in increasing order.

Fig. 5.1 Plots showing the map area covered by the robot for the trajectory in Figure 4.8b, following the same colour-coding convention.

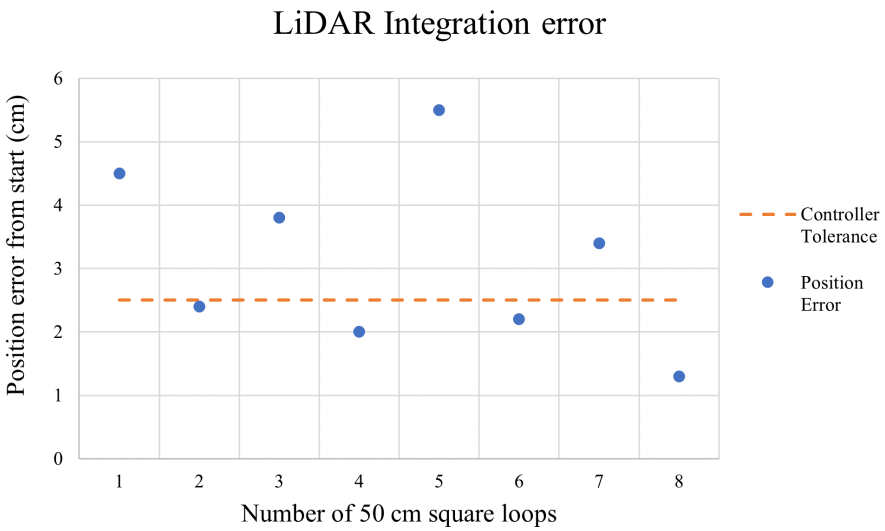
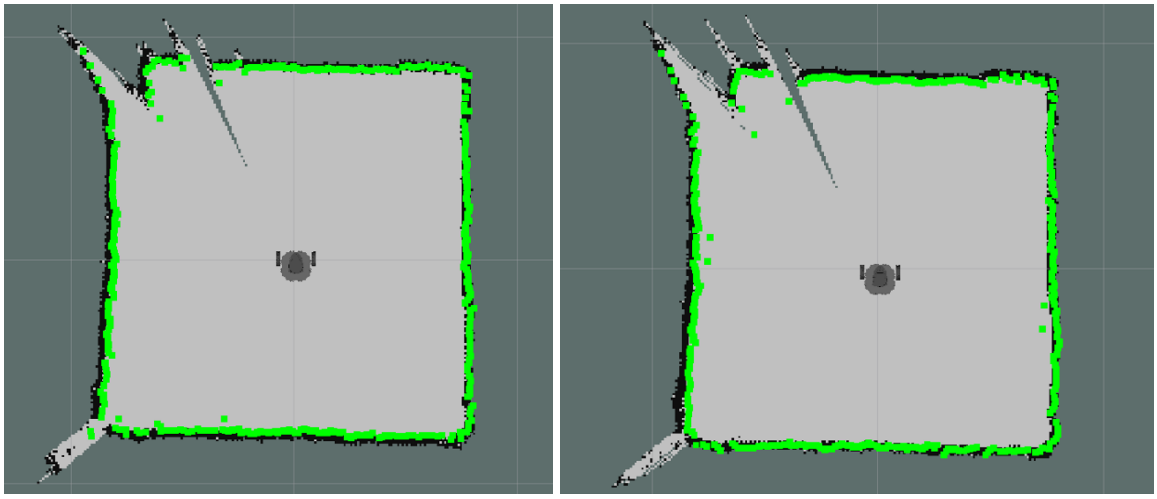


Fig. 5.2 Graph of final position estimate error against number of square loop trajectories completed by robot, to measure the LiDAR integration error.



(a) Stand-offs uncovered **(b)** Stand-offs covered in black tape

Fig. 5.3 Screenshots of RViz window showing the LiDAR point cloud and the robot's estimated pose in the map during the localization step.

Conclusion

The goals of this section are to summarize the main findings of the project, and to give an outlook into a number of possible extensions that could be researched to expand upon the progress made so far.

6.1 Conclusions

After evaluating the results from the individual elements, as well as performance and errors of the integrated system, this section will discuss the takeaways from the project, shedding light on some of the issues encountered, and how decisions made would have been handled differently in retrospect. Particularly, with regard to completion of the set of objectives laid out at the beginning of the project (Section 1.2).

- The path planning algorithm is perhaps the most robust element in the system, as it has responded well to all the tests, managing to work for different quality map files, map sizes and types (obstacles vs empty), and sample spacing requirements.
- The navigation algorithm, although simplistic, appears to be very robust, leading the robot safely along the pre-computed route, failing only in very few situations. It allows the robot to meet and even exceed the pose estimation specification, having an accuracy of $\pm$ 2.5 cm.
- Radiation patterns for the small antenna in the side position, show it is quite poor even when close to the source. The larger antenna mounted in the top layer is almost omnidirectional when close to the source, but as it moves further away it becomes considerably worse, tending towards a directional antenna. This allows us to assume a constant gain for the map sizes tested in this project, but this assumption would likely not hold for larger areas. In practice, the field would be unknown, so there is no way of compensating for an irregular radiation pattern.

- The RF contour and signal decay vs distance plots for the data collected are very different from the theoretical FSPL model predictions. The results show good agreement close to the transmitter and closely match the slope of the power decay with distance line. However, they are very noisy, and the contour shows an irregular pattern with a very subtle trend. This is partly due to the simplicity of the model, which fails to capture many real-world effects like ground and surrounding reflections and fading, but also likely means there are a lot of improvements to be made to the data collection and processing systems, to reach an acceptable industry standard.

- While the measurements recorded have improved significantly with the antenna on top, the navigation is considerably worse due to the LiDAR being blocked by the stand-offs supporting the extra layer, as the Figures in section 4.3 showed. This suggests an important trade-off between the navigation and RF measurement subsystems for the project in its current state. Solutions to this might lie in different, untested antenna arrangements and/or an upgraded version of the controller, which uses more advanced techniques.

- Testing in simulation is a good way to assess whether the controller and navigation algorithms are effective, but in no way guarantees the same level of performance in the real world, as exemplified by the results obtained. It is a valuable stage in the development pipeline, specifically because testing is straightforward and quick, but the testing environment presents many limitations compared to the real world.

- Although battery life is quoted at 2.5 hrs by the manufacturer, testing shows that this number is closer to 1 – 1.5 hrs. This could be due to battery deterioration as a result of extended usage, or because of the additional demands placed by the RF measurement system.

- RTL-SDR bandwidth was experimentally determined as 24 – 1700 MHz from testing in the RFID lab with a signal generator, which lies inside the acceptable requirement range.

- Because the LiDAR uses reflected laser light, the whole system can run in the dark, making it suitable to be used in the commercial applications listed in the motivation section, as it could be deployed autonomously after hours, without disturbing normal operation during the day.

These conclusions reflect that many of the criteria have been met or even exceeded, but a lot of work remains to be done in order to turn this project into a commercially viable solution. The next section looks at some potential next steps that could be taken to reach a truly successful system.

6.2 Future Work

Here we will look at further ideas that were planned to be developed, but had to be forgone due to lack of time, and other logical research pathways that could be explored following on from the experiments that have been carried out.

- Relax the constraints placed on the STC algorithm, and design a program capable of working in dynamic environments, as well as a version that can run online, and compare the performance against each other. This would be useful for unknown environments.
- Upgrade the robot controller into a PID controller by finding a way to tune each of the gains, to achieve better path tracking, eliminating oscillations and steady-state errors.
- Investigate the use of more advanced controllers using novel techniques like Linear Quadratic Regulators (LQR), H-infinity control or Model Predictive Control (MPC) as the basis for navigation [35].
- Simulation of RF field strength according to further models (e.g Two ray model [45]) or adding Rician fading [46], and comparing whether they are a better match for the data already collected.
- Factor in runtime estimates at the CPP trajectory generation stage, to roughly calculate how long a run will take. In addition, it would also be useful to find a maximum range based on battery capacity, letting users know whether a given area is too large to be covered on one battery charge.
- Investigate the use of several TurtleBot robots to work simultaneously to cover a given area, and integrate the measurements at the end.
- Test the SLAM approach for map generation on different surfaces to see whether they make an impact in the quality of the map file. This could place a limit on the kind of environments where this solution can be deployed.
- Explore different antenna configurations, in particular the use of a perspex cylinder to support the antenna layer, as it is transparent, and doesn't block the LiDAR, hence combining the best of both worlds in terms of navigation and RF measurement.
- Investigate sensor fusion by combining LiDAR data and odometry to obtain a more accurate pose estimate of the robot in the localization stage.
- Test the system in larger, more cluttered environments and see how well it responds, and if there is a point where it breaks.

Bibliography

[1] Oliver Woodman and Robert Harle. Pedestrian localisation for indoor environments. page 114. ACM Press, 2008.

[2] Dr Yu-Han Chang Raghu Das and Dr Matthew Dyson. RFID Forecasts, Players and Opportunities 2022-2032. *IDTechEx*, 2021.

[3] Sophia Anitpolis. Radio Frequency Identification Equipment operating in the band 865 MHz to 868 MHz with power levels up to 2 W and in the band 915 MHz to 921 MHz with power levels up to 4 W; Harmonised Standard for access to radio spectrum, 2018.

[4] David Swarbrick. Robotic RF Field Measurement. Master's thesis, University of Cambridge, 2020.

[5] Sebastian Thrun. *Probabilistic robotics / Sebastian Thrun, Wolfram Burgard, Dieter Fox.* Intelligent Robotics and Autonomous Agents Ser. MIT, Cambridge, Mass. ; London, 2005.

[6] D. Hahnel, W. Burgard, D. Fox, K. Fishkin, and M. Philipose. Mapping and localization with RFID technology. pages 1015–1020 Vol.1. IEEE, 2004.

[7] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Monte carlo localization for mobile robots. In *Proceedings 1999 IEEE international conference on robotics and automation (Cat. No. 99CH36288C)*, volume 2, pages 1322–1328. IEEE, 1999.

[8] Denis Chikurtev, Nayden Chivarov, Stefan Chivarov, and Ava Chikurteva. Mobile robot localization and navigation using LIDAR and indoor GPS. *IFAC-PapersOnLine*, 54:351–356, 2021.

[9] Xiaoran Fan, Han Ding, Sugang Li, Michael Sanzari, Yanyong Zhang, Wade Trappe, Zhu Han, and Richard E. Howard. Energy-Ball. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2:1–22, 7 2018.

[10] Quentin Bonnard, Séverin Lemaignan, Guillaume Zufferey, Andrea Mazzei, Sébastien Cuendet, Nan Li, Ayberk Özgür, and Pierre Dillenbourg. Chilitags 2: Robust Fiducial Markers for Augmented Reality and Robotics., 2013.

[11] Alejandro Gonzalez-Ruiz, Alireza Ghaffarkhah, and Yasamin Mostofi. A comprehensive overview and characterization of wireless channels for networked robotic and control systems. *Journal of Robotics*, 2011, 2011.

[12] Wen-Tzu Chen and Chen-Hsun Ho. Spectrum monitoring with unmanned aerial vehicle carrying a receiver based on the core technology of cognitive radio–A software-defined radio design. *Journal of Unmanned Vehicle Systems*, 5(1):1–12, 2016.

[13] Steven M Lavalle. *PLANNING ALGORITHMS*. 2006.

[14] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The International Journal of Robotics Research*, 30:846–894, 6 2011.

[15] Alejandro Perez, Robert Platt, George Konidaris, Leslie Kaelbling, and Tomas Lozano-Perez. LQR-RRT*: Optimal sampling-based motion planning with automatically derived extension heuristics. In *2012 IEEE International Conference on Robotics and Automation*, pages 2537–2542. IEEE, 2012.

[16] Jerome Barraquand and Jean-Claude Latombe. Robot motion planning: A distributed representation approach. *The International Journal of Robotics Research*, 10(6):628–649, 1991.

[17] Lanzhou Jiaotong University, Institute of Electrical, and Electronics Engineers. *A Research on Area Coverage Algorithm for Robotics IRCE 2018 : 2018 IEEE International Conference of Intelligent Robotic and Control Engineering : August 24-27, 2018, Lanzhou, China.*

[18] Hai Van Pham, Philip Moore, and Dinh Xuan Truong. Proposed Smooth-STC Algorithm for Enhanced Coverage Path Planning Performance in Mobile Robot Applications. *Robotics*, 8:44, 6 2019.

[19] Qile He and Yu Sun. An Optimized Cleaning Robot Path Generation and Execution System using Cellular Representation of Workspace. pages 13–25. Academy and Industry Research Collaboration Center (AIRCC), 11 2020.

[20] Stanford Artificial Intelligence Laboratory et al. Robotic Operating System (ROS). `https://www.ros.org`, 2018-05-23.

[21] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. `https://gazebosim.org/home`.

[22] Amazon Web Services Educate. AWS Robomaker Badge series. `https://aws.amazon.com/robomaker/`, 2022.

[23] Wikipedia contributors. Robot operating system — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Robot_Operating_System&oldid=1081050080`, 2022. [Online; accessed 18-May-2022].

[24] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[25] I. ROBOTIS. PLATFORM - Turtlebot 3 - ROBOTIS. `https://www.robotis.us/turtlebot-3/`, 2022.

[26] H.L Data Storage. LDS1.5 SPECIFICATIONS MODEL : HLS-LFCD2. `https://emanual.robotis.com/assets/docs/LDS_Basic_Specification.pdf`, 2014.

[27] I. ROBOTIS. Turtlebot 3 Robotis e-Manual Setup,. `https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/`, 2022.

[28] ROBOTIS-GIT. ROS packages for Turtlebot3,. `https://github.com/ROBOTIS-GIT/turtlebot3/blob/master/turtlebot3_example/nodes/turtlebot3_pointop_key`, 2022.

[29] Luis Bustillo Ortiz. IIB Project Repository. `https://github.com/LuisBustillo/IIB_Project`, 2022.

[30] Roger. ROS packages for Turtlebot3. `https://github.com/roger-/pyrtlsdr`, 2013.

[31] Carl Laufer. THE HOBBYIST'S GUIDE TO RTL-SDR:A GUIDE TO RTL-SDR AND CHEAP SOFTWARE DEFINED RADIO BY THE AUTHORS OF THE RTL-SDR.COM BLOG.

[32] Wikipedia contributors. Near and far field — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Near_and_far_field&oldid=1086282425`, 2022. [Online; accessed 20-May-2022].

[33] Wikipedia contributors. Travelling salesman problem — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1064222176`, 2022. [Online; accessed 21-May-2022].

[34] Wikipedia contributors. Depth-first search — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Depth-first_search&oldid=1087546220`, 2022. [Online; accessed 21-May-2022].

[35] Atsushi Sakai, Daniel Ingram, Joseph Dinius, Karan Chawla, Antonin Raffin, Alexis Paques, and Alexis@unmanned Life. PythonRobotics: a Python code collection of robotics algorithms ENSTA ParisTech. `https://atsushisakai.github.io/`, 2018.

[36] 18alantom. Coverage Path Planning. `https://github.com/18alantom/CoveragePathPlanning/tree/master/cpp_algorithms/coverage_path/stc`, 2020.

[37] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[38] Shoudong Huang and Gamini Dissanayake. *Robot Localization: An Introduction*. John Wiley Sons, Inc., 8 2016.

[39] Jackack. Turtlebot3 Complete Coverage. `https://github.com/Jackack/Turtlebot3_complete_coverage`, 2020.

[40] Kaiyu Zheng September. ROS Navigation Tuning Guide.

[41] Wikipedia contributors. Pid controller — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=PID_controller&oldid=1080202662`, 2022. [Online; accessed 17-May-2022].

[42] George Ellis. *Control system design guide: using your computer to understand and diagnose feedback controllers*. Butterworth-Heinemann, 2012.

[43] Dr Marc Lichtman. PySDR: A Guide to SDR and DSP using Python. `https://pysdr.org/`, 2018.

[44] Wikipedia contributors. Free-space path loss — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Free-space_path_loss&oldid=1080664589`, 2022. [Online; accessed 17-May-2022].

[45] Wikipedia contributors. Two-ray ground-reflection model — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Two-ray_ground-reflection_model&oldid=1071641723`, 2022. [Online; accessed 17-May-2022].

[46] Wikipedia contributors. Rician fading — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Rician_fading&oldid=1083787801`, 2022. [Online; accessed 17-May-2022].

[47] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.

Risk Assessment Retrospective

At the outset of the project, a risk assessment was conducted to evaluate possible risks of the experimental activities associated with this project.

As the project is mainly computational in nature, very few risks were identified. Those included prolonged computer usage, and side effects like bad posture, eyesight strain and headaches, and tripping over loose wires. To prevent these potential health risks, several measures were taken, like sitting correctly, staying hydrated, taking regular breaks, and switching the device into night mode to lower the brightness, as well as following the 20-20-20 rule which consists in focusing on an object 20 feet away, for 20 seconds every 20 minutes, to prevent eye strain.

Even though a robotic platform was at the centre of the practical work, there were no real robotic risks, as the TurtleBot robot is quite small, has few moving parts (no manipulators) and was to be operated at low translational and rotational speeds. The biggest hazard posed by the robot, is the Li-ion rechargeable battery. To mitigate this, battery charging guidelines were followed – only charging the battery when present and under active supervision, and never charging/unplugging when the robot is powered on.

In retrospect, the assessment of the project as being low risk was accurate, with minimal steps needed to be taken to guarantee its completion safely and securely, for everyone involved.

Extra detail on ROS Framework

ROS processes are represented as nodes in a graph structure, connected by edges called topics. ROS is structured as a publisher/subscriber paradigm, which allows processes on either the Host PC or SBC to be defined as 'subscribers' or 'publishers' of pre-defined message structures [23]. These messages are made available to other processes on each machine, and over the network connection between them. ROS nodes can pass messages to one another through topics, make service calls to other nodes, provide a service for other nodes, or set or retrieve shared data from a communal database called the parameter server. A process called the ROS Master makes all of this possible by registering nodes to itself, setting up node-to-node communication for topics, and controlling parameter server updates.

The implementation of SLAM and Odometry used in the project are realized through the **gmapping** and **navigation** ROS Packages written in Python. The python programs executing navigation and RF field measurement are represented as nodes. A graph of the nodes and topics is shown in Figure B.1. ROS also offers a wide variety of tools which were at the centre of running the system, including:

- *rviz*: a three-dimensional visualizer used to visualize robots, the environments they work in, and sensor data.
- *catkin*: system used to build packages in ROS.
- *roslaunch*: tool used to launch multiple ROS nodes both locally and remotely, as well as setting parameters on the ROS parameter server.

This abstraction of the communication protocol allows quick development of programs which can be run on either computer, depending on desired processing/latency considerations. This project only scratches the surface in terms of what is possible to achieve through ROS, and such an understanding of the framework was possible by completion of the AWS Robomaker badge series [22] and Michaelmas part IIB module 4M20 *Introduction to Robotics* lectured by Dr Amanda Prorok.

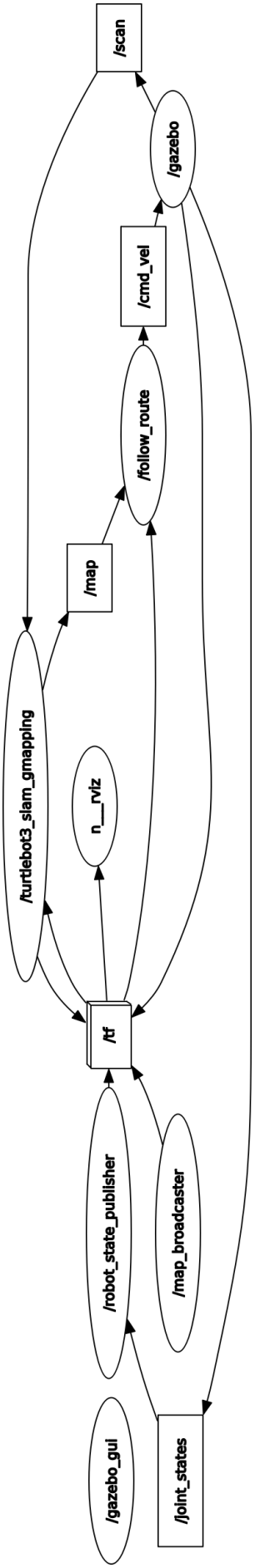


Fig. B.1 ROS computation graph showing the running nodes and topics, as well as the communication between them, for the system designed.

User Manual

Prerequisites

The programs needed to run the various systems in this project require the use of ROS Noetic (`www.ros.org`), Gazebo (`http://gazebosim.org/`) and Python. You will need to have a laptop / workstation / bootable USB drive / virtual machine [1] that runs Ubuntu, which can be installed following a step-by-step guide [2]. Install ROS Noetic and Gazebo locally by following this tutorial [3]. The following instructions might work on older versions of the above software, but the results presented in this report were produced using the newest, most stable versions, which at the time of writing this guide are, Ubuntu 20.04, ROS Noetic and Gazebo v11.

Python is the main programming language used for the files in this project. On Ubuntu, you can install Python (including the NumPy and SciPy modules [47]) as follows:

```
$ sudo apt install python3 python3-pip
$ pip3 install numpy scipy
```

The next step is to install ROS dependent packages [4] and create a workspace for the project, by opening a terminal and running the following commands:

```
$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
    ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc  ros-noetic-rgbd-launch \
    ros-noetic-depthimage-to-laserscan ros-noetic-rosserial-arduino \
    ros-noetic-rosserial-python ros-noetic-rosserial-server ros-noetic-rosserial-client \
    ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server ros-noetic-move-base \
    ros-noetic-urdf ros-noetic-xacro ros-noetic-compressed-image-transport \
    ros-noetic-rqt-image-view ros-noetic-gmapping ros-noetic-navigation \
    ros-noetic-interactive-markers python3-catkin-tools
$ source ~/.bashrc
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin build
```

[1] `https://github.com/sutd-robotics/virtualbox-ubuntu-ros`
[2] `https://www.ubuntu.com/download/desktop/install-ubuntu-desktop`
[3] `http://wiki.ros.org/noetic/Installation/Ubuntu`
[4] `https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/#install-dependent-ros-packages`

The final step to set up the environment, is to clone the GitHub repository for this project in the *catkin* workspace, and build the packages:

```
$ cd ~/catkin_ws/src/
$ git clone https://github.com/LuisBustillo/IIB_Project.git
$ cd ..
$ catkin_make
```

Mapping

As mapping wasn't strictly a part of the development focus of this project, the base implementation provided by ROBOTIS was employed for this step [5].

Path Planning

To run the CPP algorithm, navigate to the *python* directory with the file *cpp.py*, ensuring that the *map.pgm* and *map.yaml* files are saved in the same directory, before running the file in a terminal:

```
$ cd ~/catkin_ws/src/.../python
$ python3 cpp.py
```

This produces a figure showing the route overlaid on top of the map, and a file, *route.yaml*, containing a list of the sampling points in the path. The sample spacing can be edited in the *cpp* function parameters inside *cpp.py*.

Simulation Testing

For this type of testing, a physical TurtleBot robot is not required. The first step is to open a world map in Gazebo simulator. The following example uses a map created by ROBOTIS [28], pictured in Figure 3.6 to illustrate this process.

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

The next step is to run the navigation program. This involves navigating to the *ros* directory with the python file *follow_the_route.py*, and running this file in a new terminal:

```
$ cd ~/catkin_ws/src/.../ros
$ python3 follow_the_route.py
```

[5] https://emanual.robotis.com/docs/en/platform/turtlebot3/slam/#map

Real-world Testing

This requires you to have a physical TurtleBot3 burger robot. First, ensure the machine running Ubuntu, and the robot are connected to the same network (e.g by setting up a Wi-Fi hotspot) and run the following command in a terminal:

```
$ export TURTLEBOT3_MODEL=burger
$ hostname -I # Make a note of the output (e.g., 192.168.225.137).
$ ROS_MASTER_URI=http://192.168.225.137:11311 # Terminal 1
$ ROS_HOSTNAME=192.168.225.137 # Terminal 1
$ roscore # Terminal 1
```

Turn on your robot, and connect to it by running the following commands in a new terminal:

```
$ ROBOT_IP=192.168.225.210 # Use your value
$ scp pi_sdr.py pi@192.168.225.210:~ # Default password is turtlebot
$ ssh pi@192.168.225.210
$ export TURTLEBOT3_MODEL=burger
$ ROS_MASTER_URI=http://192.168.225.137:11311 # Robot terminal 1
$ ROS_HOSTNAME=192.168.225.210 # Robot terminal 1
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch # Robot terminal 1
```

Open another terminal on the computer, and connect with the TurtleBot to establish a connection with the SDR by running:

```
$ ssh pi@192.168.225.210
$ export TURTLEBOT3_MODEL=burger
$ ROS_MASTER_URI=http://192.168.225.137:11311 # Robot terminal 2
$ ROS_HOSTNAME=192.168.225.210 # Robot terminal 2
$ python pi_sdr.py # Robot terminal 2
```

Finally, on the laptop again, run the following in two additional terminals:

```
$ ROS_MASTER_URI=http://192.168.225.137:11311 # Terminal 2 & 3
$ ROS_HOSTNAME=192.168.225.137 # Terminal 2 & 3
$ export TURTLEBOT3_MODEL=burger # Terminal 2 & 3
$ roslaunch turtlebot3_slam turtlebot3_slam.launch # Terminal 2.
$ cd   ~/catkin_ws/src/.../ros # Terminal 3
$ python3 follow_the_route_rf.py # Terminal 3
```

When the robot completes the route, it will stop and two files, *path.yaml* and *data.yaml*, will be generated. If you wish to test just the path tracking algorithm, run *follow_the_route.py* instead, as per the simulation case.

RF measurement results

This section is only suitable for results obtained in a real-world test. Make sure you edit the python files in this section, to reflect your specific file paths for the *path.yaml* and *data.yaml* files, in the relevant functions. To visualize the real trajectory and RF results collected during the experiment, run the following command:

```
$ cd ~/catkin_ws/src/.../python
$ python3 heatmap.py
```

To visualize the simulated RF results following a theoretical model, run the following command, after editing the file to select which model to use:

```
$ cd ~/catkin_ws/src/.../python
$ python3 heatmap_simulation.py
```