Using SLAM with LEGO Mindstorms to Explore and Map an Environment

Samuel Brown

Artificial Intelligence

Session 2013/2014

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)_____

Summary

The primary aims of this project are to design an build a robot using Lego's Mindstorms NXT system, and to develop an appropriate implementation of a Simultaneous Localization And Mapping (SLAM) technique for the robot. This also required a suitable testing environment to be designed and constructed. Once The algorithm had been implemented, various methods of displaying and processing the data were explored.

Acknowledgements

Firstly, I would like to thank my project supervisor, Professor Tony Cohn, for his support and advice throughout this project. I would also like to thank Dr Andy Bulpitt for the loan of the equipment used, and for his feedback of my Mid-Project Report. Finally, I would like to thank my friends and family for their continued support throughout my time at Leeds.

Contents

Chapter 1

Introduction

1.1 Aim

The aim of this project is to design and implement a system to detect and evaluate changes in an environment using simultaneous localisation and mapping (commonly referred to as SLAM). The hardware of this system, constructed using a LEGO Mindstorms NXT kit, will comprise a mobile robot equipped with sensors for object detection and distance measurement.

1.2 Objectives

The objectives for the project are:

- Design and build a suitable hardware platform using the Mindstorms NXT system.
- Implement an appropriate mapping algorithm to be performed by the robot.
- Implement a method of displaying results of the algorithm and comparing different maps.

1.3 Requirements

The minimum requirements for the project are:

- Construct a robot using the NXT system that can navigate a simple environment and detect the edges (walls) of the environment.
- Construct an environment suitable for testing and evaluating the robot in.
- Run a simple SLAM implementation on the robot to collect data about its environment.
- Explore methods of processing, displaying and comparing collected data.

1.4 Methodology

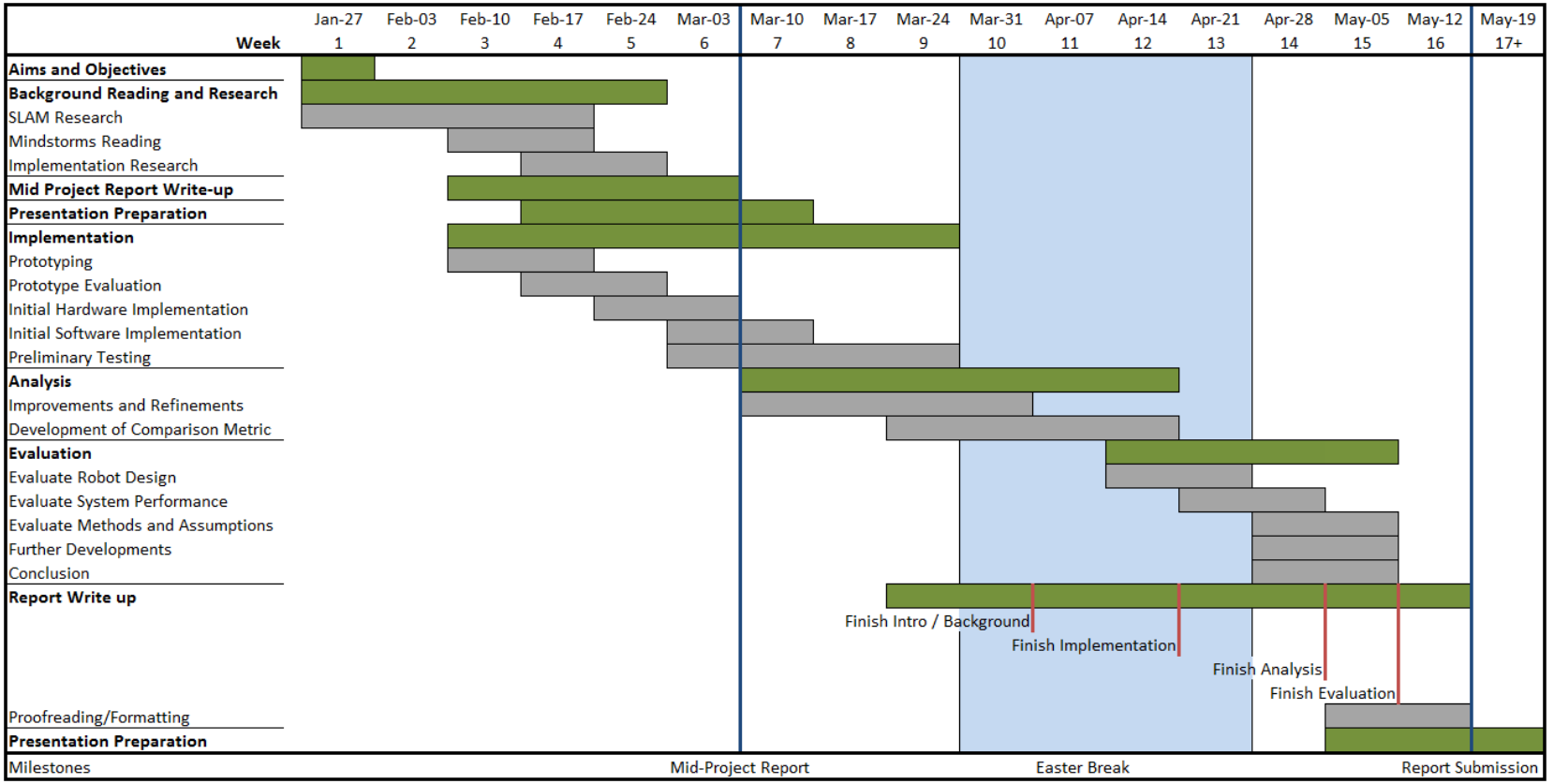
I will be employing primarily iterative methodologies during the course of this project. For the development of the hardware, I shall follow an iterative methodology, with each iteration of the hardware correcting flaws in the previous iteration. As developing new hardware mostly involves dismantling previous work to make even small improvements, I plan on making two or three prototypes before the 'final' hardware (which may be tweaked during software development/testing).

Software development will mostly fall into two categories: the robot's on-board software, and the PC-based application. I intend to employ a modular approach for the robot software, developing and testing various modular behaviours (movement, scanning &c.) separately before combining them. For the PC application, I plan to follow an iterative approach, writing and improving functionalities in stages. Once a command line application has been created and debugged, I will then create a graphical user interface (GUI) to display the map and localisation.

1.5 Schedule

Below is the initial schedule for the project (Figure 1.1).

Figure 1.1: Initial Project Schedule



Chapter 2

Background Reading

2.1 An Introduction to Lego Mindstorms

The Lego Mindstorms NXT kit is the successor to Lego's first consumer targeted robotics kit, the Mindstorms RCX (Robotic Command eXplorer), released in 1998. The RCX consisted of a single programmable 'brick', and several sensors and motors that could be connected to the brick. Programs for the kit would be written on a computer, either in Lego's own proprietary software or through a number of third party libraries for common programming languages. These programs could be download to the brick over an IR connection and then run. However, due to the size and power requirements of the brick, the RCX was better suited to static applications, such as robotic arms.

Released in 2006, the Mindstorms NXT system uses the same paradigm of a single control brick and separate sensors. The kit was released with instructions for several projects, and included the NXT-G programming software. As with the RXC, many third party interface libraries have been released, and some will be discussed in Section 2.1.2.

2.1.1 Breakdown of the kit

The NXT kit comes with a number of modular components connected to the brick using the I^2C protocol [9], allowing third parties to develop more and accurate sensors than those available from Lego. Below are descriptions of the components I will be using in this project.

Figure 2.1: The NXT Brick.

Servo Motors

The servo motors are both sensors and actuators. As well as providing rotational movement, they can actively resist movement and have integral speed and rotation sensors. The rotation sensor is accurate to within $\pm$ 1 degree, and can measure full rotations [17].

Ultrasonic Sensor

The ultrasonic sensor is being employed here as a distance sensor. It has a stated range of $0-255cm$ with $\pm 3cm$ accuracy [18], which should be more than adequate for this project. However, it has been noted that the ultrasonic readings can be affected by many factors, including the nature of the surface the ultrasonic waves are reflected off, and the angle of incidence of the wave relative to the surface [9]. This raises a number of points to consider when designing the testing environments. In a previous Final Year Project on using the NXT System, it was ascertained that covering the environment walls in aluminium foil provided a superior accuracy in measurements [7].

NXT Brick

The Brick is the programmable core of the NXT System. It houses a 32-bit microprocessor, 256Kb of flash memory, a USB interface and inbuilt Bluetooth connectivity [16]. The brick has four ports for sensor connections, and three ports dedicated to servo motors. Both the USB or Bluetooth connections can be used to connect the brick to a computer either for downloading programs or for real time communication.

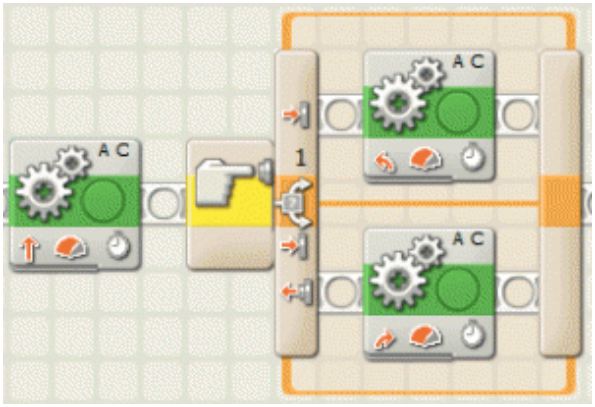


Figure 2.2: Example of a switch statement 'code block' from NXT-G.

2.1.2 Programming the NXT System

In addition to the provided NXT-G software, many third party libraries for programming the NXT have been developed, allowing support for more complex applications in a range of languages.

NXT-G

The NXT-G software is provided by Lego and written in LabVIEW. NXT-G provides a graphical programming environment designed for children and non-technical users of the Mindstorms kit. Its drag and drop 'code blocks' allow for easy development (as shown in Figure 2.2), but significantly limit the complexity of produced programs.

LeJOS

LeJOS is perhaps the most widely used third-party library for the NXT, even having been used by NASA on the International Space Station [10]. Based on Java, it runs a Java Virtual Machine (VM) on the NXT Brick. LeJOS can provide a high level abstraction, allowing developers to work above low level details such as addresses of sensors and the binary signals used to communicate with them. LeJOS also natively supports many third party sensors [6]. An important factor to consider however is that because LeJOS runs on a Java VM, the NXT Brick's native firmware must be overwritten to allow LeJOS applications to run. This prevents code from other sources being used, meaning that using multiple languages for testing or comparison is more time consuming, as the firmware must be swapped between trials.

Mindstorms NXT Toolbox for MATLAB

The Mindstorms NXT Toolbox for MATLAB was developed at RWTH Aachen University, originally starting as a project by a group of electrical engineering students [14]. Like LeJOS, the toolkit provides a simple object-orientated access to peripherals connected to the Brick. As MATLAB is an interpreted language, the RWTH toolbox allows for real-time control of the NXT system over Bluetooth or USB. MATLAB has many inbuilt methods for signal and image processing, in addition to its native visualisation libraries. This makes the RWTH Toolbox highly popular. However MATLAB is a commercial product, and as such many individuals who are not affiliated to an academic or research institution are unlikely to have access to it.

Not eXactly C

Not eXactly C (NXC) is a high-level language with syntax based on C. It is compiled into Next Byte Codes (NBC) , which can run natively on the NXT Brick. NXC is the successor to Not Quite C (NQC), a similar language developed for the previous RCX system [5]. Like LeJOS, programs must be compiled and transfered to the Brick before running. However, as NXC doesn't require the firmware to be flashed, it is possible to develop applications using NXC and the NXT-G software (or other libraries) side by side.

2.2 Architectures for Robot Control

Whilst robotics is a large area with an ever-increasing number of applications, many robots can be classified as following one of the common paradigms for robotic control.

2.2.1 Sense-Plan-Act

Perhaps the oldest defined methodology is Sense-Plan-Act. "Shakey", a robot developed in the late 1960's by the then Stanford Research Institute (now SRI International) is claimed to be the first mobile robot with the ability to perceive and reason about its surroundings. Its software was built upon the Sense-Plan-Act idiom [15]. The three logical stages (primitives) are:

SENSE - Gather raw information about the environment from sensors

PLAN - Process the raw information, generate a world model and plan action required to move closer to goal state

ACT - Perform the action

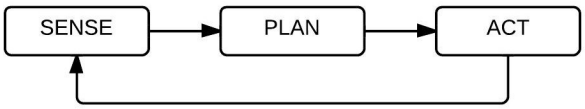


Figure 2.3: The Deliberative Paradigm.

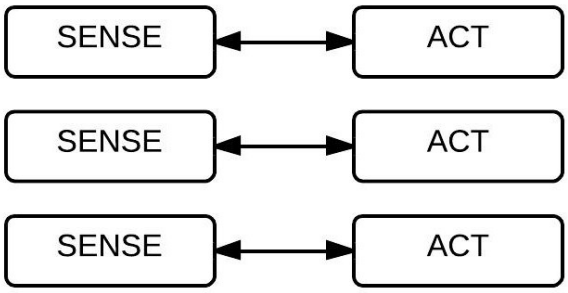


Figure 2.4: The Reactive Paradigm.

From these three primitives, a number of different schemas have arisen. The original implementation of Sense-Plan-Act was through the deliberative paradigm. This involved the three stages in a repeating cycle (see Figure 2.3). The main issue with this architecture, especially in the early days of robotics, was that the Plan stage was normally very computationally expensive, taking a long time to process. This caused delay in acting, and meant that robots using deliberative processing were highly unresponsive. In a dynamic world rather than a static environment, a time delay could be significantly problematic.

To improve performance, the reactive paradigm was developed, cutting out the Plan element in favour of many independent parallel Sense-Act couplings (Figure 2.4). Whilst this drastically speeds up the operation of the robot, all actions are now reactions to the environment rather than a reasoned sequence with forward planning. This can lead to the robot being less able to perform certain tasks as efficiently or at all.

2.2.2 An Alternative Approach

In 1986 Brooks introduced the Subsumption Architecture [1]. Building upon the ideas of the reactive paradigm, subsumption employs a hierarchy of behaviours to accomplish tasks. More rudimentary, low level behaviours are subsumed by more complex behaviours higher in the hierarchy either through inhibition or suppression. Inhibited behaviours are actively suspended, suppressed behaviours are permitted to continue but may be overridden higher behaviours that affect the same outputs. The subsuption architecture is convenient for iterative, bottom up development, allowing the most basic underlying behaviours to be implemented and thoroughly tested before more complex behaviours are attempted.

2.3 An Introduction to SLAM

2.3.1 What is SLAM?

Simultaneous Localization and Mapping (SLAM) is concerned with building an accurate map of an unseen environment, and simultaneously using that map to navigate through the environment. SLAM is not a single algorithm, but rather a family of concepts describing solutions to the same problem.

SLAM is often characterised as a ”chicken and egg” problem [3]:

- An accurate map is needed in order to localise the agent.
- The agent needs to know its precise location to build an accurate map.

SLAM can be broken down into a number of key areas; Landmark extraction, data association, state estimation, state update and landmark update. Each of these stages can be approached in a number ways, allowing different implementations of SLAM to vary considerably [13].

Thus, SLAM is not a straightforward problem. It is one of the biggest areas in robotics research, with many decades spent refining and solving the many aspects of SLAM.

2.3.2 A Brief Outline of Maps

Humans and other animals perform the localisation side of the task almost subconsciously in familiar environments, memorising landmarks and using spacial awareness to situate themselves. Ancient civilisations such as the early Polynesians and the Classical Greeks used stars and the night sky as environmental landmarks for navigation without maps (i.e. localisation) on a much larger scale. In spite of this seemingly innate ability to self-situate, humans struggle with creating accurate maps without the benefit of exact measurement.

Older hand drawn maps often correctly display notable geographic features (such as bays or rivers), but the incorrect judgement of distances can cause distortion to the map. Figure 2.5 shows this distortion in action. This map was compiled in 1513 by Piri Reis, an Ottoman admiral and cartographer. Whilst the map unmistakably matches the outline of Europe, areas that were less familiar and hadn’t been measured (such as the north European coast) are noticeably disfigured. This deformation may cause some problems for human navigation, but there are many fall-back measures we can use that allow us to adapt misreported information and mentally correct it. However, if a computerised system was given such a map to navigate by, it would require very complex heuristics to adapt such a map, and may still fail. As such, accurate maps are crucial for computer driven navigation.

Figure 2.5: The Piri Reis Map of Europe [12].

2.3.3 Mapping using Mathematics

Around the turn of the 19th Century, mathematicians such as Laplace were using statistical methods such as least-squares to calculate increasingly accurate measures of the earth's surface, reducing errors in geodesy and cartography [11].

Even though the advent of aerial and then satellite photography has had a massive effect on mapping, the statistical methods still play an important part in situations where photography based methods are unsuitable or impractical. As many of the sensors available for robotics today have some level of error, SLAM implementations often use probabilistic methods to determine measurements, either to landmarks or an estimate of the robot's pose (position).

Probabilistic methods were only introduced to the SLAM Problem at the 1986 IEEE Robotics and Automation Conference. In the years following of recognition that statistical methods could be vital for solving SLAM, many papers investigating these techniques were released, and the most common model for SLAM (recording robot pose and landmark location upon recognition of a landmark) was developed.

Initially it was believed that errors in landmark location relative to robot pose and other landmarks would increase over time, and so researchers made assumptions trying to eliminate correlation between landmarks. The purpose of this was to attempt reduce the complexity of the mapping problem by considering only the relationships between landmarks and the robot [2]. This lead to mapping and localisation being considered as separate problems. A key breakthrough in the field came when it was realised that treating mapping and localisation as a single estimation problem caused it to become convergent. The breakthrough was presented in 1995, along with a formal definition of the SLAM problem, and the acronym SLAM at the International Symposium on Robotics Research [4].

Kalman Filters and the Extended Kalman Filter

The Kalman Filter, published in 1960 by Rudolf Kálmán, is an algorithm used to extract a signal from a sequence of incomplete, inaccurate and/or noisy data (such as may be gathered from sensors). One of the first notable applications of the filter was in the trajectory estimation software used by the Apollo Program [8].

Kalman filters provide optimal state estimation for linear models, but unfortunately most of the systems used in real-life scenarios are non-linear. Consequently, the Extended Kalman Filter (EKF) was developed at the NASA Ames Research centre. It has since become one of the most widely used statistical methods in SLAM.

2.3.4 Issues for SLAM

Whilst SLAM is generally considered to be a ”solved” problem, there are many important considerations to be made before developing an implementation of SLAM. Portability of the system can be an issue with SLAM. The key factor is the degree of hardware dependency; in a system utilising SLAM the software is usually heavily tailored to the specific hardware platform. The hardware, in turn, will likely be designed with a specific environment in mind; a system designed for a controlled indoor locale (such as the system to be implemented in this project) is unlikely to have much success in an open outdoor environment.

In larger implementations of SLAM, the scalability of the system must be taken into account. When using techniques such as Kalman filters, the complexity of updating the internal representation is $O(n^2)$, where n is the number of features recorded. Given a large enough environment with many landmark features to trigger updates, this will prevent real-time processing [2].

Chapter 3

Hardware Design and Implementation

3.1 Limitations and Considerations

Just as with software design, there are certain constraints involved with building hardware. Whilst Lego is a versatile building material, the kit has a limited number of pieces, restricting possible designs. The Brick must be in a position where it can be removed without major dismantlement to allow the batteries to be replaced. Similarly, there must be sufficient clearance at ether end of the brick for the cables attaching the motors and sensors to attach.

There are a number of different sensors for the kit that could be used, but the ultrasonic sensor was deemed the more suitable over other options such as touch sensors or third party range finding sensors for two main reasons. Firstly, the achievable level of accuracy in the robot's odometry was thought to be insufficient to use a tactile approach, indicating a range-based approach should be used. The other deciding factor was that even thought there are a number of third-party range-finders on the market with superior accuracy, the ultrasonic sensor was immediately available to begin work with. Choosing an alternative and waiting for it to be shipped could take two to three weeks, which isn't a feasible delay in this project's time scale.

Another restricting factor is the number of motors that can be used. Three motors are provided, and no more than that can be supported by the Brick. In order to steer the robot, there are two possible configurations; a rack and pinion mechanism (used in cars and most other wheeled vehicles) or differential steering (typically employed in tracked vehicles). For this project it was judged that a rack and pinion construction would be needlessly complex and would not produce the zero-displacement

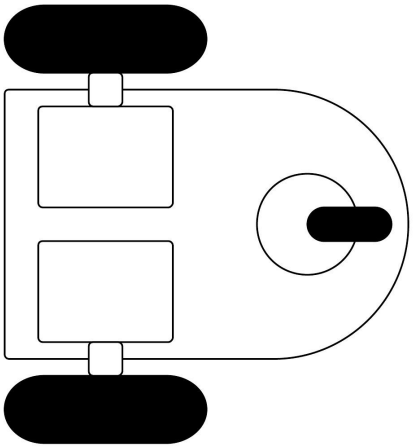


Figure 3.1: Wheel layout of Prototype 1.

turning required for the robot to rotate on the spot.

Differential steering requires two wheels (or tracks) on either side of the robot to be powered independently, meaning the robot turns by the two wheels moving at different rates. Using differential steering means that there are a certain other restrictions for the chassis. If a four wheel or track based layout is used, the un-powered wheels can't be attached to the same axle; two separate axles on the same axis of rotation are required. If a three wheeled approach is used, the third wheel needs to be able to rotate freely and act as a caster.

3.2 Prototypes and Final Design

The initial design of a piece of hardware will inevitably exhibit flaws and shortcomings. For this reason, the first two or three iterations of hardware are designated as prototypes, with each correcting problems highlighted by the previous generation.

Prototype 1 was constructed using a three wheeled configuration, with two independently powered driving wheels and a freely rotating caster (see Figure 3.1). Whilst this design is frequently used in robotics, the implementation attempted was not very successful. As the majority of the weight of the robot was mounted slightly in front of the caster, the pivot allowing the wheel to rotate freely was under some strain and didn't rotate as desired, dragging perpendicularly along the floor surface rather than turning as intended. Consequently, the use of tracks rather than wheels was adopted.

Stability issues were highlighted in Prototype 2; The NXT Brick (which is by far the single heaviest component in the kit, due to the 6 AA batteries) was mounted directly above the rear axle, perpendicularly to the floor (see Fig 3.2). When the robot accelerated, the weight on the rear caused the robot

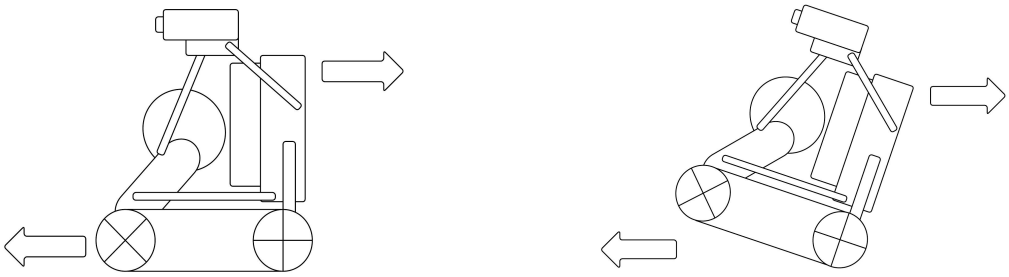


Figure 3.2: Stability issues in Prototype 2.

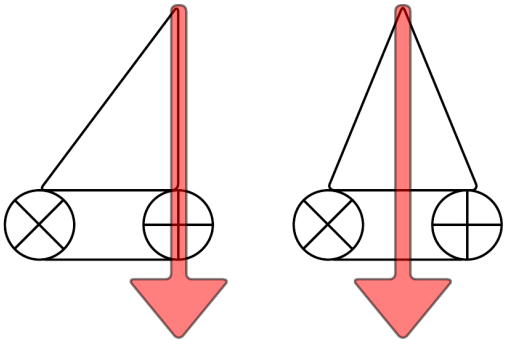


Figure 3.3: Center of Gravity in Prototype 2 vs Final Design.

to begin to tip. On a four wheeled solution this would have lifted the driving wheels out of contact, causing the robot to decelerate and self-stabilise. However, because both 'wheels' were linked by the tracks, the robot continued to move forward and would then fall backwards. This issue wasn't apparent in the initial prototype, as the caster was mounted behind the center of gravity.

The NXT Brick obviously couldn't be removed, and the movement system otherwise worked as required. Rather than completely redesigning the chassis, I altered the angle of the motors to allow the Brick to 'lean back' into the wheelbase. The additional motor (for the Ultrasonic sensor, discussed below) at the front of the robot also helped by increasing the weight above the front axle. Shifting the center of gravity forward was sufficient to almost entirely eliminate the tipping problem (see Fig 3.3). The robot does still tip when on an incline or when colliding with surfaces too short to be detected by the sensor, but under normal operating conditions in the testing environment stability is no longer an issue.

Another key design issue highlighted by Prototype 2 was mounting the ultrasonic sensor directly to the chassis. The tracks used for on the robot have a small amount of slip in them, which is exacerbated when using tracks to rotate on the spot. A single turn through 90° appears to be sufficiently accurate for navigation and mobility purposes, but many small turns (as would be required to perform a sweep

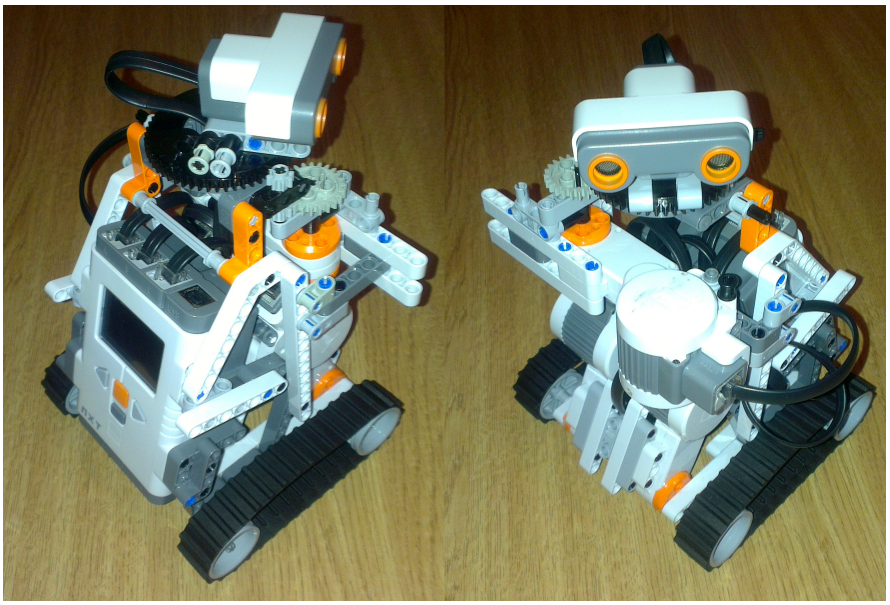


Figure 3.4: Final Design.

of the local environment) introduce a significant error. Mounting the ultrasonic sensor on a motor would allow it to turn independently of the rest of the robot. In order to keep the sensor as close to the center of the robot as possible (to reduce displacement through the whole robot rotating), the sensor was mounted on a geared turntable above the motor. The gearing allows for more accurate movements of the sensor, with several motor rotations equating to a single rotation of the turntable. The final hardware design is shown in Figure 3.4. The robot was built in such a way that the NXT Brick can be removed quickly and easily (to allow for battery changes &c). Due to the constraints of the connection cables, the sensor assembly can't turn more than 190° in either direction. This doesn't pose a problem though, as from the neutral (forward) position, the local environment can be scanned by scanning one way, returning to neutral, then scanning the other way.

3.3 Environment Design

The test environment is a rectangular area with foil coated walls and a hard floor surface (shown in Figure 3.5). The walls are constructed from cardboard boxes, a coffee table and a corner sofa, allowing the size and shape of the environment to be changed quickly and easily.

The environment has a hard floor (laminated wood), as early tests showed that the tracks cope better on hard surfaces than on carpet. This was most noticeable when the robot is turning on the spot; on carpet there is too much friction with the body of the track (rather than the protruding tread) to allow for accurate turning. Additionally, when turning on carpet the treads have a tendency to be pulled off the driving wheels. Whilst forward and backwards movement was less affected by the floor surface,

the tracks appeared to slip less on the hard surface.

The use of foil was suggested by a previous Final Year Project student, who found that using foil increased the accuracy of measurements [7]. In addition, foil was used as the materials used to assemble the environment are not uniform, and it was felt that the mix of leather, cardboard and laminated wood would not provide a consistent testing surface. However, it may be the case that all of these surfaces reflect (and absorb) ultrasonic waves at the same rate.

The environment can be re-configured to provide alternate shapes for testing. In addition to the walls, a foil coated box can be placed in the environment to alter the topography, else the environment will always be a rectangle of varying proportions.

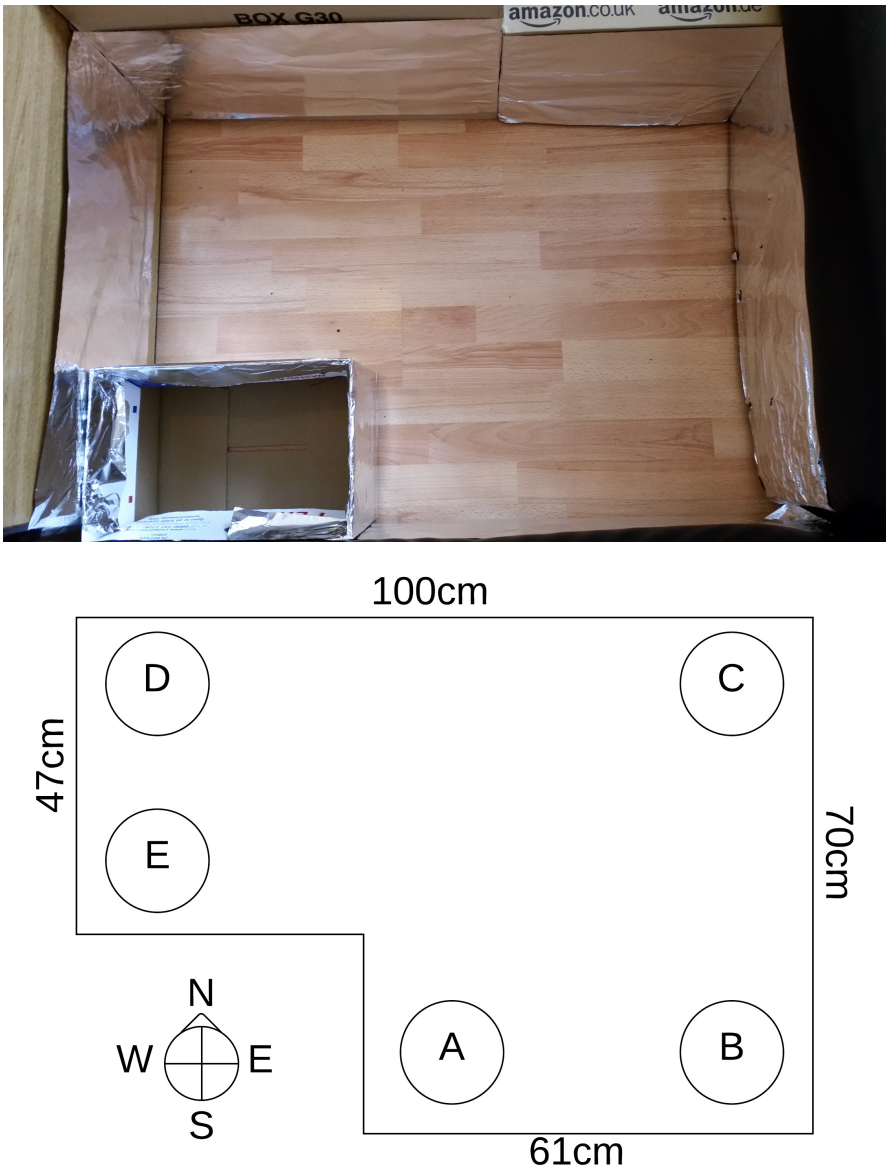


Figure 3.5: The testing environment. The compass on the diagram is used in Appendix D.

Chapter 4

Software Design and Implementation

4.1 Choice of Language

As discussed in Section 2.1.2, there are many options available for programming the NXT System. The two libraries considered for use are the RWTH MATLAB toolkit and the LeJOS Java library. Because both libraries implement similar functions (and I have no prior experience with either of them), I have decided to base my choice instead on the underlying languages.

There are some benefits that MATLAB provides that are unavailable in Java. MATLAB's `imshow` allows matrices to be displayed as images in a single function call, automatically building a GUI and handling image generation and saving. Nothing similar exists in Java, so a GUI needs to be constructed from scratch to display data in this way. Similarly, MATLAB implements image processing functions such as erosion and dilation that may be useful in the post-processing of maps.

However, there are solutions to these absences. Java's Swing GUI toolkit is suitable for building an interactive GUI for the project, allowing all necessary functions to be run at the press of a button with no command line input required. And implementing new post-processing functions rather than using pre-existing methods allows them to be tailored specifically for the purpose. Also, I have more programming experience with Java than with MATLAB, and a better understanding of the language.

Availability of development is also an issue to be considered. Where as Java is a programming language with open source implementations and IDEs, MATLAB is a proprietary software suite with

licensing fees. In order to use MATLAB therefore, all testing and running of the hardware would have to take place in the university computing labs. This would mean transporting the robot to and from the labs frequently, increasing risk of loss of parts/damage to the kit. Not only is this inconvenient, but using the labs would be potentially disruptive to other students.[1]

For these reasons, I have chosen to develop using Java and the LeJOS library.

4.2 Controlling the Hardware

A basic framework has been developed to allow for simpler access to functions that would be frequently used by the software. A custom 'Pilot' class (called `TreadPilot`) to control movement was created to handle actions such as turning on the spot and travelling a specified distance.

The `TreadPilot` class has a hard-coded ratio in of $1cm$ travelled = 34.95° motor rotation. This was calculated from the robot performing a 3600° motor rotation of each of the driving motors simultaneously, and working out therefore how many rotations equate to a certain distance. Over three trials, 3600° gave $103cm$ consistently, and the ratio was calculated. This scales down to 1 cm travelled for every 34.95° the motors rotate.

Even though the LeJOS `Motor` class only takes rotation commands as integers, the value 34.95° is used internally in calculations, and the required value is converted to an integer immediately before being passed to the LeJOS class. It is hoped that this will reduce error from internal calculations to $\pm 1^\circ$, leaving the only significant source of error coming from the hardware (i.e. motor slip and slack in the treads).

Another class called `SensorTurntable` was developed to streamline access to the ultrasonic sensor construction. `SensorTurntable` also keeps track of the current angle of the sensor relative to the neutral position (straight forward), to prevent the turntable rotating further than allowed by the cable. The gears in the construction are arranged in a 24:8:56 ratio, with the 24-tooth as the driving gear. This scales down to 3:1:7, giving the $\frac{3}{7}$ constant used in the class. The middle gear (8-tooth) is employed to allow the driving gear to sit underneath the turntable, making the mounting of the motor much more stable.

In order to send data back and forth between the robot and the application, a Bluetooth connection remains open continuously. Java's `DataOutputStream` and `DataInputStream` are used for the transmission of data, and data is sent as a string prefixed with a code describing the reason of transmission and/or the meaning of the data. These codes are listed in Table 4.1. The results of a sensor sweep are sent together as one string rather than individually. The reasoning behind this is that data

[1] The labs are made available to students as a teaching space and an environment for quiet work. Hardware development with the NXT kit typically involves looking for pieces of Lego in a large, rattling box; software testing often requires the robot (with its noisy motors) to be run. Working on this project in the lab is not really suitable.

Code	Meaning
-1	Terminate - Triggers the robot to shut down
0	Ready - Confirms the connection is active, and sends initialisation data: (Scan Angle, Minimum Clearance, Maximum Detection Distance)
1	Detections - Prefixes the results of a sensor sweep as (Angle,Distance) pairs
2	Next Move Request - Triggers the application to calculate and send next move
3	Next Move Data - Prefixes the next move in (Angle, Distance) format
4	Move Correction - Prefixes the corrective data calculated upon movement completion in (Distance Travelled, Direction of Drift, Drift Distance) format

Table 4.1: Bluetooth Prefix Codes

stream write and read operations are (typically) slow compared to other operations, and so one single write/read pair should be much more efficient than 72+ pairs. In addition to the any savings from the I/O, if every detection was sent individually they would all need prefixing, increasing the amount of data sent and the amount of decoding needed to comprehend the incoming messages.

4.3 Structural Overview

The software is implemented in a client-server style architecture. The client software runs on the robot and streams collected data back to the server (a laptop/PC) for processing and display. This architecture is proposed for two reasons. The on board memory and processor are unlikely to be sufficient to perform all the algorithms and retain results. Secondly, visualising the current state of the robot on the NXT Brick's screen, a 100 X 64 pixel monochrome LCD, is difficult (as well as impractical; having to follow the robot round as it moves to get output is not user-friendly).

Using this strategy, the robot hardware becomes a peripheral, and can in theory be replaced by any similar hardware that uses the same control and data structures.

Figure 4.1 shows an outline of how the two halves of the software work together. Not shown on the diagram are the termination of the robot software (occurs when 'Terminate' code is received) or the GUI (which sits on top of the PC application shown).

4.4 Movement and Navigation

The 'Calculate Next Move' and 'Perform Move' processes in Figure 4.1 can be implemented in a number of ways. The first iteration of the software chose the furthest cardinal distance as the next move, cardinal distance being the distance to detections at $0°$, $90°$, $180°$ and $-90°$ ($270°$) from the robot. The distance and angle were then transmitted to the robot, which moved to the new position and performed the next scan. This approach had a number of issues. The primary problem was that the

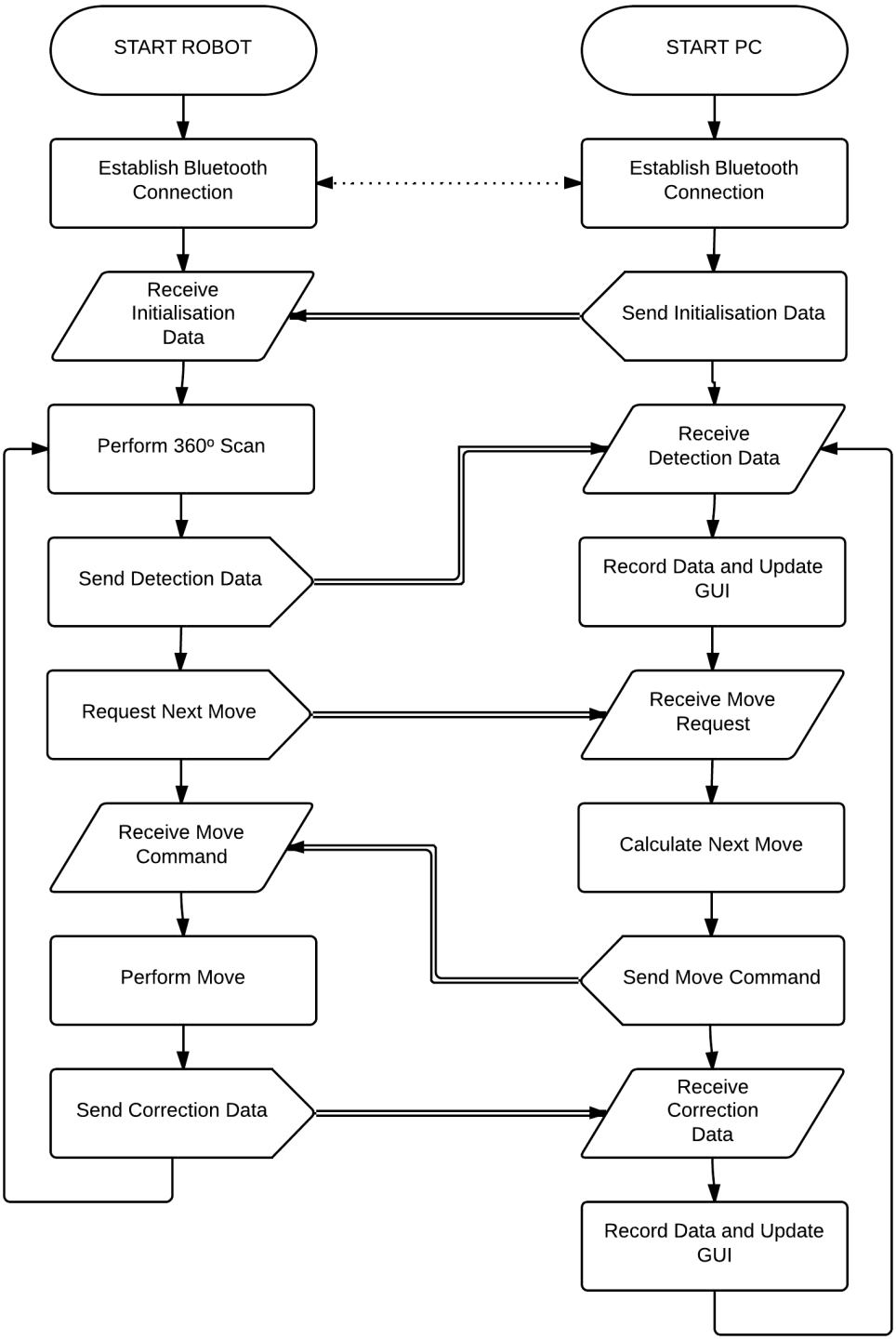


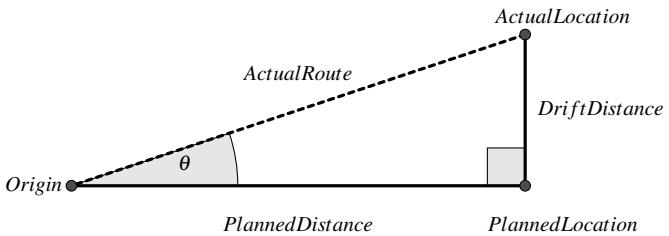
Figure 4.1: Software layout used by the robot and PC applications.

movement relied totally on previous detections and robot rotation to be accurate, as the robot made the movement without actively scanning to prevent collision. In many cases the robot ended up colliding with the environment, mainly due to slight inaccuracies in turning being amplified as the robot moves forward. And in cases where the robot didn't collide, it slowly became further and further from the estimated position. The other main issue with this approach was that the robot would tend to move back and forth between two locations, each being the furthest point from the other.

The next iteration of the software sought to address these issues. The next move for the robot is was determined based on a history of the previous moves, whilst still using a 'furthest-first' approach. In cases where the two previous moves were both turns of 180°, the next move cannot be 180°. The idea behind this was that it would prevent the robot getting stuck in a loop.

The biggest change to the software is that the robot now moves incrementally in steps of 10*cm*. Before moving, the robot scans straight ahead to ensure that the nearest detection is at least 20*cm* ahead (10*cm* for the movement and 10*cm* clearance). The robot will always try to leave a 10*cm* space between the sensor and the walls to allow for the bulk of the robot and slight movement when turning. The robot scans the environment as it moves to determine if the it has deviated off the predicted path. To do this, a scan of the robot's West, North and East directions ($-90°$, $0°$ and $90°$) is taken after every 10*cm* step and stored in an array. Whilst the North direction is measured only to prevent head on collision, the East and West measurements are used to calculate errors in movement by recording if the robot moves towards (or away from) a wall.

In order to find a wall, a number of assumptions are made. It is assumed that the side with the most constant rate of change is likely to be the best wall to use for calculating deviation from planned course. The diference between the initial and final measurements is assumed to be the 'drift' distance. This distance is sent back to the PC with the distance travelled, allowing for the angle of deviation to be calculated, as shown in Figure 4.2. This angle is then added to the next move, allowing the robot to correct the drift as it moves. Additionally, this step allows the robot to keep track of its position more accurately, as it will now record the 'Actual' location rather than the 'Planned Location' as before.



The angle of deviation, $\theta = atan(\frac{DriftDistance}{PlannedDistance})$

Figure 4.2: Calculation of actual distance travelled and angle of deviation.

4.5 Data Storage

In order to store the data gathered by the robot, a sensible data structure is needed. For storing spacial data associated with poses and detections, there are two different design options that can be implemented; a list of detection and location coordinates, or a matrix representing the environment. Both methods were implemented to find their relative strengths and weaknesses.

Point-List Based storage

- **Ease of Programming** - A point-list approach is (theoretically) much easier to program as native Array and List structures exist in Java and are suitable for this approach.
- **Ease of Use** - Coordinates are stored as an (x, y) pair, with the values of x and y representing the displacement (in cm) from the robot's starting location. A new point can be added to the list without effecting prior contents in any way.
- **Visualisation Considerations** - Because points are stored with the origin not necessarily being the smallest point (and therefore at a corner when visualising). The most distant x and y values need to be stored (or calculated) in order to provide an offset value to move the smallest point from $(-n, -m)$ to $(0, 0)$. This is an additional step that must be taken prior to every visualisation.

Matrix Based storage

- **Ease of Programming** - There are no native matrix classes as such in Java, but a matrix can be emulated by using an array of arrays: `int[][]`. A Boolean array could be used to represent a wall as true and empty space as false, but Integers are chosen so that detections and locations can be stored in the same structure.
- **Ease of Use** - As the size of the environment is not known before-hand, the matrix must resize dynamically as needed. The origin of the robot is always at the center of the matrix, and the matrix grows as needed. This requires allocation a new array and copying over all previous data.
- **Visualisation Considerations** - Because of the nature of the matrix, all space in the environment (detections, locations and empty space) is stored. Every `[i][j]` in the matrix can be easily mapped to an (x, y) on the canvas by the following formula:

$$(x, y) = ((i * Scale) + xOffset, (j * Scale) + yOffset)$$

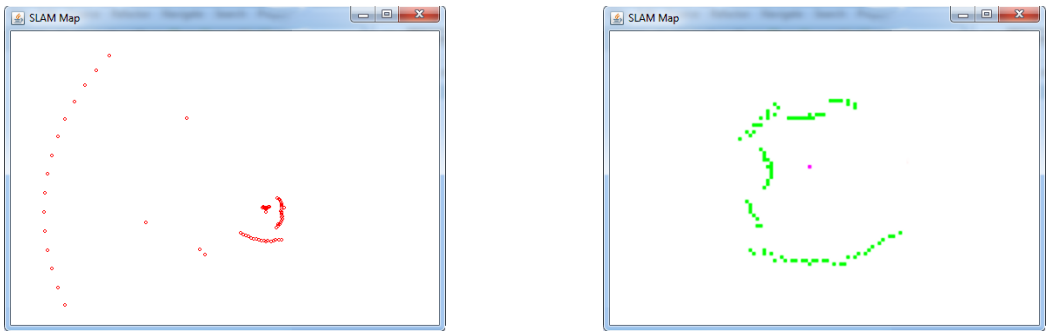


Figure 4.3: Initial map visualisation using point-based (left) and matrix-based (right) storage. The two maps show different data sets.

> *Scale* represents the size of every square drawn. A *Scale* of n means that each matrix entry is represented by a square of n-by-n pixels on the canvas. $xOffset$ and $yOffset$ are variables used to help align the matrix on the canvas.

Ultimately, a matrix based approach was used as the primary storage mechanism as it proved much easier to visualise, and provided an intuitive way to perform post-processing techniques on the data. In addition, using a matrix-based format emulates most image storage structures, massively simplifying the process of saving maps for later use.

4.6 Visualisation and GUIs

As mentioned in Section 4.1, a visualisation mechanism is required to display the current map and localisation data. It is therefore a reasonable to suggest that a GUI should be constructed to better display this information, and provide an easy interface for interacting with the robot. The existing application prior to the GUI being implemented works non-interactively from the command line, with variables such as the number of movement steps or the angle between scans being hard coded in. Ideally these ought to be alterable on the GUI before the robot initiates.

There are multiple ways to visualise the data, and the GUI should be able to switch between these on the fly. Additionally, the GUI must show the current location and orientation of the robot in order to demonstrate the localisation.

The most important part of the GUI is the visualisation pane. A number of visualisation options are available, including Java's native `Graphics` and `Graphics2D` classes, and external libraries such as OpenGL bindings packages Java Open GL (JOGL) or Lightweight Java Game Library (LWJGL). The latter were considered as I am familiar with the workings of OpenGL in C++. However, after some initial research it became apparent that Java's native libraries were more than suitable for the task.

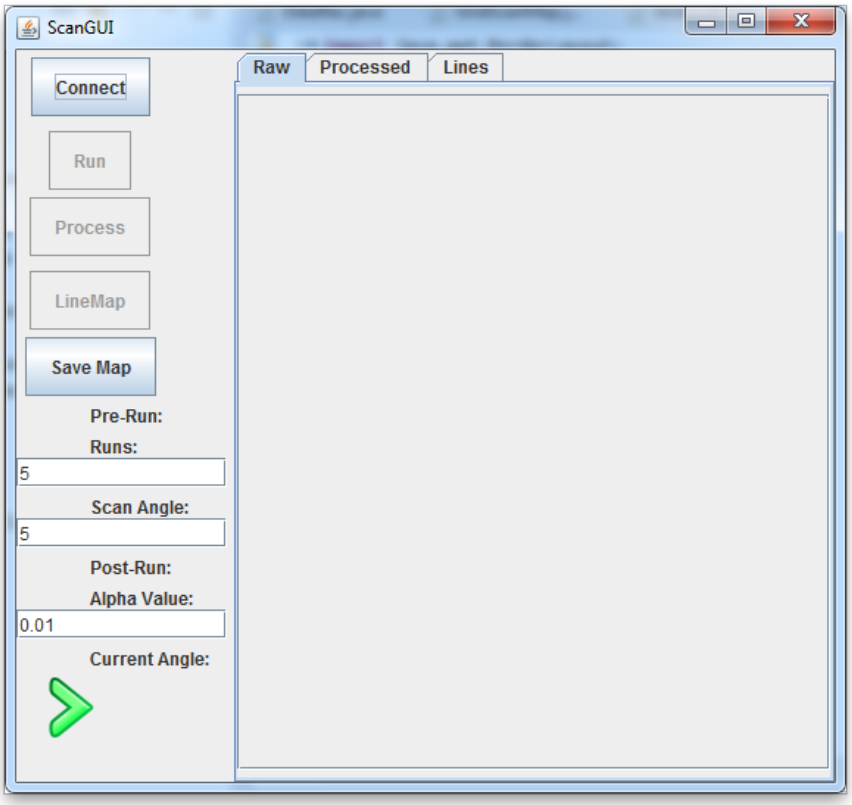


Figure 4.4: Screenshot of layout of the GUI using a `JTabbedPane`.

Initially, the Point-based data storage method was used to provide a simple way of testing methods. The points were drawn as small ovals on the canvas, but at this stage the position of the robot was not shown, and a new window spawned each time the data updated as no GUI was built to house the panel. Figure 4.3 shows the first successful visualisation attempt.

After evaluating how point-based storage worked with the visualisation, the matrix storage method was tested. It proved easier to program, and also highlighted a flaw from in the point-method. In point-based storage, many detections can be very close to one another, leading to overlapping clusters of points. In the matrix however, all detections are 'snapped' to the nearest grid square and multiple detections in a single space are only processed and rendered once. This results in a computational saving (which at this scale is practically irrelevant) and also a clearer output.

Once the basis of the visualisation was implemented, a GUI was needed to regulate the canvas and provide an interactive way to control the robot. The command line program automatically runs the robot immediately upon connection with hard-coded initialisation variables. For the GUI, it was decided that once the connection had been established, there should be opportunity to alter the runtime parameters.

In addition to the view showing just the 'raw' detection data, a second view showing 'processed'

data was also created using the same visualisation functions. A number of processing methods were implemented and tested, and are discussed in detail in Section 5.3. These methods require their results to be displayed on a separate map from the raw data. Java's Swing toolkit provides a number of components that help in this regard. The `JTabbedPane` component allows switching between the different views in a simple manner, whilst also keeping the interface clean and simple. Instead of displaying and managing a separate window for each view, all the views are tied to one window. This also makes adding additional views easier, as rather than building a new window for the component, the method call `addTab("Name", Component)` will automatically handle sizing and other issues. Figure 4.4 shows a near-finished GUI.

Whilst this GUI worked well, in order to save maps a file dialogue must be used to name the output file for each view of the map. This is far from ideal, especially when up to six views can be generated per run. The additional views are the 'Lines' view described in Section 5.4. A more robust GUI was built, allowing all maps to be saved with the same base filename, with appropriate suffixes.

The new GUI (Figure 4.5) displays the current map, and the localisation of the robot. The green squares are detections, the blue squares are locations that the robot has visited and scanned, and the larger red square indicates the current location of the robot. The current angle is indicated by the green arrow beside the map. Because both the raw and processed maps work from the same data, when flipping between the two maps unaltered data points will be drawn in the same place, facilitating a quick comparison.

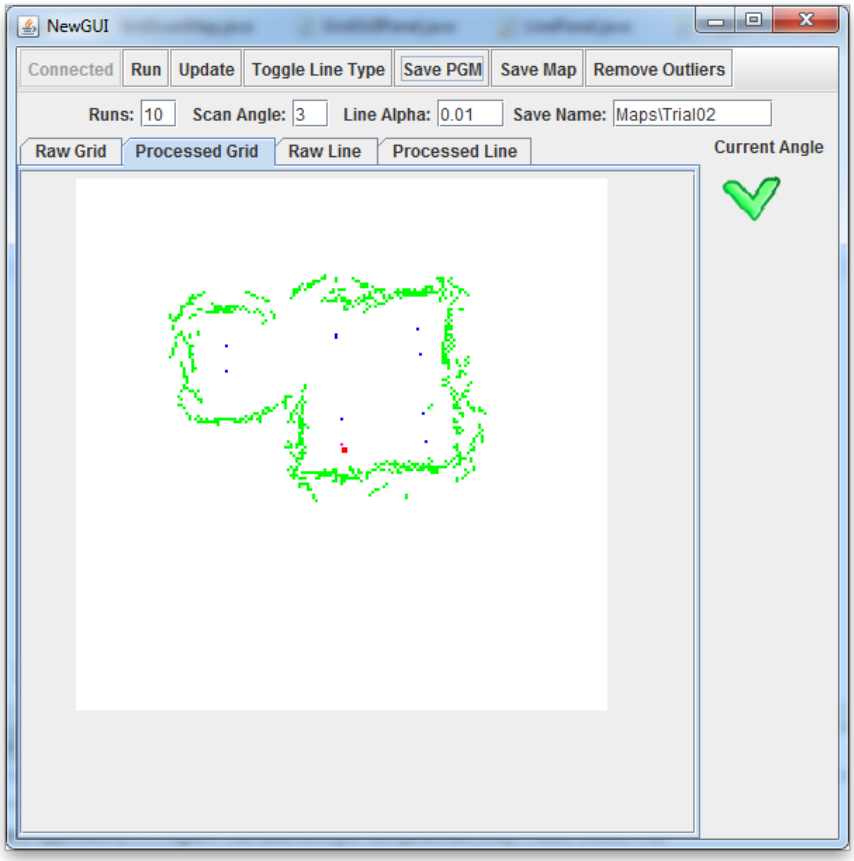


Figure 4.5: The second (and final) iteration of the GUI.

Chapter 5

Analysis and Improvements

5.1 Hardware Component testing

In order to assess the success of the robot, an understanding of how accurate the system can reasonably be expected to be is required. Working on the notion that the system as a whole can't be more accurate than its component parts, the accuracy of the motors, sensors and control classes (as described in Section 4.2) can be tested as distinct units.

5.1.1 Linear Movement

The accuracy of the motors (and the `TreadPilot` class developed to provide an interface to them) was measured by issuing "`Move forward` *n* `cm`" commands and assessing the resulting movement. The results given in Table 5.1 show that forward distance is fairly accurate. However, the robot deviates to its right, and these deviations become more significant at longer distances. It was believed that these deviations are caused by the `TreadPilot` class. LeJOS provides no way of simultaneously controlling two motors, instead offering an option to 'Immediately Return' and allow the next command to be processed. However, swapping the order in which the motor commands were called did not remedy the problem, and the issue remains unexplained.

	Distance Travelled					
	Forward			Sideways		
Distance Specified	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3
10	10	10	10	<1	<1	<1
20	20	20	20	1	<1	1
50	51	51	50	3	3	2
100	101	100	101	7	6	7
150	150	151	150	14	15	15

Table 5.1: Results of Linear Movement Testing. (All values in cm)

5.1.2 Ultrasonic Sensor Accuracy

The accuracy of the ultrasonic sensor was tested by manually placing the robot at a given distance from a surface, and recording the distance reported by the sensor. Whilst this process could have been automated, automation would have added other sources of error. The robot was placed at 10*cm* intervals from 10*cm* to 250*cm* from the surface. A reading of 0*cm* would not have been possible, as the front of the robot protrudes beyond the front of the sensor. When the sensor is unable to measure a distance, it returns 255*cm* as the measurement.

The results, shown in Figure 5.1, indicate that the sensor is reasonably accurate (within the $\pm 3cm$ reported accuracy) at distances of $10-90cm$. Interestingly, all three assessments at 100*cm* yield results around the 75*cm* mark. This could indicate systematic error in the testing process, but the measured surface directly in front of the robot was the closest surface in the testing environment, making this unlikely. Results at the 110*cm* and 120*cm* marks correlate with the previous results. Distances beyond 120*cm* give more varied results, and no measurement was recorded at any distance beyond 150*cm*. These findings have lead to a distance limit of 80*cm* in the code, with any measurement above this distance being regarded as incorrect and ignored (i.e. set to 255).

5.2 Navigation Strategy

As discussed in Section 4.4, always choosing the furthest location to move to would frequently lead to the robot moving between two 'optimum' points (such as C and D in Fig 3.5). The use of a 'move history' was supposed to correct that. However, when observing the robot it became apparent that this method didn't break the cycle, it simply extended the loop caused by the 'furthest-first' approach. A new method of determining the next move was needed.

A method was implemented to investigate whether a stochastic selection would be appropriate. This method picks a random cardinal direction from a subset of suitable directions. A direction is deemed suitable if the closest detection in that direction is far enough to allow the robot to move at least once (according to the movement strategy described in Section 4.4). If no suitable direction is found, the

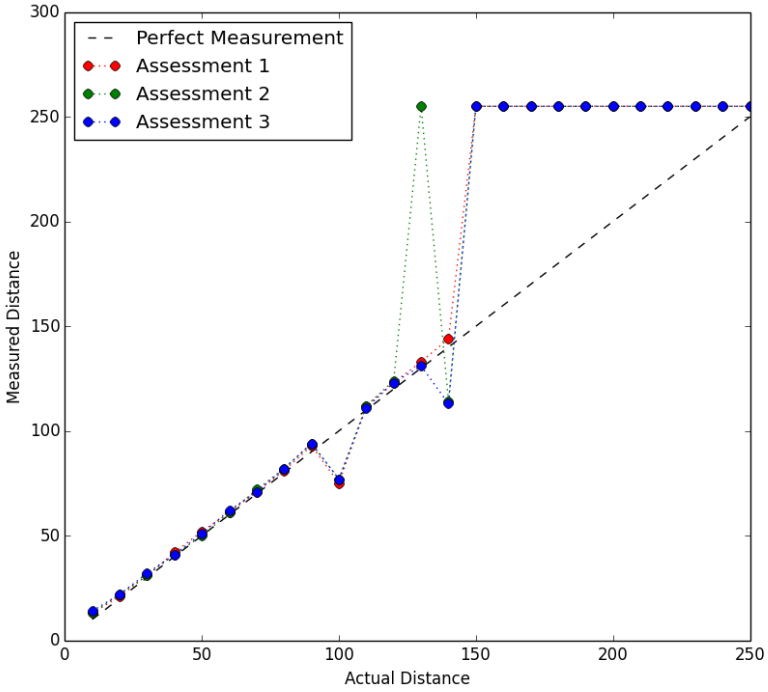


Figure 5.1: Results of Ultrasonic Sensor Testing.

robot is asked to turn 180° and make a 'blind move' of 80*cm* (the maximum accurate distance of the sensor from Section 5.1.2). In this context, a 'blind move' is a move taken to get the robot out of an otherwise impossible environment. It is 'blind' as the robot has no notion of how far the walls may be in that direction, and so it relies on the 'step-and-scan' movement strategy to avoid collision.

5.3 Data Processing Methods

The first of these was a simple algorithm that joined adjacent points (described in Figure 5.2. This was somewhat successful, but in addition joining the lines representing the walls (as was intended), false detections were also joined. if the algorithm was run again, points added by the algorithm could trigger new points to be added. The algorithm was altered, and points added were the recorded differently from actual detections. Even after alteration though, the algorithm didn't prove as useful as imagined. An alternative approach was then taken. Instead of adding points to the map, it was possible that removing points would improve the quality of the map. An outlier stripping method was implemented, similar to the 'erosion' process commonly found in image processing. If any detection had no neighbours within n squares, it was removed. However, the algorithm could remove detections that may have had neighbours after subsequent scans and erode correct detections. To avoid this, each time the algorithm is run it pulls a copy of the current raw map to erode and saves that as the new processed map.

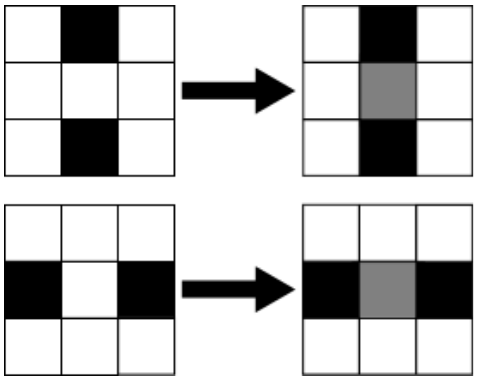


Figure 5.2: Line joining algorithms.

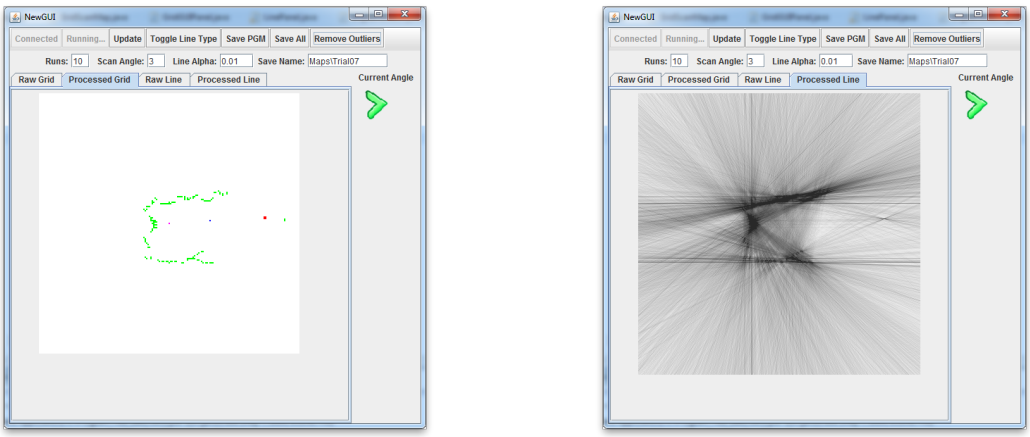


Figure 5.3: Map visualisations using grid and infinite line methods.

5.4 Alternative Visualisations

Whilst the 'grid' map provides a good way of displaying the data, other methods were explored. The most successful of these was the 'lines' method. This method involves drawing an infinite line going through every pair of points. If any two lines intersect, a point is drawn the location of the intersection. The idea behind this is that the edges will have many intersections, and will appear darker, providing another way of finding the walls of the environment. After implementing the method, it became quickly apparent that the visualisation worked much better with the outlying points removed. This method introduces lots of noise, but the majority of this could be removed from the output image by a simple thresholding algorithm. The two images in Figure 5.3 show the two map views in the running application, and Figure 5.4 shows the grid map overlaid onto the line map.

This method is more useful with fewer data points spread across the environment. The more data in the map, the more lines draw, and the map quickly gets saturated. Every line drawn is transparent, so by increasing the transparency (decreasing the alpha value) of the lines, the map can remain comprehensible with more data. This led to the 'Line Alpha' box on the GUI. The alpha value can't be

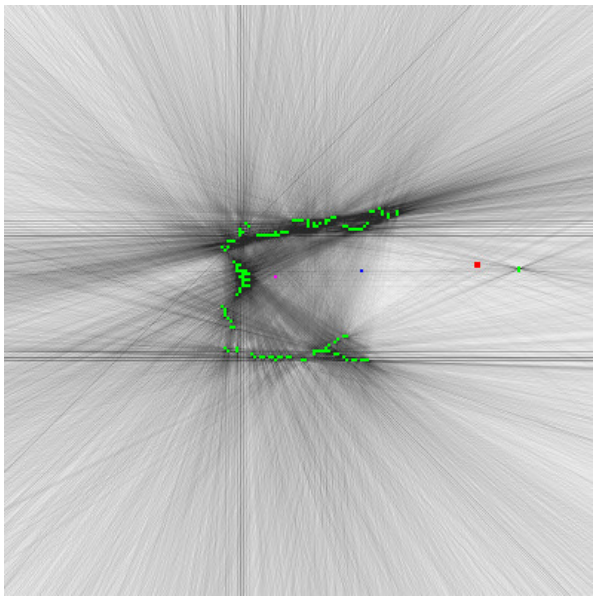


Figure 5.4: Overlaid grid and line maps.

decreased indefinitely however, and values lower than or equal to 0.001 are not rendered. Even with (relatively) few data points, this method is susceptible to 'point clustering'. A cluster of close points can cause undesired saturation local to one area, such as that on the left wall of Figure 5.4.

A use was found for the saturated maps however. By chaining the line type from an infinite line thought the points to a line segment between the points, the shape of the environment can still be determined. Rather than highlighting the edges, the entire interior of the map is coloured giving a silhouette. Whilst the map in Figure 5.6 isn't saturated, it gives an idea of how the silhouette works. It also highlights a significant issue with it too. As the algorithm connects every set of points, it will always end up giving the convex hull of the environment rather than its actual outline. For environments that are convex shapes this poses no issue, but concave environments like the testing environment will not be correctly represented.

5.5 Comparison of Maps

Appendix D provides the data output of ten runs of the robot giving both the raw and processed maps. The line maps aren't included, as they had become unhelpful, some being so saturated that they are essentially a solid black image. Appendix E shows the processed maps overlaid onto the actual outline of the environment. Because every pixel in the map represents a 1cm square, it is fairly simple to create a template matching the dimensions of the environment given in Figure 3.5. The line ups had to be made by hand, as no suitable automatic way of composing the template to the maps was found.

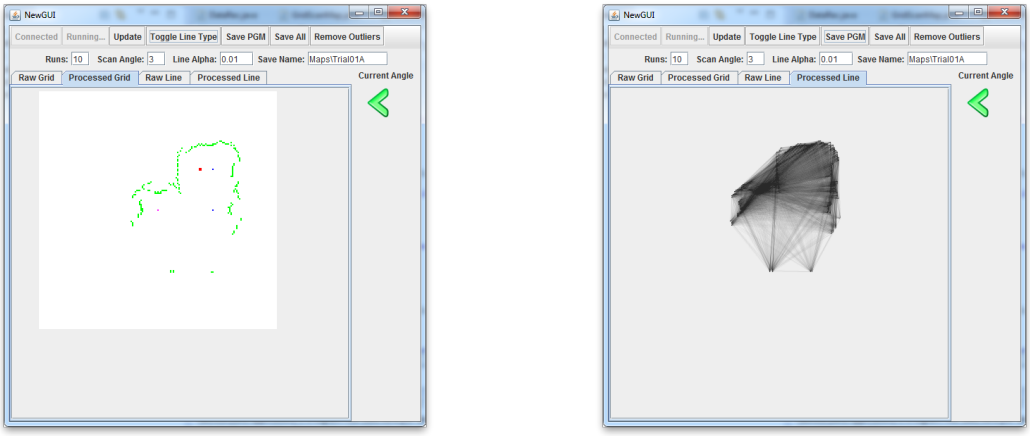


Figure 5.5: Map visualisations using grid and line segment methods.

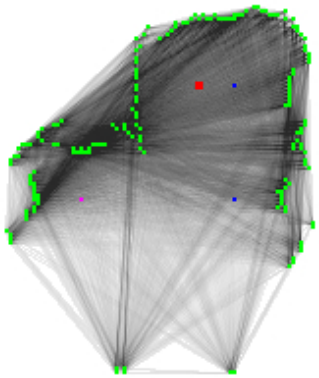


Figure 5.6: Overlaid grid and line maps.

Run	Perfect Points	Interior Outliers	Exterior Outliers	Total Outliers	Total Points
1	542	265	138	403	945
2	510	246	248	494	1004
3	422	303	268	571	993
4	416	259	314	573	989
7	437	253	334	587	1024
8	472	221	312	533	1005
9	445	181	253	434	879
10	489	221	207	428	917
5	529	182	280	462	991
6	337	289	262	551	888

Table 5.2: Breakdown of Detections from Maps in Appendix E.

It is difficult to numerically quantify how close a map is to the ground truth. One possible method would be to determine the distance between each point and the template and then give the average of these as a measure of accuracy. Alternatively, points could be given a weighting based on distance, with perfect detections scoring zero. This method would unfairly favour maps with fewer points though, as a map with 4 perfect points would receive a better score than a map with 500 perfect points and one significant outlier.

The method implemented is more simplistic than either of these, simply counting the number of points inside or outside the environment. Both categories of errors have different consequences. Interior errors can cause the robot believing that part of the environment has been explores, leading to areas not being explored. Exterior errors can make the environment seem larger than it really is, and prompt the robot to attempt to move to places it can't.

The results of this method are shown in Table 5.2. The method is applied to the processed maps that have been subject to the erosion method discussed in Section 5.3. The results for Maps 5 and 6 are unreliable, as in order to fit the template the data had to be rotated. This led to artifacts in the image that had to be removed by thresholding, which may have removed some actual data points. As such, any further analysis of the data does not use these maps. As every map has a different number of detections, these figures are more easily compared as percentages of total detections, as shown in Table 5.3.

Although the outlier rates may seem discouragingly high, it must be considered that a point is classed as an outlier if it is outside $\pm$1cm of the actual edge of the environment.[1] To put this in context, the average size of the processed map files was 122 by 96 pixels (after trimming excess whitespace). This gives a total number of $11,712$ locations for the robot to have allocated points. The total number of pixels considered 'perfect' by the template was $1,020$. This gives around 8.7% of the environment

[1]Please note that **outlier** here refers to points that are not classified as 'Perfect'. This is distinct from the meaning used in Section 5.3, where it refers to candidates for the erosion method.

Run	Perfect Points	Interior Outliers	Exterior Outliers	Total Outliers	Total Points
1	57.35%	28.04%	14.60%	42.65%	100%
2	50.80%	24.50%	24.70%	49.20%	100%
3	42.50%	30.51%	26.99%	57.50%	100%
4	42.06%	26.19%	31.75%	57.94%	100%
7	42.68%	24.71%	32.62%	57.32%	100%
8	46.97%	21.99%	31.04%	53.03%	100%
9	50.63%	20.59%	28.78%	49.37%	100%
10	53.33%	24.10%	22.57%	46.67%	100%

Table 5.3: Scaled Breakdown of Detections from Maps in Appendix E.

being considered as locations of 'perfect' detections. Over 48% (on average) of all detections were in the correct 8.7% of the environment. When looking at the data visually, it can be seen that the majority of outliers would have been caught if the template covered $\pm$2cm, significantly increasing the perceived accuracy.

Chapter 6

Evaluation

6.1 Schedule

After the initial meeting where aims and objectives were discussed, a project schedule was drawn up in the form of a Gantt chart. Although the project started out following the schedule fairly closely, the amount of time needed to construct the hardware and implement all the software was not well estimated. The issues with the Bluetooth connection (discussed in Section 6.2) added about a week to the start of the implementation phase, and held up hardware development too, as prototypes couldn't be tested. In addition to this, the construction of a GUI was not initially anticipated, and so not accounted for in the schedule.

Whilst the Gantt chart proved useful as a list of tasks that needed to be accomplished, its value as a schedule of tasks lessened considerably in the latter half of the project. A revised schedule more accurately reflecting the timescales of the project is given in Figure 6.1.

6.2 Choice of Language and Libraries

LeJOS was a straightforward and simple library to use. It includes a plugin for the Eclipse IDE to manage firmware and upload programs to the robot. This is also possible through using packaged command line applications, but the IDE integration is much faster to use, with the code being compiled, uploaded and run by a single command.

Debugging the applications running on the robot proved less straightforward, with the Brick's screen giving a number of codes that need to be run through a command line tool called `nxjdebugtool`. And any time a change was made to the, the program had to be re-compiled and re-uploaded, a process that can take some time as the Bluetooth connection was temperamental and would frequently fail to connect. Once a connection had been made though, it was stable.

The use of LeJOS had some unforeseen consequences. The Bluetooth connectivity libraries rely on a third party library called BlueCove that has a number of implementations for different system architectures. Unfortunately, the OSX build relied on methods that had been deprecated and removed in recent versions of OSX. A significant amount of time was spend troubleshooting the problem, downloading old drivers and trying alternative libraries get the Bluetooth connection to work. The solution found was installing a Windows VM onto the system in order to access the Bluetooth functionality.

However, it is possible that the RWTH toolkit would have its own issues surrounding connectivity. The toolkit's documentation suggests that setting up a connection requires a number of additional steps, including generating a configuration file for the machine being used. LeJOS however allows users to create a `NXTConnector` object, and then call a single method to initiate a connection, without any additional configuration detail needed. A further method then gives the programmer access to the connection through Java's native `DataOutputStream` and `DataInputStream` classes.

Java's `Graphics2D` library, as discussed in Section 4.6, was deemed to be suitable for the visualisation required. This judgement held for the grid-based visualisation, but the rendering and saving of the line-based maps was very slow, taking as long as a few minutes for the map data from Appendix D. If a more robust graphics package such as OpenGL was used then these maps would likely have rendered much faster. It is also possible that the saturation problem of line maps discussed in Section 5.4 would have been avoided by using OpenGL, as it is designed for 3D graphics and so has been optimised to render transparent objects.

6.3 Design of the Robot

Overall, it seems that the produced hardware solution was appropriate for the purpose. It met all the required criteria and performed as expected. The primary goal of the robot's design was to make the robot as compact and stable as possible. Initial investigations during building showed that if the structure of the robot was not suitably reinforced then it would warp. This is mainly because of the way that Lego pieces are joined together; beams are connected by using round pegs. Whilst this allows for beams to be joined at any angle, the pegs don't hold the angle, meaning the joint can rotate. This can be desirable in some circumstances, but not when using the joints for structural purposes. In order to reinforce the beams, diagonal bracing was required. Unfortunately, due to the fixed spacing of holes in the beams, it is often difficult to find an exact fit, leading to the beams to be put under strain.

In retrospect, it may have been possible to avoid this by planning the design on paper. However, this would be a very time consuming activity and may not help at all.

Whilst the final design of the robot was undoubtedly suitable for the purpose, there are many improvements that could have been made to it. The biggest improvement that could be made would be to improve the accuracy of turning. Possible ways of doing this would to include a gearing system for the drive tracks. This would theoretically allow for finer control over the rotation of the tracks, just as gearing was used in the construction of the sensor turntable. However, it is possible that the gearing would decrease the power output to the treads, causing other issues.

Although the three-wheel configuration was dropped at the prototyping stage in favour of tracks, it the flaws exhibited were fixed then the development of the software may have been much simpler. LeJOS has inbuilt navigation and control systems for vehicles following the three wheel scheme, but this wasn't discovered until after tracks had been adopted. Wheels wouldn't have suffered from the slippage problems that tracks do, though it is still possible that the tyres would work loose of the wheel hub just as the tracks occasionally did.

On the whole, the robot performed well. The design made it simple to program for, and the use of gearing in the sensor rotation mechanism allowed the sensor to be rotated accurately. Naturally with the benefit of hindsight there are a number of different design choices that would have been made, but the produced solution more than met the requirements.

6.4 Navigation Strategy

As examined in Section 5.2, the navigation strategy employed by the robot was stochastic. This has obvious downsides, primarily in that there is no guarantee that the robot will explore the whole environment. To help combat this, the robot can only perform a 180° turn and directly 'undo' its last move if there are no other valid moves that can be made. In all the test runs documented in Appendix D (and all those not recorded), the robot did manage to explore the whole testing environment within ten movement phases.

It has been suggested that a deterministic method such as 'wall-crawling' would have been more suitable. This approach is designed to strictly follow the outside walls of the environment, ensuring it is entirely covered. The only exception to this in environments with more than one non-continuous wall, such as an 'island'. In these cases, the robot relies on its sensor to detect that wall (but not visit it), or once the robot has made a circuit of a wall a secondary strategy may become active and try to find a new wall to follow. If the robot 'loses' a wall (as may happen when travelling past the internal corner of the test environment) it will have to attempt to turn towards the wall last followed. All of the navigation strategy implemented assumed that the environment was linear and that all walls were either parallel or perpendicular to all other walls.

6.5 Environment

Only one environment was used throughout most of the testing process. This is not ideal for testing the system, and environments of varying sizes may have been more suitable. In early stages of development a plain rectangular environment was used. This worked well for testing movement and sensor accuracy, but didn't provide enough complexity to really test the navigation strategies. In the rectangular environment, the robot would move until finding a corner, then turn to follow the next wall and so forth. The internal corner gave better opportunities to test strategies, with the robot having to choose a way to turn rather than 'sticking' to a wall. The internal corner also acted a s a demonstration of how the scanning methods coped with reflex angles. All other angles in the environment are 90°, and prior to the addition of the box there was no way of testing other angles.

As it stood, the environment used was as large as it could have been with the materials I was using. Smaller environments could have been produced by moving the 'North' wall (the wall at the top of the image in Figure 3.5) further into the environment, and the box that became the internal corner could have been moved round. It had to be taken into account that the robot's footprint measures about 17-by-15cm, and that it ideally requires about 5cm clearance on top of that in case of track slippage. This means that the robot is liable to collide with the environment in spaces much smaller than 25cm. A ring-shaped environment was tried, but it was found that the robot would frequently clip the edges of the internal box and drag it around, changing its position and therefore altering the environment.

6.6 Software Design

In most SLAM implementations, the locations of previous landmarks are updated as the robot moves through the environment. However, this technique relies on having unique recognisable landmarks. For this project, the landmarks would equate to detections. However, because detections aren't unique there are a number of issues with this approach.

One possible way of getting round the problem would be to 'follow the line of sight' of a detection and find other detections that could correspond to the same landmark. These detections would then be assessed by some criteria such as distance from each other, and if they were judged to be the same detection then they would both be removed and replaced with a detection half way between. This method was considered but not implemented, as an immediate problem arose. In an environment with internal an internal corner (such as the testing environment) it is possible for the robot to be on the same 'line' as a wall such that a detection of the corner will also include detections of the entire wall. In less extreme cases, the line of sight may cross through the outside of the environment and end up 'cutting the corner'.

As an alternative, a method of comparing the latest scan to the map was attempted. The idea behind this approach was that the software would compare the latest scan and try and match it up to previous scan data on the map. Obviously if the robot had not explored or scanned that area previously then no match would be found, so the method would only trigger if the current robot location was within a certain distance of a previous location.

There proved to be a number of issues with this method though. Due to the inherent inaccuracies of the sensor, two scans taken from the same location can look fairly different. This is down to whether a scan at a given angle actually returns the correct value, or returns 255 and is ignored. This problem wasn't encountered in the testing of the sensor (Section 5.1.2), most probably as the readings were taken from head on to the surface rather than at an angle to it. As such, comparing two scans that were of the same place proved difficult. The method attempted to compare the scan with the map by running a comparison of the two with the scan's origin at a certain position on the map, recording the comparison, then moving the scan's origin to the next i, j on the map and repeating. The actual comparison consisted of counting the number of overlapping points. I planned to extend it to assign scores to the points, with a perfect match getting 0, a match within one square getting -1, and so forth until certain threshold, where the point would receive a very low score. The overall highest scoring set of matches would be picked.

The method didn't really work though, and I hadn't managed my time well enough to continue with this apparent dead end. I needed to progress and gather test data, consequently the method was abandoned without being implemented at all. The final implementation of the software didn't have this landmark updating stage, and as such might not be considered a true SLAM implementation, but rather a mapping algorithm with some naive localisation.

6.7 Analytical Methods

The data analysis methods used were all ultimately subjective. As noted in Section 5.5, no automated way of applying the ground truth was implemented and so the templates were overlaid by hand. The main reason for the lack of automation was that although all maps used the same template, the robot's starting location is always recorded to be the centre of the image. Thus, the rest of the map will be represented in a different part of the image for every starting location tested. This is avoidable by running all tests from the same starting location, but it was felt that the robot's performance should be assessed from a number of different starting poses.

Because there was no 'naive' way of applying a template, more complex methods such as the Generalised Hough Transform were explored. Finding suitable implementations of these methods was difficult, and each implementation required data formatting in a specific way. Because the storage mechanism used was 'home made', there was a significant complexity involved in attempting to

adapt these methods to fit the required purpose. Ultimately, it became apparent that implementing these methods in the time left was not feasible, and so the manual approach was used as a compromise.

The metrics used to quantify the maps don't particularly express how 'complete' a map is. For example, Trial 9 was one of the higher-scoring runs in terms of percentage of 'perfect' points, even though it had the lowest total point count of any of the trial runs. Looking at the map (Figure D.9) shows a clear 'gap' in the environment. A metric to convey how much of the template was left unmapped would also have been useful for this reason.

Because all the trials were run for the same number of movement steps, comparing the relative accuracies of maps based on time (i.e. steps) was not considered. However, when combined with the completion metric described above, it would be a useful measure of how many steps are needed to map the environment. It could also be used to provide a real-time measure of accuracy if a way of automatically fitting the template was found. This would be beneficial, as all other methods are only really applicable after the data has been gathered.

6.8 Assumptions and Limitations

In any project, there are always some assumptions made. Within this project, the most significant assumptions relate to the environment in which the robot operates.

This solution relies on the environment to be entirely linear, and the solution copes best when the environment consists of walls that are 'aligned to a grid' (i.e. all walls are either parallel or perpendicular to all other walls). This assumption rather constrains the applicability of the solution when it comes to real world application, as these assumptions can't be guaranteed. That being said, floor layouts of many buildings, the University included, are inherently linear and are usually built around straight lines and 90° angles, so the application maybe isn't entirely irrelevant outside the testing environment.

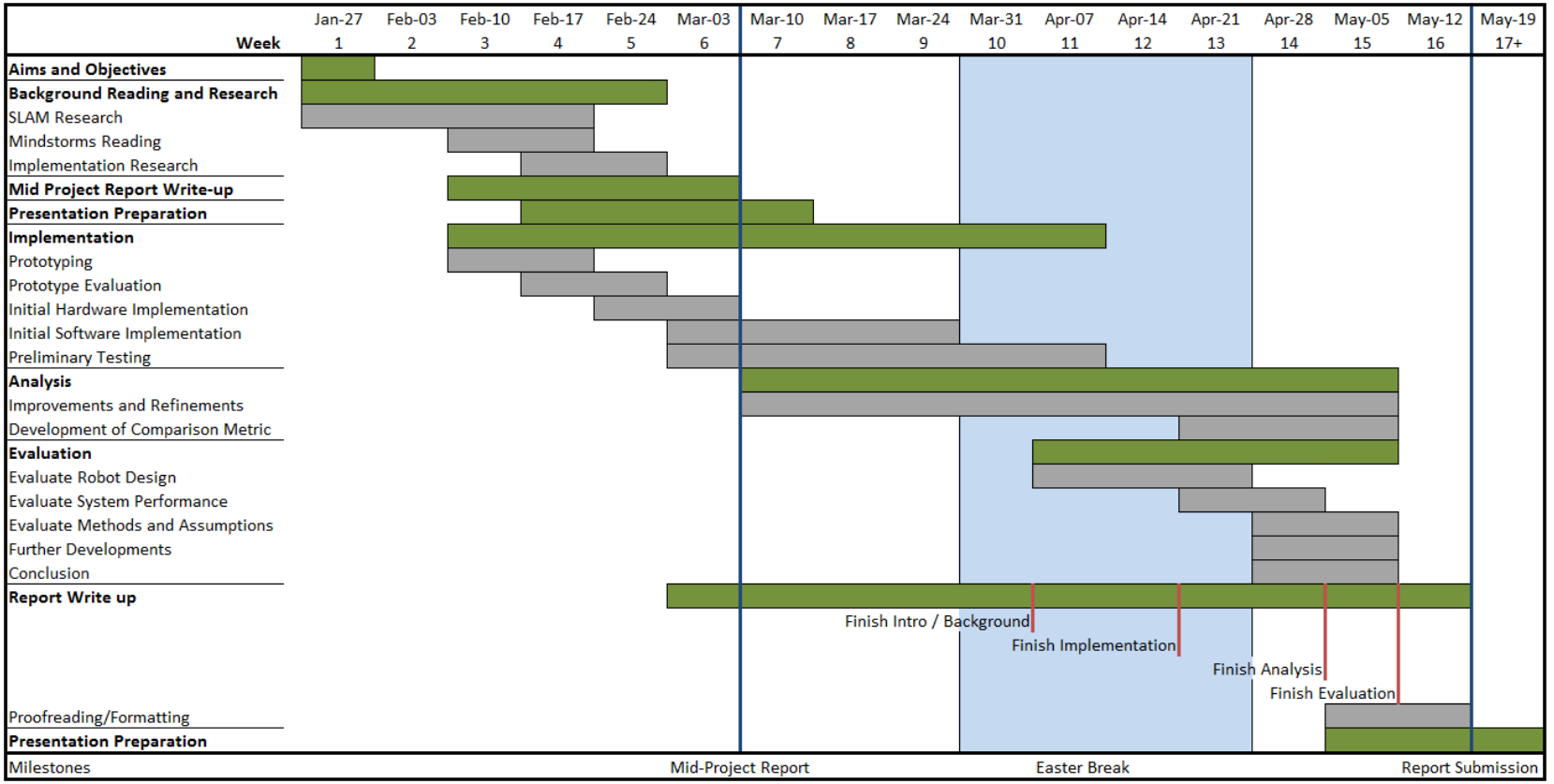
Where as more robust implementations of SLAM assume no prior knowledge of the environment, this makes assumptions about the starting pose of the robot. The solution assumed that the robot began facing either directly at or at 90° to a wall. A more robust application would have allowed any starting pose, and more variety in the environment.

The key limitation of the hardware is the ultrasonic sensor. The sensor was found to have an accurate range of $80cm$, with any measurements above that being largely unreliable. Additionally, the sensor would only return an integer value of between $0-255cm$, and as such the software was only built to be as accurate as $1cm$. Data storage structures were built to store integer measurements, and the grid storage used by the final iteration considered the environment to be a grid of $1cm^2$ units. Even if a more accurate sensor were to be fitted, it would not be able to measure the environment more finely,

only more precisely. However, given the size of the robot and the accuracy of the motors, a granularity of less than $1cm^2$ may not be feasible.

The placement of the sensor was also a limiting factor. Any obstacle or surface in the environment had to be at least $21cm$ tall to be detected; anything less than this was a collision hazard. If the robot did collide with something, the treads protruding at the front of the robot would 'pull' it up the object, leading to the robot tipping over.

Figure 6.1: Actual Project Schedule



Chapter 7

Conclusion

7.1 Extensions

There are many different ways in which this project can be extended. Perhaps the easiest extensions possible are those in which the hardware is upgraded. Provided that the appropriate variables (such as gear rations) are updated, the PC application should be usable without any modification. The easiest hardware modification would be to replace the ultrasonic sensor with a third-party range-finder. This would improve the accuracy of detections and should also reduce the number of outliers, resulting in a more accurate map.

One common test for SLAM systems is the kidnapped robot problem. The current solution relies on localisation by 'assisted odometry' (i.e. odometry with correctional scans), so will not perform well in this test. An alternative approach may place new scans into a second temporary map, and comparing this to the 'master' map to assist with localisation. Adopting a new localisation strategy (such as using Kalman Filters) would also greatly improve localisation. Kalman methods were looked into, but the documentation provided by the LeJOS API was not sufficient to work from, and it was decided that getting the software to a working state was more important than exploring an possibility that have proved inappropriate.

The current PC application can export maps for comparison purposes, a useful extension would be the ability to import maps. The robot could then localise within the loaded environment, and continue mapping it. If an automated way of comparing maps to a ground truth template was implemented, then maps could be compared in real time. A more useful way of presenting maps may be to give a

topographic breakdown of the environment. This would likely require the testing environments to be more complex however.

The implemented navigation strategies always used the data from the most recent scan. A more robust method may assess the whole map and then determine which areas need further exploration. This method would probably also cope better with larger environments, and with annular environments (i.e. environments with disconnected 'islands').

7.2 Conclusion

Looking back to the objectives and requirements set in Chapter 1, I would consider this project to have fulfilled all of the requirements and achieved most of the goals. The only unmet goal was the implementation of an algorithm to compare maps, as the only comparison achieved was manually derived. This aside, I have designed and built a robot to gather data, and produced software that runs a SLAM implementation on the device[1]. I have constructed a suitable environment to test the system in, and I have developed an interface for displaying and processing the data gathered. Without a doubt, the solution is far from perfect, but within the scope of the project, the system has been successful.

[1] See Section 6.6 for a discussion on this

Bibliography

[1] R.A. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, Mar 1986.

[2] H. Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *Robotics Automation Magazine, IEEE*, 13(2):99–110, June 2006.

[3] H.F Durrant-Whyte and J.J. Leonard. Simultaneous map building and localization for an autonomous mobile robot. In *Proceedings of IROS 1991*, volume 3, pages 1442 – 1447. IEEE, Nov 1991.

[4] Hugh Durrant-Whyte, David Rye, and Eduardo Nebot. Localization of autonomous guided vehicles. In Georges Giralt and Gerhard Hirzinger, editors, *Robotics Research*, pages 613–625. Springer London, 1996.

[5] Hansen, John. NBC & NQC homepage. `http://bricxcc.sourceforge.net/nbc` Accessed February 2014.

[6] LeJOS. Advanced motors, sensors and third party hardware. `http://www.lejos.org/nxt/nxj/tutorial/AdvancedTopics/Advanced_Hardware.htm` Accessed February 2014.

[7] Ruth Lonsdale. Demonstrating Simultaneous Localization and Mapping Using LEGO Mindstorms. University of Leeds School of Computing Final Year Project, 2012.

[8] Leonard A. McGee and Stanley F. Schmidt. Discovery of the Kalman Filter as a Practical Tool for the Aerospace Industry. NASA Ames Research Center, 1985. `http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19860003843_1986003843.pdf` Accessed February 2014.

[9] Michele Moro. Appendix A: Lego Mindstorms NXT (Hardware and Software). Teacher Education on Robotics-Enhanced Constructivist Pedagogical Methods (TERECoP) Project. `http://www.terecop.eu/downloads/Appendix_1.pdf` Accessed February 2014.

[10] NASA. Filming of space robot "Jitter" assembled out of legos. `http://www.nasa.gov/mission_pages/station/research/experiments/468.html` Accessed February 2014.

[11] Yves Nievergelt. A tutorial history of least squares with applications to astronomy and geodesy. *Journal of Computational and Applied Mathematics*, 121(12):37 – 72, 2000.

[12] Piri Reis. Map of Europe, from *Kitab-ı Bahriye*, 1513. At: Istanbul: Library of Istanbul University.

[13] Søren Riisgaard and Morten Rufus Blas. SLAM for Dummies. `http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf`, 2005.

[14] RWTH Aachen University. Mindstorms nxt toolbox homepage. `http://www.mindstorms.rwth-aachen.de` Accessed February 2014.

[15] SRI International. SRI AI Center :: Shakey. `http://www.ai.sri.com/shakey/` Accessed February 2014.

[16] The Lego Group. NXT brick store page. `shop.lego.com/en-GB/NXT-Intelligent-Brick-9841` Accessed February 2014.

[17] The Lego Group. Servo motor store page. `http://shop.lego.com/en-GB/Interactive-Servo-Motor-9842` Accessed February 2014.

[18] The Lego Group. Ultrasonic sensor store page. `http://shop.lego.com/en-GB/Ultrasonic-Sensor-9846` Accessed February 2014.

Appendix A

Personal Reflection

This project was quite different from anything I've encountered before. Prior to this, the majority of my degree has been based around exams. There has been summative coursework, but it has been on a much smaller scale, typically around 5% of a module per piece. This work was usually quite structured, and aided by tutorials and/or lectures on the subject. The majority of this work was done in the labs, either through necessity or because I had free time on campus (usually both), so there was often someone else there in the same boat if you got totally stuck. Moving from this highly structured system to the project was quite unusual.

Because of the nature of the kit and the testing environment, I couldn't really work in the labs other than for research or writing up. This lead to me working primarily from home, which was an interesting transition. Where as before there had been a fairly concrete separation of work space (labs) and relaxation space (home), now these two spaces were one. This was a mixed blessing though. Some days I'd find myself working as soon as I'd got up, or working late at night. I was working at unsocial hours that I would never have walked into university to do because my work was there on the desk in front of me. And on the other side of the coin, there were times when I would be distracted and not work. Had I have been in the labs, I wouldn't have faced the same distractions.

The subject matter of the project was also new to me. Most final year projects relate fairly closely to the content of one or more modules from the rest of the degree. However, a couple of robotics projects stood out on the list. Robotics hadn't been mentioned in any great depth, and so the opportunity to get some experience in the area was appealing. This project more so, as not only would I be working with a robot but I would be building one and programming it too. I had used a Mindstorms kit once before, but only with the NXT-G software bundled with it. The software is quite primitive, so I had never considered that the kit could be a genuinely useful tool for developing robots.

Developing software for a robot provides a very different set of challenges that 'regular' software development. Getting started was slow, especially when dealing with the connectivity issues. I spent

at least a week downloading various Bluetooth libraries and moving system files around. Ultimately, I was considering giving up on the Bluetooth aspect completely and changing the project to have the robot just record data and then transferring it off manually for processing. As a last resort my supervisor suggested trying to get a Windows VM working, and run the software though that rather than through the native OS.

Once the Bluetooth libraries were working, work on the software for the robot could begin. Developing software for the robot had its own challenges. Error codes for the software were displayed in a seemingly cryptic fashion, and needed to be run through a command line program. However, because all the software was on the Windows VM, I had to use the Windows Command Line. This was frustrating, mainly because of the subtle syntactic differences.

Perhaps the most annoying problem with developing for the NXT were the logical errors in the code. Because they compiled fine there was nothing obviously wrong, and they only manifest at seemingly random times. Tracking these bugs down was time consuming, as there were so many different places they could crop up. A significant proportion of the time these errors were as simple as a $-$ sign in the wrong place, or something as trivial as that. However, because it of the time it took to re-upload the code after every change, these small bugs occupied a disproportionate amount of time. And I couldn't always tell if a bug had been fixed, as the particular conditions that they occurred under couldn't always be replicated. Developing the visualisation was slow too, as data needed to be gathered in order to test it. In retrospect, I could have implemented a way of loading some test data.

One aspect I am rather disappointed about was my choice to not implement the 'Line of Sight' updating method outlined in Section 6.6. I decided not to implement this relatively simple method in favour of a much more complex strategy. However, I didn't consider the amount of time I had left to work on the implementation before I had to move onto data gathering and writing up. Nor did I anticipate the difficulty of developing and debugging the method as I went along. As with the visualisation, I didn't have a test dataset to work with so had to run the robot to debug anything. I had to abandon the method so I could finish the project. This also set back the analysis phase, as I had planned to use this to compare whole maps too. Even though the 'Line of Sight' suffered from the corner flaw, it would have been better than no method at all. If I could go back, I would advise myself to implement the 'Line of Sight' method despite it's flaws to use as a backup. I ignored it and tried something too complex instead, and it didn't pay off.

Overall though, I'm pleased with how the project progressed. My advice to students attempting something similar would be to be very generous with time estimates for development when relying on hardware. The time required for developing with the robot was significantly underestimated, mainly because of the debugging issues mentioned above. Another thing I would take note of is checking the comparability of hardware early on. I didn't check the Bluetooth connectivity until after prototypes had been developed, which lead to a delay whilst the issues were worked through. Had I have spotted it earlier, it's likely that I would have been able to resolve this whilst working on other aspects of the

project too. Finally, I would say that even though the work can seem slow and frustrating at times, stick with it. Seeing a robot that you've made and programmed from scratch work as planned is worth it.

Appendix B

Record of External Materials Used

The following external materials were used during this project:

- **LeJOS Code Samples** - Some code snippets relating to initiating the Bluetooth connection and opening datastreams were taken from the example code provided with the LeJOS libraries.
- **Java Tutorial Code Samples** - Some code for GUI construction was taken/adapted from Oracle's online tutorials.
- **Eclipse WindowBuilder** - The final iteration of the GUI was built by adapting the previous iteration using Eclipse's WindowBuilder plugin. As such, some of the code in the GUI is machine generated.

Everything else not listed is my own work.

Appendix C

Ethical Issues

Although my specific project is free of ethical issues, there are considerations to be made concerning the wider areas of SLAM research and autonomous robotics as a whole.

Anything involving gathering data from a real environment can have potential issues depending on the nature of the data. Recording the distance to an object from an arbitrary location (arbitrary outside of the system, that is) poses no real issues, but SLAM methods involving video data, or wireless network scanning do have implications. If the data is to be stored in any way, then the footage be subject to privacy issues and surveillance laws. Data involving the recording of wireless networks may be subject to the same considerations, and may also have cybersecurity implications.

Another important aspect to consider is the circumstances in which the robot is used. If an autonomous robot is operating in an environment with people, it could be in a position to cause harm. This may be from seemingly trivial things, such as a robotic vacuum posing a trip hazard, to more serious risks, such as large robots (e.g self-driving cars) making unpredicted movements.

Furthermore, the purpose of the robot may raise issues. There are concerns that just as the mechanisation of factories put labourers out of work, advances in autonomous robotic technologies may lead to human unemployment. For example, a small number of businesses in Japan and South Korea have employed robots to act as guides and waiters. Whilst cases like these are primarily used as marketing gimmicks, it is becoming less unusual to here about robots being used in places like warehouses. Again, at this stage robots like this are still a rarity, but as technology advances and robots become more capable, they will become more common.

Perhaps the biggest issue for autonomous robotics today is military applications. Whether used for confrontational purposes or as utilities, the use of autonomous robots can have massive ethical consequences. Autonomous drones are becoming more advanced, and whilst an operator must take control to engage weapons, some people question how long it will be before the machines are trusted to make such judgements themselves. And in non-combat rolls, robots are being developed for roles

such as minesweeping. As with any software or hardware in positions such as this, any bug, flaw or unexpected behaviour could lead to the endangerment of human life.

Appendix D

Data Output

The following pairs of maps were are the result of a number of trial runs. Each trial run consisted of 10 move phases, and the scan interval was 3°. In each case, the map on the left is the 'raw' data including all detections, and the map on the right is the 'processed' data, where outlying points have been removed. Each map pair was started from a different location and position (e.g. A, North), and the locations are defined in Figure 3.5. The fainter grey dots indicate the locations detections were taken from.

The images have been modified slightly from the .pgm files produced from the application. The command line image modification program ImageMagick was used to make the images more suitable for document display using the command:

```
mogrify -negate -flip -trim -format jpg *.pgm
```

This command inverts the colours the image before trimming out the extra whitespace. LaTeXdoesn't accept PGM files, so the images are converted to JPEG format. Additionally, all maps have been rotated for better comparison.

Figure D.1: Output from (A, North)

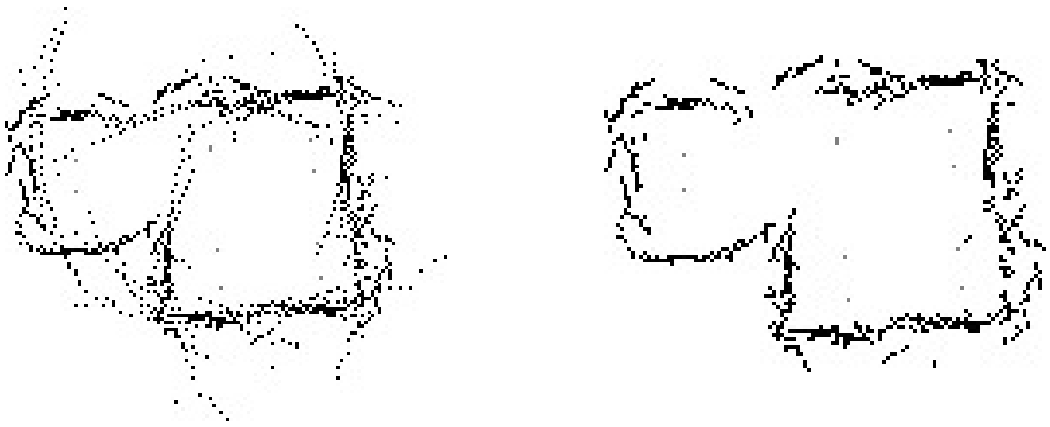


Figure D.2: Output from (A, East)

Figure D.3: Output from (B, North)

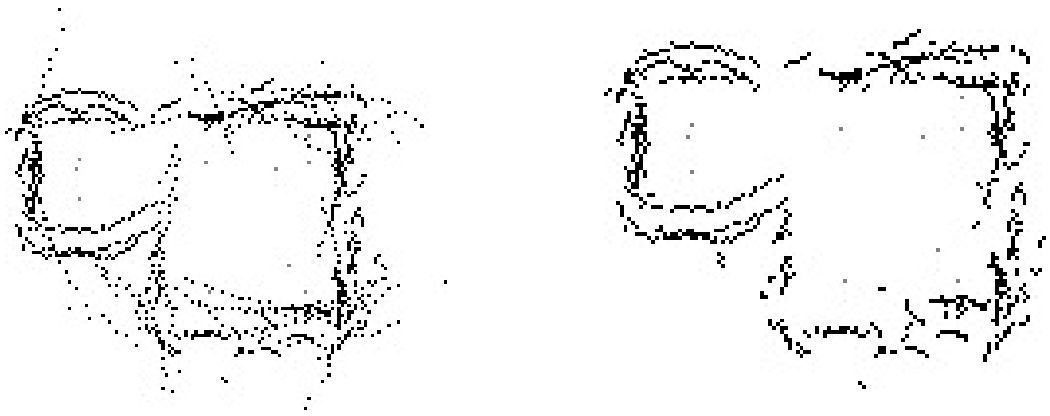


Figure D.4: Output from (B, West)

Figure D.5: Output from (C, South)

Figure D.6: Output from (C, West)

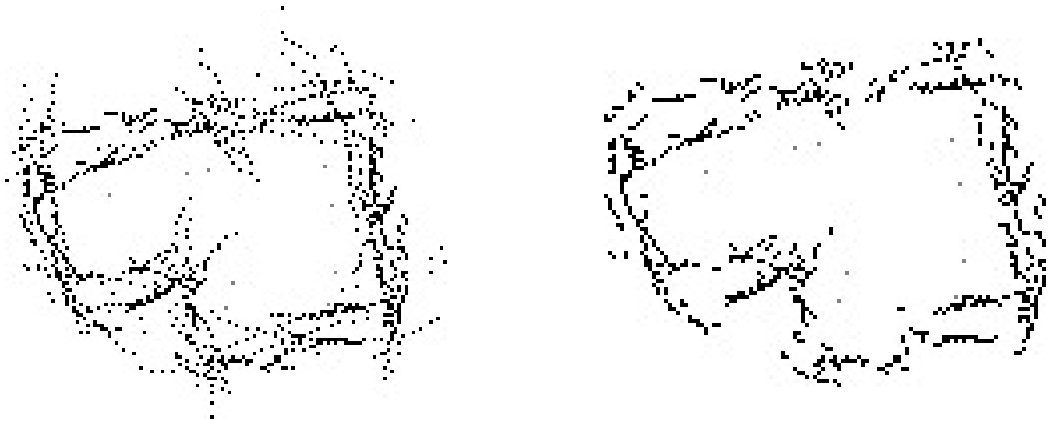


Figure D.7: Output from (D, East)

Figure D.8: Output from (D, South)

Figure D.9: Output from (E, North)

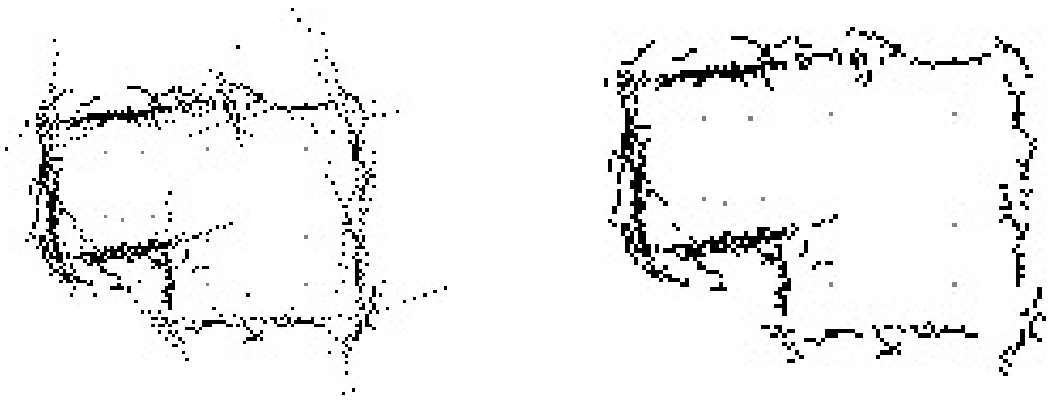


Figure D.10: Output from (E, East)

Appendix E

Data with Ground Truth

The following maps are the 'processed' map from each run from Appendix D, with the 'ground truth' of the environment fitted over them. The fitting was done by hand, as no reliable automated method was found to process them. The centre of the red lines represents the walls of the environment. The line is three squares wide, as this was deemed a reasonable threshold to indicate perfect detections. When calculation the number of detections inside or outside the environment in Section 5.5, detections inside the red line are not counted as being in either category.

For Maps 5 and 6, the data had to be rotated in order to fit the template, and this caused artifacts in the image editing program that had to be removed by thresholding. This may have affected the data points, so any metrics gathered from them will not be as accurate as from other maps.

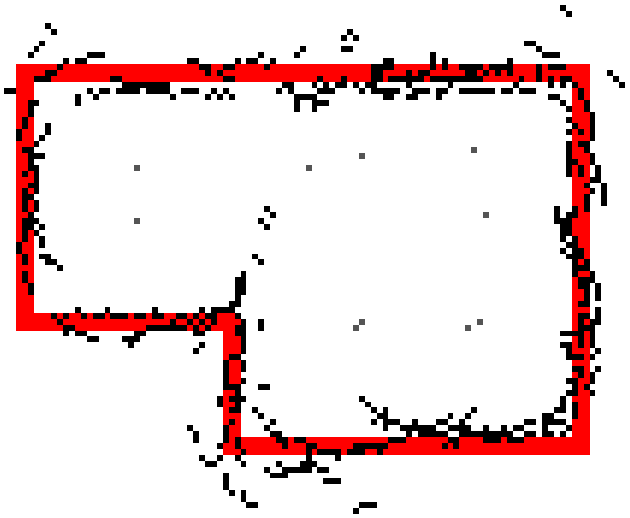


Figure E.1: Processed Output from (A, North)

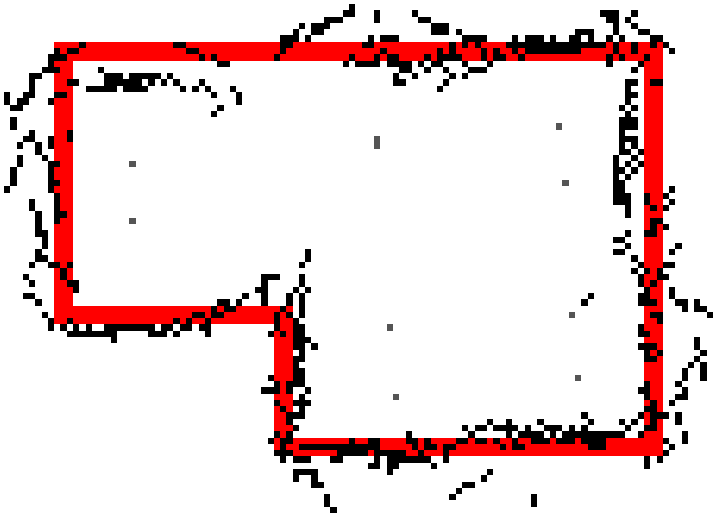


Figure E.2: Processed Output from (A, East)

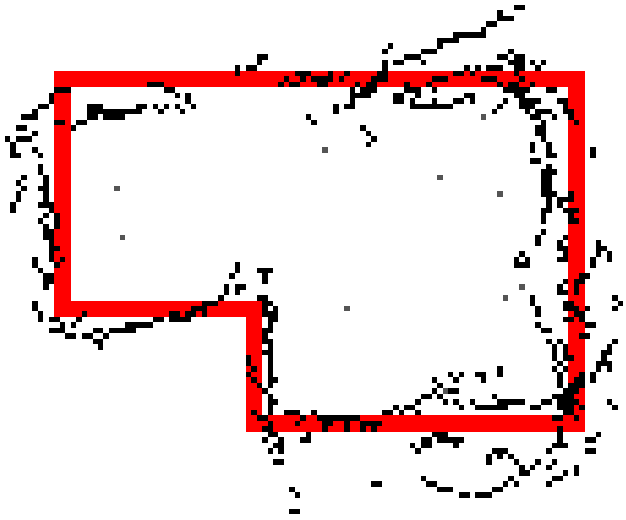


Figure E.3: Processed Output from (B, North)

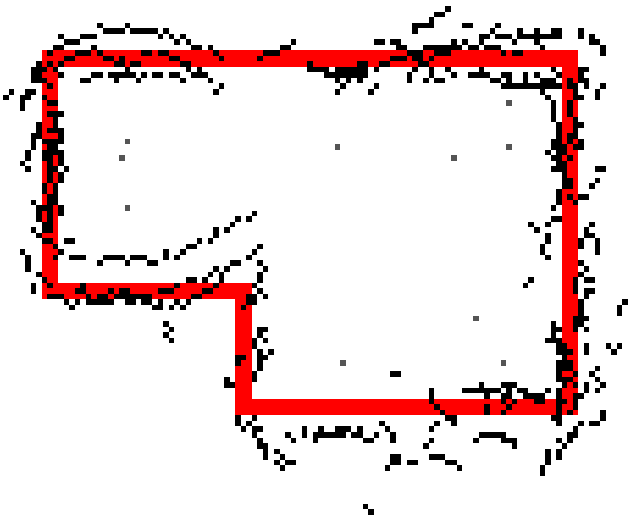


Figure E.4: Processed Output from (B, West)

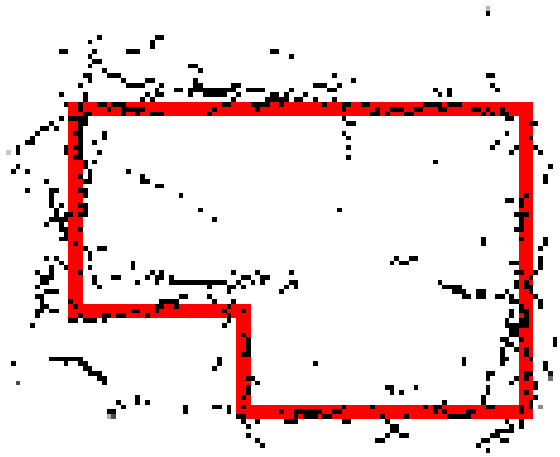


Figure E.5: Processed Output from (C, South)

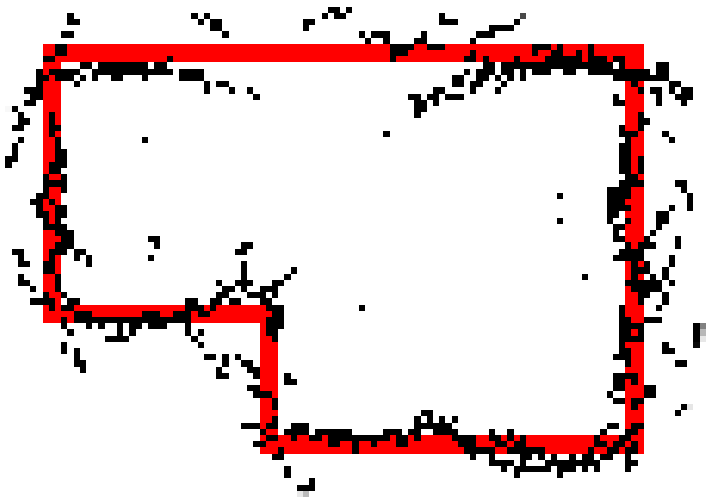


Figure E.6: Processed Output from (C, West)

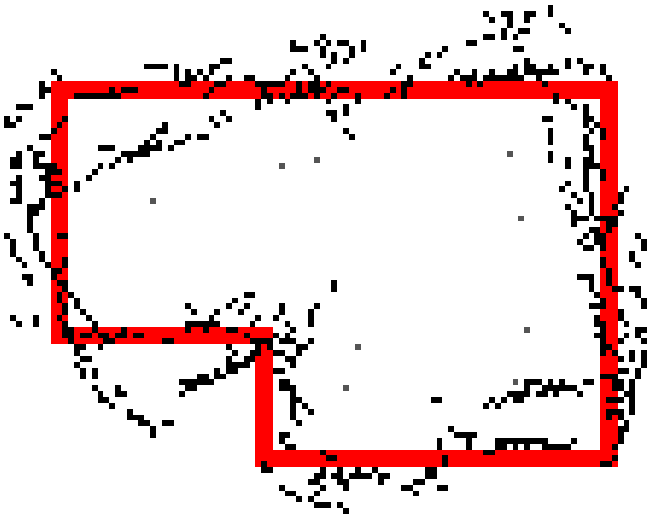


Figure E.7: Processed Output from (D, East)

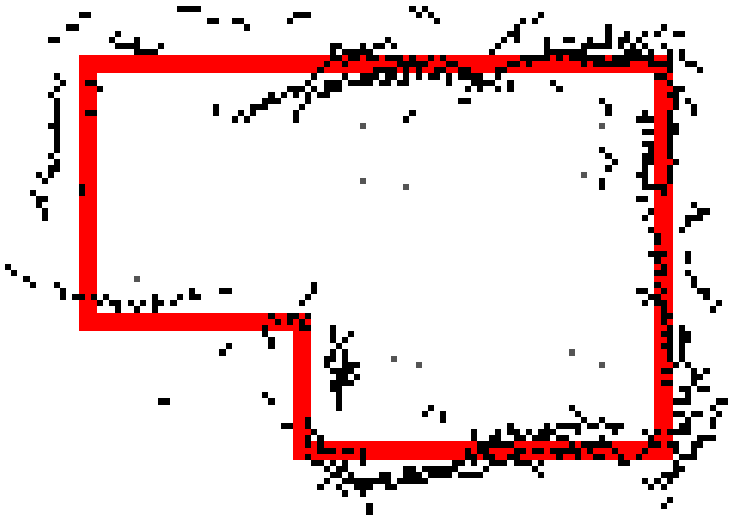


Figure E.8: Processed Output from (D, South)

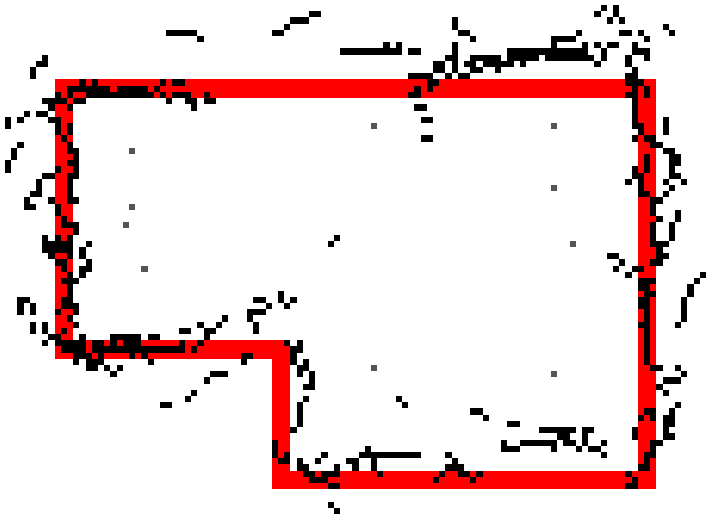


Figure E.9: Processed Output from (E, North)

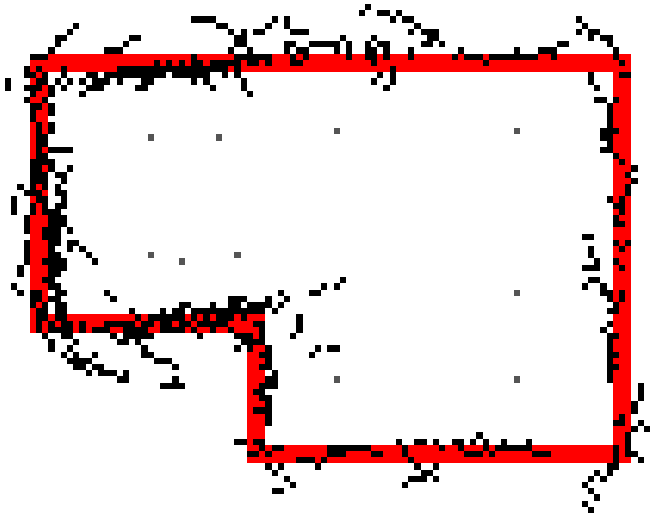


Figure E.10: Processed Output from (E, East)