A Robotics Framework for use in Simultaneous Localization and Mapping Algorithms

Submitted in partial fulfilment of the requirements of the degree of Bachelor of Science (Honours) of Rhodes University

Shaun Egan

Grahamstown, South Africa
October, 2010
Abstract

In the field of robotics, the problem of Simultaneous Localization and Mapping (SLAM) is that which describes the ability of a robot simultaneously to localize itself within an environment and map that environment. While this topic has been the subject of much excellent research in past decades, it is one in which algorithms are continuously re-implemented for use with specific hardware and approaches. This research project is concerned with the implementation of a dual framework for use with either online or offline SLAM, providing common functionality for both such as odometry, landmark extraction and data association using the Python programming language for the online SLAM framework, and Java for the offline SLAM framework. Results show that the use of classes provided by this framework significantly shorten the amount of code required to implement even the most basic forms of pose-tracking and dead-reckoning using ultrasonic sensors. The use of interfaces in the offline SLAM framework allows developers to re-implement classes that may require alternative methods of calculation.
ACM Computing Classification System Classification


D.3.3 [Language Constructs and Features]: Frameworks
I.2.9 [Robotics]: Autonomous vehicles, Kinematics and dynamics, Sensors

General-Terms: Autonomous Robot, Framework, SLAM
Acknowledgements

This thesis has taken many months to produce with support from many people. I would like to thank all those people who have contributed and assisted me in any way to make this research a success. I would like to thank a number of people specifically.

I acknowledge the financial and technical support of this project of Telkom SA, Comverse SA, Stortech, Tellabs, Easttel, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University.

I would like to acknowledge my family, friends and loved ones for all of the support, guidance and funding that they have given me during this time. This opportunity would not have been possible without them and for that I am especially grateful.

Finally, I would like to thank my supervisor, Dr. Karen Bradshaw for all of her hard work in obtaining the hardware required, as well as her highly valued input, guidance and support throughout the project.
# Contents

1 Introduction \hspace{120pt} 7

1.1 Problem Statement and Research Goals \hspace{8pt} 7
1.2 Thesis Organization \hspace{8pt} 8

2 Background \hspace{120pt} 10

2.1 SLAM \hspace{8pt} 10

2.1.1 Representing Pose \hspace{8pt} 11
2.1.2 Organization of a SLAM Algorithm \hspace{8pt} 11

2.2 Tools (Hardware) \hspace{8pt} 12

2.2.1 Encoders \hspace{8pt} 12
2.2.2 Encoders used in odometry \hspace{8pt} 13
2.2.3 Why encoders are needed \hspace{8pt} 13
2.2.4 Sensors \hspace{8pt} 14

2.3 SLAM Process \hspace{120pt} 15

2.4 Range and Odometry Data \hspace{8pt} 16

2.5 Landmarks \hspace{8pt} 17

2.5.1 Landmark Extraction \hspace{8pt} 19
## CONTENTS

2.5.2 Spike Landmarks ........................................... 19
2.5.3 RANSAC ..................................................... 20
2.6 Data Association ............................................. 22
  2.6.1 Triangulation Based Fusion .............................. 22
2.7 Localization ................................................... 25
  2.7.1 The Extended Kalman Filter (EKF) ...................... 25
  2.7.2 Monte Carlo Localization ............................... 26
2.8 Failure ......................................................... 27
  2.8.1 Failure Detection ....................................... 27
2.9 Loop Closure ................................................. 28
2.10 Cooperative SLAM ........................................... 29
2.11 Robotics Frameworks ....................................... 30
2.12 Summary ..................................................... 30

3 Implementation .................................................. 31

3.1 Hardware Setup ............................................. 31
  3.1.1 Final Configuration .................................... 32
3.2 Basic Hardware Testing Software Development ............. 33
  3.2.1 Initial Problems ....................................... 33
  3.2.2 First Program ......................................... 34
  3.2.3 Calibration ............................................. 34
  3.2.4 Ultrasonic Sonar Sensor Testing ....................... 36
3.3 Basic Framework Overview .................................. 38
CONTENTS

3.4 Online SLAM ......................................................... 39
  3.4.1 CommonUtils .................................................. 39
  3.4.2 CommonUtils Organization for Online SLAM ................. 39
  3.4.3 Point .......................................................... 39
  3.4.4 Pose .......................................................... 40
  3.4.5 Configuration ................................................. 40
  3.4.6 ConfigurationFetcher ......................................... 41

3.5 Offline SLAM (Java) .................................................. 41
  3.5.1 CommonUtils Organization for Offline SLAM ................. 41
  3.5.2 Point, Pose and Configuration ................................ 41
  3.5.3 ScanStep ...................................................... 42
  3.5.4 Landmark ...................................................... 42

3.6 Odometry Implementation ........................................... 44
  3.6.1 Straight Line Odometry ....................................... 45

3.7 Triangulation Based Fusion ......................................... 48
  3.7.1 Triangulation .................................................. 48
  3.7.2 Associating Points with Landmarks ......................... 50

3.8 Python Variations .................................................. 52

4 Results ............................................................. 54
  4.1 Introduction ..................................................... 54
  4.2 Online SLAM ..................................................... 54
  4.3 Offline SLAM ..................................................... 56
  4.4 Results .......................................................... 59
5 Conclusion

5.1 Project Summary .................................................. 61
5.2 Outcomes of Objectives ........................................... 61
5.3 Extensions .......................................................... 62

Bibliography .......................................................... 62

A Algorithms .......................................................... 66

A.1 FT_Odometer ....................................................... 66

B Programs ............................................................ 71

B.1 Sensor Monitor Program ........................................... 71
B.2 Offline SLAM pose and landmark tracking .................... 73

C Mathematical steps performed by the algorithms implemented 76

C.1 Odometry .......................................................... 76
C.2 TBF ................................................................. 77

C.2.1 Triangulation ...................................................... 77
C.2.2 Point refinement .................................................. 78
List of Figures

2.1 The Fischertechnik robot that is used in this project. It has two ultrasonic sensors, two encoder motors and a Bluetooth controller allowing the robot to be controlled by a computer. ........................................ 12

2.2 The Fischertechnik ultrasonic range finder. It has an arc width of 60 degrees and a maximum range of 4 meters, reporting in a resolution of 1 centimeter. 14

2.3 High level view of the SLAM process. (Adapted from [19]) .................. 16

2.4 The outcome of systematic errors. The white arrow indicates the position according to the odometry data while the green arrow indicates the actual position of the robot due to drift. ........................................ 17

2.5 The odometric position of the robot, the actual position of the robot (green) and the position of a landmark ................................. 18

2.6 The result of the EKF algorithm’s correction of the robot’s position after performing a range scan on the nearby landmark. The corrected position is shown in orange. ........................................ 18

2.7 Spike landmarks with dots representing chair legs. (This image is taken from [19]) ................................................................. 20

2.8 Results of an example range scan ............................................. 21

2.9 Associations of range values with a line determined by the RANSAC algorithm using the least squares approximation. ....................... 21

2.10 Representation of what is indicated by a single sensor range scan. The small dot indicates the actual position of the landmark, but the only output of a scan is the range and the arc width of that sensor. ................... 23
2.11 The TBF process using two range scans of the same object from different positions, where the scans overlap is where the landmark is. ....... 24

3.1 Output of the calibration program. ......................... 35
3.2 The code used to determine the variables used for calibration ........ 35
3.3 Organization of the CommonUtils package ..................... 40
3.4 Organization of the CommonUtils package for Offline SLAM ........ 42
3.5 UML representation of the ScanStep class ...................... 43
3.6 The code to initialize the odometry algorithm ................... 45
3.7 Odometry algorithm ........................................ 46
3.8 Code for determining the distance between the two sensors .......... 49
3.9 Code for determining angle between the robot and the detected object ... 49
3.10 Code for checking if a point is actually visible to a specific sensor ...... 50
3.11 Code for determining the position of a sensor .................. 52
3.12 This method associates as many scans as possible from the sliding window with the current scan ....................................... 53

4.1 Simple Offline program analyzing encoder data to perform odometry updates 57
4.2 Output of the odometry program .................................. 57
4.3 Offline odometry and triangulation program that does not use data association ......................................................... 58
4.4 Offline odometry and triangulation program using data association ...... 60
Chapter 1

Introduction

Simultaneous Localization and Mapping (SLAM) is a problem within robotics that has been researched substantially in the past couple of decades [25, 24, 28, 15]. The idea is that for a robot to create a map of its environment successfully, it needs to be able to localize itself within that environment. Secondly, for a robot to be able to localize itself within an environment, it needs a map of sufficient quality. In short, to maintain an acceptable level of accuracy, both of these processes need to be executed at the same time [28, 7]. This problem is not an easy one to overcome, as each of the two goals of a SLAM algorithm requires the other to work.

While this problem of duality is one which has been largely overcome, it is still a difficult one. Several algorithms have been developed to provide solutions to various minor steps within the major goals, yet no simple, or unified tool set to cover these exists. For each new SLAM application or research project, an entirely new program is written to deal with the specifics of the robot’s platform.

1.1 Problem Statement and Research Goals

The goal of the research project to build two versions of a framework that will allow for the development of an autonomous robot framework with specific application to a Simultaneous Localization and Mapping (SLAM) algorithm through a unified set of interfaces and algorithms that are common to many SLAM algorithms.

The first version framework should provide algorithms for the calibration and calculation of odometry data, extraction of landmarks from ultrasonic range finding data and data
association for those landmarks. The framework is not primarily concerned with the actual control of the robot as it is built on top of another framework that already provides this functionality. These algorithms run in online mode, that is, they run as the robot is moving through its environment.

The second version of the framework works as an offline implementation. This means that it analyzes data recorded by a robot as it moves through its environment. This allows for the debugging of programs on a known input, enabling the developer to compare different approaches to the same problem.

The following is a list of the goals outlined by the problem statement:

**Primary objectives**

- Create an online SLAM framework to supply functionality common to many SLAM algorithms used on the fly by the robot as it moves through its environment.
- Create an offline SLAM framework to supply the same functionality as the online framework, but with the intention of use in applications that analyze data from a pre-recorded run by the robot through its environment.
- Functionality included in the framework should be a set of classes and interfaces that allow the storage of sensor and motor information in an Object Oriented fashion.
- This functionality should include odometry, landmark extraction and data association classes and interfaces.

**Secondary objectives**

- To build implementations of state update algorithms such as an Extended Kalman Filter and a Monte-Carlo filter.
- Extend the functionality to include the use of multiple robots within the framework.

### 1.2 Thesis Organization

The remainder of the thesis is divided into 4 chapters which are briefly described in this section.
Chapter 2 introduces the topic of Simultaneous Localization and Mapping, the various aspects and problems related to SLAM and several approaches for solving these problems.

Chapter 3 describes the implementation of the framework, the problems encountered and the reasons for design decisions within the framework.

Chapter 4 discusses the tests used and compares the expected results and the obtained results, as well as describing and discussing differences between the expected and obtained results.

Chapter 5 presents the results of the project and possible extensions to the research.
Chapter 2

Background

This chapter describes the SLAM process as a whole and then continues to describe various parts of the process in the form of specific problems relating to SLAM. It also describes specific algorithms that try to solve these problems and the specific situations in which these algorithms are used. It by no means discusses every algorithm available for each of these problems as there are simply too many to mention here. It is more of an introduction to the type of algorithm used and common methods for overcoming a variety of problems.

2.1 SLAM

Simultaneous Localization and Mapping (SLAM) is a family of algorithms used in robotics that is primarily concerned with creation and maintenance of maps (co-ordinate systems) of previously unknown static environments and then using the map to determine its position [6, 25, 8]. This process is often called Robot Pose Tracking in many of the papers. The benefit of using a map in conjunction with other means of recording movement is that it allows the algorithm to correct positioning estimates and detect and recover from errors regarding sensor data. It is also concerned with, on a secondary basis in this research area, navigation using the map.

A complete solution to the SLAM problem would allow the construction and programming of truly autonomous robot [6]. While SLAM largely has working implementations, it still requires the solution to many issues required for a more general SLAM implementation [6].
The algorithms can be more easily understood by characteristic questions to which they try and answer: "Where am I?", "Where am I going?" and "How should I get there" [13]. Localization is captured by the first question and is the process by which the robot uses sensory data and the map to determine its position on the map, and is the most directly applicable to SLAM as a whole. This approach is referred to as stochastic mapping in both [20, 30].

### 2.1.1 Representing Pose

One approach to representing landmarks in a coordinate system is to think of it as a database where the robot’s position is represented by the state matrix at time $t$

$$P(t) = \begin{pmatrix} x(t) \\ y(t) \\ \theta(t) \end{pmatrix}$$

The $x$ and $y$ represent the Cartesian position of the robot from its starting position (global co-ordinate) and $\theta$ represents its orientation in relation to the $z$-axis (its rotation) [20].

### 2.1.2 Organization of a SLAM Algorithm

SLAM is considered a difficult problem within robotics and the reason for this, according to [13], is that being able to extract accurate sensor data and odometry data and then being able to associate this data with known landmarks (or geometric beacons as they are called in [13]).

Common to all SLAM implementations are the smaller parts which make up the algorithms including: Landmark extraction, data association, state estimation, state update and landmark update [19]. The algorithms differ mainly in the approach and implementation of these smaller parts [19] and extensive research has been conducted in the past few decades relating to these problems. What these separate parts are responsible for and how they interact will be discussed later in this thesis.
2.2 Tools (Hardware)

The first and most important part of the development of a SLAM algorithm is the robot that is to be used. Important factors to SLAM are the ability of the robot to do some form of range measurement and to be able to report odometry data (data relating to the actual position of the robot through the turning of its wheels.)

Figure 2.1: The Fischertechnik robot that is used in this project. It has two ultrasonic sensors, two encoder motors and a Bluetooth controller allowing the robot to be controlled by a computer.

The use of odometry data, known as the plant model, is concerned with how the vehicle’s position changes over time as a result of the wheels turning through control signals and through error such as drift and drag [13]. Sensor data is part of the measurement model and measured as relative ranges from the robot’s current position and orientation [13].

2.2.1 Encoders

The first and most basic requirement of SLAM is the ability for the robot platform to supply an estimated position of itself through the movement of its wheels. This type of pose tracking is known as odometry and is useful because it is inexpensive to calculate in real time [2].
2.2. TOOLS (HARDWARE)

Odometry, unlike the tracking of landmarks which works in absolute positioning co-ordinates, works by relative positioning [2]. It is known as relative positioning because the movements of the wheels are recorded in relation to the starting position of the robot, not the position of the robot in world co-ordinates. Once the robot starts tracking some landmarks, a landmark association algorithm such as TBF [26] and a localization algorithm such as EKF [26], will try to determine the position of the robot in terms of world coordinates, thus translating the odometry data from a relative position, to an absolute world position.

2.2.2 Encoders used in odometry

Encoders are a special type of motor used for odometry estimation as they can record the amount of rotation that a wheel has performed. Firstly, they are stepper motors which require impulses from the controller to perform movement in either a forward or backward direction. The problem with simply turning the motors for a specific amount of time and using a simple calibration is that certain factors impede or improve the rotation of the motor.

2.2.3 Why encoders are needed

One of the most obvious of these factors is battery levels of the robot. If the robot were using this approach to record movements, it would only work for a short period of time while battery levels were at a known level of charge. They quickly drop which causes the motors to spin less quickly, and thus the measurement of distance rapidly grows in error.

Encoder motors overcome this problem by tracking the actual rotation of the motor output shaft. There are several different types of implementations used to achieve shaft encoding. Mechanical encoders use magnets or metal brushes around the shaft in regular known positions that, when moving past a specific sensor or contact that is stationary within the motor, generate a 'tick'. These ticks are then counted several times a second to provide an amount of ration in terms of known angular ticks [18].

Another approach called Optical encoders use a disc with a specific pattern or pattern of wholes which rotate with the shaft. This is again interpreted with a static sensor that can either interpret the visible pattern on the disc, or count the holes through the disc as they pass by, again recording ticks.
2.2.4 Sensors

The second important requirement for a functioning SLAM algorithm is a method of determining range through its sensors. This data is used in combination with the odometry data by the algorithm to localize itself and to generate new map readings.

There are several types of sensor that can be used for this purpose. The first and most prominent these days is the laser sensor. They can perform very accurate measurements at the speed of light, and their output, an array of range measurements through a specific angle, is easily processed. The main problem associated with laser range finding sensors is the price [19].

Another option, which was widely used previously, is the sonar scanner. These work by using an ultrasonic ping and measuring the time taken for the wave to be recorded again by the scanner. This method of range measurement is known as Time of Flight (TOF). Laser scanners may also employ this type of range measurement [30]. The advantage of sonar scanners over laser scanners is their very low price [19] and the fact that they work under any lighting conditions [19, 30].

![Fischertechnik ultrasonic range finder](image)

Figure 2.2: The Fischertechnik ultrasonic range finder. It has an arc width of 60 degrees and a maximum range of 4 meters, reporting in a resolution of 1 centimeter.

However, there are several problems associated with sonar scanners. The first and foremost is the degree of width in the line of measurement [19, 3] and the fact that PRM sonar will only report the range of the nearest object within its scan arc [12]. While laser scanners generally scan in a width of 0.25 degrees, sonar scanners scan in a much wider line width. Authors have various measurements of this width, but most agree on a range
between a width of 22.5 degrees up to 30 degrees [3, 19]. This problem can be overcome with techniques such as Triangulation Based Fusion (TBF) [26].

TBF is a method that tries to increase the certainty and accuracy of less accurate sonar scanners by using multiple scanners which scan an area that is potentially visible to more than one scanner, and then combine the information found by these scanners to increase the certainty of any detection to an acceptable level [20].

There are several other problems, which have been highlighted in [30], including multiple reflections (the ping is reflected by multiple objects before reaching the receiver), crosstalk (when a receiver receives a sound wave sent by a different sensor), a limited sensing range when trying to detect small objects, and the speed of sound, are associated with sonar scanners. Although the speed of sound is relatively slow, and that the robot may move before receiving the ping again, the actual speed of the robot for our purpose means that the actual distance travelled by the robot may be ignored.

The third option is to use a vision based range finding solution. This option gives the ability to use surfaces to localize the robot and to extract 3D information from the images [30]. Unfortunately though, this method of range detection has several problems, also highlighted in [30], such as the high computational overhead required in extracting information from the images, its extreme sensitivity to lighting conditions and the fact that it is still a relatively expensive option when compared to sonar scanners.

In [19] and [30] it is suggested that laser scanners are by far the best, and most widely used, approach to range measurement in SLAM problems. While this is correct, sonar has been used successfully in many other applications, see [13]. Also, the costs of laser scanners may be far out of the reach of many researchers, and the success achieved by others shows that sonar can be used with an accuracy that is acceptable for most low cost applications and research projects.

2.3 SLAM Process

As previously mentioned, SLAM consists of a number of smaller objectives, which, when used together, can be used to localize the robot [19]. The process relies on range and odometry data to achieve this goal, but odometry data can be erroneous for several reasons. Odometry ”drift” can occur as a result of two factors: imprecise assembly or design of the robot (systematic errors) and external factors such as wheel slippage.
2.4. RANGE AND ODOMETRY DATA

on different surfaces (non-systematic errors) [30]. Although systematic errors can be compensated for, they cannot be eliminated completely [30]. For this reason, range scans are used and landmarks are extracted to correct the position of the robot [19, 16]. An EKF (Extended Kalman Filter) is often used to correct of the robot’s position based on the extracted landmark information [19]. The EKF is explained in greater detail in section 2.7.1.

After the robot moves a new scan is done, and then landmark data is extracted. This data is used to correct the position of the robot (localization). Any landmarks that have not previously been observed are stored for re-observation later. During this entire process the EKF keeps track of the uncertainty of any observation and position of the robot [19].

Figure 2.3: High level view of the SLAM process. (Adapted from [19])

2.4 Range and Odometry Data

The question "Where am I" is initially partially answered by the ability of the robot to scan its surroundings and extrapolate landmark information. This is usually done with a range finding sensor such as a laser scanner or a sonar scanner. These can be used to find ranges of objects within a specific angle around the robot. However, they are not completely accurate and cannot answer the question on their own [3, 4].

The second input used for localization is the odometry data. That is, the estimated position of the robot obtained from measuring the movement of the wheels. Again, this
2.5. **LANDMARKS**

is subject to error and merely serves as an initial "guess" as to the position of the robot [19].

![Figure 2.4: The outcome of systematic errors. The white arrow indicates the position according to the odometry data while the green arrow indicates the actual position of the robot due to drift.](image)

The only consideration to be made when fetching odometry and range data is that of timing [19]. The odometry data will not be relevant if the range data sampling is taken some time later as the robot may have moved. The best solution is to simply query both sets of information at the same time so as to avoid this kind of error [19].

### 2.5 Landmarks

Landmarks are geometric entities that can be recorded by rangefinders and are used by the EKF algorithm to localize the robot [19]. A landmark can be anything from the corner between walls or a door, to obstacles in the path of the robot.

Landmarks are interpreted from range finding data with several different approaches available depending on the type of landmark to be extracted [19]. The EKF algorithm uses this data to recognize previously seen landmarks for localization. Landmarks are used to update the robot state and are recorded in a co-ordinate system (the map) for later passes through the same area. This map may also be used to generate a human-readable geometric map.
Figure 2.5: The odometric position of the robot, the actual position of the robot (green) and the position of a landmark.

Figure 2.6: The result of the EKF algorithm’s correction of the robot’s position after performing a range scan on the nearby landmark. The corrected position is shown in orange.
Landmarks should be unique enough to be observed and not be miss-associated [19]. They should also be far enough apart to avoid being interpreted as a single landmark [19].

The robot should not spend extended periods of time without any visible landmarks [19]. This could cause the algorithm to reach impractical levels of uncertainty or even to lose track of its position entirely.

Finally, landmarks should be stationary [19]. If a landmark, on subsequent observations, is not in the same place where it was first observed, the algorithm will not be able to determine its position accurately within the environment.

### 2.5.1 Landmark Extraction

Landmark extraction may be achieved via a variety of methods. This is a critical part of the algorithm as it is used as an input to the EKF algorithm. The method used is generally determined by the type of landmark to be extracted in the environment and the type of sensor used to detect these.

With a laser scanner, landmark extraction can be done easily as the scanner typically returns an array of ranges through a 180 degree arc at 1 degree resolution. This makes identifying walls and point landmarks easy through methods discussed later in sections 2.5.2 and 2.5.3.

With sonar scanners it can be more difficult. When they work on a TOF method, the pulse expands on its trajectory in up to a 30 degree angle and the closest object to reflect the pulse is returned as the closest range. This means that the range may be anywhere along a 30 degree arc in front of the sensor [30].

As mentioned earlier, this problem can be overcome through the use of TBS techniques. TOF means that a range probe will generate an arc of possible positions for the object that generated the range reading. Should the robot move and then detect the object again, it generates a new arc. The intersection of the two arcs yields the actual position of the landmark [30, 26, 29].

### 2.5.2 Spike Landmarks

Spike landmark extraction is a simple algorithm concerned with landmark extraction from laser or sonar scan range data. In scanning systems where scans yield multiple
values within a certain angle of scanning, this algorithm tries to find extreme differences in the values read by the scanners [19]. This happens when the distance measured at one angle is largely different to the distance measured at the next angle. This indicates that there is a geometric change between the angles, which can be interpreted as a landmark.

![Spike landmarks with dots representing chair legs.](image)

Another approach to finding landmarks is to try and find spikes within the readings. This is done by using values in triplets. The second value is subtracted from the first value to yield a number, which is in turn also subtracted from the third number to yield a second result. The sum of these two results yields a spike [19].

Due to the method in which this extraction algorithm works, it has an inherent flaw. It requires that the environment be highly populated with landmarks as it requires a high degree of change between range scans, which means that it will not work in an environment with either zero or smooth edges [19].

### 2.5.3 RANSAC

Random Sampling Consensus is a method of landmark extraction that works by trying to identify lines from range scans. This can be particularly useful in indoor environments as walls are easily identified by this method [19].

RANSAC associates range readings with lines and is known as a form of scan matching [21]. This is done by taking a random sample of range values from an unassociated scan and then using a least-squares algorithm to try and find a line that best fits these readings. Once the line has been found, the algorithm will count the number of range values that match that line and if this count is above a consensus threshold, these values are then associated with a line [19].
Figure 2.8: Results of an example range scan

Figure 2.9: Associations of range values with a line determined by the RANSAC algorithm using the least squares approximation.
2.6 Data Association

Once landmarks have been extracted, if it is the first time they have been encountered, they are stored together with an observation count. Every time they are subsequently observed, their observation count is increased until it reaches a threshold at which point the landmark is added to the map so as to help deal with the problem of random sensor errors. This is done so that they may be used later on as objects for the robot to recognize and to localize itself within the state map. This process of interpreting sensor data and trying to map it against already recorded landmarks and is known as the problem of data association [19, 15].

Three of the most common problems identified in [19] are:

1. Landmarks may not be re-observed at every scan.
2. A landmark may only ever be observed once.
3. Landmarks may be incorrectly associated.

The first two issues should not be encountered often as they are indicative of bad landmarks, i.e. landmarks that should not be seen as such [19]. The third problem, incorrect association, is possibly the worst problem to be encountered as the EKF algorithm is particularly susceptible to this kind of error [6]. When this problem occurs, the robot will not be able to localize itself, which means that its ability to create the map or navigate through it would be severely compromised.

TBF employs a method of data association by comparing distances between landmarks on the map. Other than TBF, there are several other techniques employed to achieve data association, such as Combined Constraint Data Association (CCDA) which is described in [1].

2.6.1 Triangulation Based Fusion

Triangulation Based Fusion (TBF) is an algorithm that tries to create useful data from sonar scans. As already mentioned, sonar scans have a very wide arc width, making simple estimation of a landmark’s bearing impossible from a single scan input. Heading is an
Figure 2.10: Representation of what is indicated by a single sensor range scan. The small dot indicates the actual position of the landmark, but the only output of a scan is the range and the arc width of that sensor.

important part of the SLAM process as it allows the localization algorithm to measure angular change in the robot’s pose.

TBF tries to correct this by using overlapping scans to triangulate the bearing of the landmark. Two separate scans from different sensor positions that are recording the range to the same object generates two arcs that intersect each other at the point where the landmark is.

Several problems exist with this though, the first of which is sensor range error. The Fischetechnik sensors that we have only report ranges in a resolution of centimeters. This is to cover up the associated error with ultrasonic Time of Flight (ToF) range finding.

A second problem with this kind of point landmark heading finding is that points may not be easily defined from different positions as the landmark may not be (and usually is not) a perfect measurable point. It will most likely have several (relatively) large surfaces, all measurable by a sonar range scan.
Another problem is known as unstable intersections. The illustration in Figure 2.11 would actually result in what is known as an unstable intersection, due to the small angle between the scans, with the relatively large error from ultrasonic scanners, the width of the error area is quite large. This makes it quite important to try and achieve scans of the same landmark from different perspectives, so as to create much smaller intersection areas.

TBF tries to overcome these problems by incorporating several range scans of the same landmark and taking the mean of the intersections to determine the most likely actual position of the landmark. The most accurate way of achieving this would be to use all scans performed by the robot and try to triangulate with the most current scan.

While this would achieve the most accurate output possible, with the highest chance of correct data association, it would require a massive amount of processing time as the number of scans increases. TBF, on the other hand, uses the concept of a sliding window. A sliding window uses a predetermined number of the most recent scans for triangulation. The window size is implementation specific, but its size will stay static. Every time a new scan is performed, it will ‘push’ the oldest scan out of the window, leaving only the $n$ most recent scans. This keeps the computational complexity down, while still producing an acceptably accurate output.
TBF also associates landmarks within a certain range of each other with the same landmark and updates its position accordingly, to accommodate odometry drift.

2.7 Localization

Localization, state update and mapping are all done through an algorithm that uses the current position estimate, given by the previous estimates and combines it with the motion and sensor model inputs. This is done by using the current odometry output and sensor readings. There are several different algorithms available to achieve this, two of which are highlighted here: the Extended Kalman Filter and Monte-Carlo Localization. There are several other approaches to localization, being the subject of much research. A very interesting approach is documented in [9] uses WiFi signal strength measurement techniques.

2.7.1 The Extended Kalman Filter (EKF)

When the robot moves it is subject to non-systematic errors such as drift and drag from different surfaces. This can cause the robot to report a position that is incorrect to the algorithm. As discussed above, this can seriously impede the robot’s ability to localize itself.

The role of the EKF is considered to be key to SLAM processes within which it is implemented [19]. It uses the robot’s odometry readings as a basic starting point to determine the robot’s position. Thereafter it will uses inputs from the sensors, in other words, it uses the landmarks that it can detect to correct the position of the robot. This is done by an estimation process that takes into consideration the robot’s odometry readings and landmark information through a Kalman filter.

Kalman filters, on their own, cannot be used for this purpose as they work on state estimation alone [19]. Within SLAM, though, they have been modified to work with the map updates that are generated through the algorithm, landmark and odometry data. This has led to the EKF (Extended Kalman Filter).

At this point, there are only two major goals left for the SLAM algorithm to achieve; first updating the state using odometry data and landmark data, and second updating it with new, previously unseen landmarks.
The robot’s position, in a 2D environment, is stored as three values to represent its x and y position as well as its orientation (rotation). This can be represented as \((x, y, \theta)\) and is known as the robot’s ”state”.

Updating the robot’s state using odometry data is straightforward using the following approach:

\[
\text{Estimated State} = (x + \Delta x, y + \Delta y, \theta + \Delta \theta)
\]

Updating the state using re-observed landmarks corrects the positioning errors from the odometry reading. The difference between the robot’s estimated position and its actual position is known as the innovation [19]. Each time the state is updated in this way, the certainty of the state is updated simultaneously. This covariance is continuously changing, but in the event that the loop is closed (the robot reaches its initial position and comes to a known state) the state becomes far more certain.

Finally the map is updated using newly observed landmarks. This is done by using information about the robot’s current state and the relative position of the landmark to the robot and to other landmarks [19].

### 2.7.2 Monte Carlo Localization

An alternative approach to pose tracking instead of using EKF is known as Monte Carlo Localization (MCL) and other names such as Carlo filter and Condensation Algorithm [23, 5]. This general method is known as a particle filter and consists of two steps: prediction phase and update phase [23, 5].

In the prediction phase the set of particles computed in the previous iteration of the algorithm is used to predict the current position of the robot. It takes into account only the previous time step computed most likely position and applies the motion model to it through the an input signal [23].

In the update phase a measurement model is used to apply information gathered from the sensors [23]. This process narrows down the possible positions in the form of particles and creates a new set of particles for the next iteration [23].

The authors of [7] avoided using a traditional approach to a particle filter based solution for several reasons.
The basic problem is that the hidden state is actually both the robot pose and the map itself. An important consequence of this problem is that all observations are no longer compared to a single map, which is presumed to be correct. Instead, the observations are compared against an incomplete and possibly incorrect map, identified with the particle in question. The map itself is created by the accumulation of observations of the environment and estimates of robot positions.

2.8 Failure

In any application, working in unknown environments, failure is a factor that should always be taken into account. In SLAM the three primary reasons for failure according to [30] are incorrect data association errors, map slippage and unexpected movement of the robot.

Data association errors can happen when landmarks are too close together causing them to be incorrectly identified or identified as one object. This together with the uncertainty of odometry readings can cause the EKF algorithm to incorrectly localize itself, and then update the system state, causing the error to cascade across the entire map [30, 26].

This type of error can be corrected by comparing the updated state to the state produced by the odometry reading. The odometry reading should have a constant error rate, and if the updated state is outside of the odometry error, then the state has been calculated erroneously.

Map slippage errors occur when the algorithm gradually loses track of position and the map becomes distorted due to the fact that "when the algorithm gradually loses track of position and the map becomes distorted" [30]. This causes landmarks to be recorded that in positions that differ over time from their actual positions. This type of problem can be mitigated if the robot returns to a previously known position and then by invoking a loop closure detection algorithm propagates corrections through the bot path [19].

2.8.1 Failure Detection

Although detection of failures is simple to implement, it is a very important part of SLAM. In [30] a simple algorithm is described, which can robustly detect failure. In short
failure detection is achieved when a landmark is expected, but none is detected. If this happens, a warning flag is raised and a counter is incremented. If the counter reaches a certain limit then the robot considers itself lost. However, if the landmark expected is encountered before the threshold is reached, the counter is reset.

In the case of a failure-detection, the robot will try re-localize itself within the last known consistent map [30].

2.9 Loop Closure

Loop closure is a problem within SLAM that deals with the case of a robot returning to a known position [19, 17]. This is when a robot discovers that its position has already been mapped and it is actually a slightly different part of the world, it must then calculate how to adjust the map so that its current point aligns with the originally mapped point [27].

This information is used within a SLAM algorithm not only to update a robot’s position, but also to correct errors that have built up along the robot’s path through sensor and odometry error. This is done by updating the position of all the landmarks observed before the robot returns to a known position, which in turn improves the algorithm’s confidence in the location of each landmark, a process known as correction [19].

Research into loop closure in vision based systems is expanding rapidly due to the increase in the use of cameras for SLAM applications [27]. There are several approaches to this form of loop closure, all of which are discussed in detail and tested in [27].

There are three main approaches to loop closure in the field of vision based SLAM: Map-to-map in which corresponding features in terms of appearance and relative location are tracked, Image-to-image where the most recent image is compared to previous images and Image-to-map in which the most recent image is compared with features on the map according to [27].

The authors of [27] found that map-to-map loop closure did not work well within vision based SLAM as the maps contained too little information to detect loop closure reliably. The authors of [27] did, however, find that image-to-image and image-to-map approaches worked well, with the former being able to benefit from further research and the latter performing the best.
2.10 Cooperative SLAM

Cooperative SLAM is another field of research that is growing steadily, as indicated by the volume of recent research papers available concerning new and updated approaches to the core functions of the SLAM problem in relation to multi-robot SLAM.

Localization for the case of a team of robots, though, is a relatively new field. [25].

One of the advantages of SLAM is being able to map larger areas in parallel, making exploration more efficient if done correctly (such as trying to limit the time that robots spend exploring areas that other robots have already explored) [22].

Multi-robot SLAM does broaden the problem of SLAM itself as considerations such as each robot’s role, centralization, aggregation and communication methods all need to be addressed. Other issues which need to be addressed relate to the starting positions of the robots. An alternative would be the situation in which the starting positions of the robots relative to each other, are not known, and as to how to merge their data. Both of these scenarios have been successfully executed through the use of a Monte Carlo particle filter extended for use in Cooperative SLAM in [10]. In [4] the authors describe how they were able successfully to implement a multi-robot solution, in which the robots used different sensors and no knowledge of each other’s positions, to merge their maps correctly.

There are also several approaches to localization in multi-robot SLAM, such as individual instances of a localizer algorithm or the use of a single distributed localization algorithm on a group of robots. [25]

The authors of [11] have successfully implemented and tested, with good results, a Monte-Carlo based SLAM adapted for multi-robot SLAM. They stated

One of the attractive features of this multi-robot SLAM algorithm is the ease with which it may be implemented.

Further examples of successful multi-robot SLAM implementation are presented in [25] and in [8].

In [11] the author describes how he and his colleagues successfully modified a Monte-Carlo particle filter SLAM algorithm to deal with known and unknown starting positions. To
deal with unknown starting positions, they made the assumption that the robots would, at some point, encounter each other, at which time their respective maps could be updated, and their positions relative to one another in terms of the map coordinates could be tracked [11].

2.11 Robotics Frameworks

Programming robots is far from a simple task, and as a result several frameworks have been developed to increase the ease with which robots may be controlled and algorithms tested. Ref [14] gives a thorough overview of frameworks developed for use in robotics and the advantages and disadvantages to their approaches.

2.12 Summary

For many years SLAM as a whole has been the subject of much research around the world and has many applications within the world of robotics, such as indoor mapping, autonomous vehicles and off-road cross-country navigation. Several major advancements have been made by key people such as Sebastian Thrun [23, 5] and many others. The subject is a large one, with the subjects and algorithms touched here only being a small part of the research as a whole. As such, many years of cross-disciplinary research into the fields of SLAM, robotics and mathematics is required to become a true expert in the field.
Chapter 3

Implementation

3.1 Hardware Setup

Our choice of robot at the beginning of the year was limited. We had several Lego NXT robots in various different setups and a single Fischertechnik Robo Pro robot with either an RF controller or a TX Bluetooth controller.

After experiments from last year, it was decided that the sensors available on the Lego NXT robots would not be accurate enough for our purposes. SLAM requires a certain accuracy range from the sensors and encoders and it was felt that the Fischertechnik would be far more suitable for this type of application.

As for the controller, we decided to stay with the Robo TX Bluetooth controller, as it has a range of up to 10 meters, as opposed to the three meters offered by the Radio Frequency controller. Bluetooth also has the ability to keep several connections alive at the same time, which would be useful in the case that multiple robots were to be used.

Other aspects regarding the controller include that it acts as a simple interface between the computer and the "add-ons" which are connected to it. These add-ons include ultrasonic range finders, motors, light sensors, contrast sensors and counters. It simply allows a program that is running on the computer to interface with the controller over a Bluetooth connection.

The way that the Fischertechnik robots work is that the developer can develop a program that runs on the computer, and the robot is interfaced as a simple entity connection over
the Bluetooth connection. No program is actually downloaded to the robot. Instead, the
application runs and makes calls to a DLL, which then supplies commands to the robot
controller. This in turn, issues instructions to the connected sensors or motors.

At any given time the robot is in a specific state. The DLL "polls" the robot every 12
milliseconds to update its copy of the robot’s state and issue commands. When a program
requests the value from a specific sensor, it issues the command via the DLL, which then
references its copy of the robot’s state information and returns it to the calling program.

The sensors used on the robot were the supplied Fischertechnik ultrasonic sonar sensors.
They have a range of five centimeters to four meters and are capable of reporting range
data at a resolution of 1 centimeter. When the ultrasonic detects an obstacle it rounds
off the reading to the nearest centimeter. The resolution is low so as to try and cover up
the inaccuracies associated with the ultrasonic’s ability to detect range.

The next consideration was how to work with the motors. Firstly, the motors that were
supplied with the Fischertechnik kits were simple stepper motors which, like the ultrasonic
sonar sensors, are controlled by the TX controller. They supplied no information about
axle rotation rate and as a result the controller is unable to supply any information
regarding odometry data.

It was then decided that a calibration program should be built that could be used to
measure the rate of rotation of a motor, thereby allowing wheel distances to be calculated
from time and speed information, making it possible to generate odometry information.

The final consideration at this point was to decide which languages and frameworks were
to be used to control, test and implement the framework. There were many options at
this point, but it was decided to use the framework developed by Leslie Luyt in [14] as it
was designed specifically for use with the Fischertechnik robot very recently, and support
from him was still readily available.

### 3.1.1 Final Configuration

The structure of the robot was built using one of Fischertechnik’s recommended setups
with slight modifications. Space was made at the front of the robot to allow for a second
forward facing ultrasonic sensor to be fitted, 4.6 centimeters apart. The kit consists of
Lego like pieces which fit together to form a frame onto which sensors and motors may
be securely mounted.
The two motors were mounted facing opposite directions at the front of the robot with a 2:1 gear ratio. Both these motors were equipped with tracks. This configuration is known as differential drive [18]. This setup means that the robot turns by rotating the motors at different relative speeds. It also allows the robot to turn on the spot by means of rotating the motors at the same speed, but in opposite directions.

In total the robot consisted of:

- A series of Lego like pieces to build the frame onto which other components were mounted.
- An 8.4V 1500 mAh battery.
- A Fischertechnik Robo TX Bluetooth controller.
- Two Fischertechnik Ultrasonic Sonar range finders mounted facing forwards.
- Two Fischertechnik encoder motors mounted facing opposite directions using a 2:1 gear ratio.
- Two sets of tracks mounted so that the robot would act in a differential drive manner.

3.2 Basic Hardware Testing Software Development

This section describes the development of basic testing software so as to determine the capabilities of the robot, the framework for controlling the robot and then for basic prototyping of initial concepts and ideas.

3.2.1 Initial Problems

The first step in the implementation was to create a small program to test the capabilities of the robot. This would be done using the framework described and built in [14]. The framework is built in Python and provides an excellent base onto which the library of utilities this research is intended to create may be built.

Almost immediately, a problem was encountered. The framework built in [14] was built using Python version 2.4. The current version wrappers for the DLL’s used to controller
the Fischertechnik Robo TX controller are built in Python version 3.1. The syntactical changes in the new version of Python as well as the behavior of certain functions meant that the framework (from now on referred to as the control framework) would not execute.

This was quickly corrected with the help of Leslie Luyt as most of the updates were trivial, such as updating the use of the print statement. A larger and slightly more complex issue was that the new DLLs required a controller ID which was not previously needed and as a result a fair number of methods within the control framework needed to be updated to include the controller ID reference.

### 3.2.2 First Program

With the initial errors corrected, it was time to build a program to test the abilities of the robot. The first program simply had to connect to the robot and command it to drive forwards for a time of about 5 seconds.

This program was written very quickly and worked well. This indicated that the framework for controlling the robot on which the output of this research was to be built seemed stable and sufficiently scalable.

### 3.2.3 Calibration

The next phase in development was to try to create a calibration program that would allow the use of normal stepper motors in the project as we did not have encoder motors at the time.

The idea was to write a program where the robot is set in a known position; then the user would have to enter the amount of time that the robot should rotate around a spot until it reached that mark again. For a certain amount of extra accuracy, the program was intended to make the robot rotate on the spot three times before stopping at the originally marked orientation. This meant that any error built up through a single rotation would be multiplied three times, allowing the user to calibrate to a much finer accuracy. This option was a variable though, allowing the user to create a suitable level of accuracy.

Using the time it took to rotate that distance, would allow the program to calculate the rate of rotation and the wheel size. This in turn would allow the odometry algorithm to estimate wheel positions after certain amounts of time of rotation.
The program was also to include six different calibrations: a specific calibration for smooth surfaces such as tiles and wood, and a separate calibration for rough surfaces such as carpet. It would also calculate calibrations for three different speeds; slow; medium and fast on each of these surfaces. Finally the program was to create an XML file named with the name or ID of the robot which would hold all the calibrations.

```
<bot surface="1">
  <timing speed="fast">
    5.555555555555
  </timing>
  <timing speed="medium">
    11.1111111111
  </timing>
  <timing speed="slow">
    16.666666667
  </timing>
</bot>
```

Figure 3.1: Output of the calibration program.

The program itself was fairly quick to implement. The control framework requires that a class is defined that inherits from the FT_Constants class to indicate what connections have been made on the physical robot. For our purposes, it was sufficient to simply create an instance of the Bluetooth TX controller interface and define two motors; left and right.

Firstly, six variables were created to hold the timings of the various calibrations. Then, for each calibration calculations, at least one iteration through the following code block:

```
do = "y"
while(do == "y"):
    try:
        print()print("----- Fast -----")
        print()
        time_fast = int(input("Enter time for a str(accuracy) + # rotations (milliseconds): "))
        ctrl["Motor_Manager"].turnspin("left", ctrl["Constants"].speed_fast, time_fast)
    except ValueError:
        print("Invalid time value")
        print()
    do = input("Would you like to recalibrate for "+str("Fast")? (y/n) ")
```

Figure 3.2: The code used to determine the variables used for calibration

Each code block was slightly different allowing the program to store the values for each of the calibrations. Each iteration would allow the user to continuously recalibrate for that specific speed and surface type until he/she was happy with the amount of error.
Once the calibration timings were complete, the program would pass the values off to a function that would calculate their calibration values, i.e. the time it takes on that particular surface with that particular speed setting for the wheel to make a single complete rotation. This number would then be stored in the XML file already described.

Before the program was completed, though, it quickly became apparent that this approach to odometry would not work for several reasons. Firstly, and probably most importantly, the battery level began to drop exceeding quickly, causing the motors to rotate much slower. This meant that many more points of reference for motor outputs would be needed as well as a way to accurately monitor battery levels. Secondly, as the robot switched between different surface types the calibration should be changed, and this was not possible in a manner which would allow the robot to move autonomously through its environment.

At this point it was decided that dedicated encoder motors were needed. As already mentioned, encoder motors use, depending on what encoder is purchased or built, various methods of determining true axle rotations. Our encoders are of the magnetic type, which count pulses from the axle to report wheel rotations.

### 3.2.4 Ultrasonic Sonar Sensor Testing

The next step was to test the ultrasonic sensors capabilities, as the information supplied by Fischertechnik does not give accurate details about the hardware that they supply. The sensors that we had were supplied with the original radio frequency controller and were of the two wire type.

The first major issue encountered was that once the sensors were attached to the new Robo TX controller they were not detected by the controller and as a result did not report any range data at all. After some research in the manuals, which are all written in German, it was discovered that the sensors supplied for the RF controller were not compatible with the new TX Bluetooth robot platform controller, and that the new "3 wire" version was required.

The new ultrasonic sensors and encoder motors were ordered at the same time directly from Germany as there were some issues with the local suppliers and arrived within two weeks. It was then discovered that the motors mounted differently to the frame compared to the original plain stepper motors. Where the original motors mounts from the front
face (the face out of which the axle points) the new ones mounted along the side of the motor. This was not a major issue, however, as we had ordered a complete new platform which was designed with the new motors and sensors in mind. This meant that the new robot only took about a day to rebuild.

The first testing carried out on the ultrasonic sensors was to check if the range was as mentioned in the reference manual. This was done by using the software supplied by Fischertechnik that is primarily used for visual programming as a teaching aid. It also allows the user to view what the outputs of all the sensors are at any given time and to be able to control the motors individually.

The robot was then put in a known position and a simple pencil was used to check the range outputs. The pencil was mounted so that it would stand vertically, allowing for high visibility to the sensors. The sensor was then moved along the X-axis in reference to the robot. Within all of the software developed for this thesis, the X-axis is used as the reference frame for angular change, that is, if the robot is facing along the X-axis, it is facing 0 degrees.

By moving the pencil along the X-axis in front of the robot it seemed to detect, fairly reliably, correct range readings with a range of five centimeters to four meters. Like all sonar sensors, though, it did suffer from fairly severe errors. Firstly, it did not always detect the pencil, even when well within the range of the sensors. Secondly, it also often seemed to report ranges that were not possible, e.g. a range of ten meters when the pencil was within one meter of the robot, with a wall less than a meter beyond that. This confirmed the susceptibility of sonar sensors to errors related to multiple reflections of the sound wave generated to detect range using a ToF algorithm.

The second set of tests regarding the properties of the ultrasonic sensors was to determine the arc width of their scan. This is the width of the arc described by the pulse of sound that the sensor generates to detect objects. The robot was again placed in a known position (that is, it was placed at a point that we defined to be the origin facing 0 degrees, along the X-axis) on a markable surface. Then the center-points of the sensors were marked.

The pencil was used again in this test and was moved along the Y-axis to determine the width of the scan arc. Once the sensor moved out of the detectable area of the sensor, the sensor would then report the range of the wall beyond it, and a mark would be placed to indicate the edge of the sensor arc. This test was repeated four times, once for each edge of both sensors.
3.3. BASIC FRAMEWORK OVERVIEW

To calculate the width of the scan arc for each sensor, the center points of the sensors were then considered to be the origin (for their individual calculations). Then, using an arc-tan with the relative positions of the points at which the sensors lost "sight" of the pencil, it was possible to calculate the relative angle between the sensor and the point, giving the angle of the arc.

For the left hand edge of the arc (positive relative rotation) the calculation, using the point P1, was:

\[
\text{arctan}(Y_{p1}, X_{p1}) \times 2
\]

It is possible to confirm this test by repeating the operation on P2. The expected result of this test was that of most sensors which, according to the literature, is somewhere in the range of 22.5 to 30 degrees. The actual result for the Fischertechnik sensors was almost exactly 60 degrees of arc width.

When compared to a laser sensor, this width is massive, but when considered for application within this particular research, it has both particular positive and negative results. Firstly, when considering that the robot will only have two ultrasonic sensors, both facing forwards, the robot will be able to view objects that are not directly in front of it, with a considerable scan width. The downside though, is that the scans will have a higher likelihood of creating unstable intersections (intersections which have large areas of arc that are common to both due to their radius and close origins.)

3.3 Basic Framework Overview

The goal output of this research project is to create a framework with the basic tools for implementing a SLAM algorithm on different platforms, allowing for the developers to use the tools provided, or to implement interfaces. This will allow SLAM algorithms to keep a basic overall structure, and an easier, more standard approach to implementing these algorithms.

There are two basic types of SLAM algorithm: online SLAM in which the algorithm will try to interpret the sensor and odometry data on the fly, and offline SLAM in which all the data from the sensors and encoders are recorded and analyzed later for interpreting and playback. As a result, it was decided that two different versions of the framework would be implemented.
3.4 Online SLAM

The first version of the framework was designed for use in online SLAM applications and as such is written in Python, for use with our Fischertechnik robot and the control framework built in [14]. The framework consists of several classes, separated into three different libraries: CommonUtils, OdometryUtils and TBFUtils.

The distinction of being built for online SLAM is an important one. Algorithms such as odometry and pose tracking need to happen on the fly as the robot moves around its environment. For this reason, the Odometer class and the TBFCalculator class are implemented as Threads, which is the main distinction between the offline and online variants of this framework.

3.4.1 CommonUtils

*CommonUtils* is a set of classes that are frequently used by the other classes in the framework. They provide common functionality such as representing the pose of the robot information gathered from scanners. Essentially, these classes are how the different parts of the framework communicate in a unified fashion, using standard interfaces and classes. *OdometryUtils* and *TBFUtils* contain the classes supplied in the *CommonUtils* library and provide further functionality that is specifically used within SLAM implementations. The list of classes available follows, as well as an explanation of each class.

3.4.2 CommonUtils Organization for Online SLAM

The overall structure of the *CommonUtils* module is shown in Figure 3.3.

3.4.3 Point

This is a class for representing a point in Cartesian coordinates, used for representing positions of various objects, such as landmarks. It has a *toString()* method for debugging purposes as well as an *equals()* method for testing equality of Cartesian coordinates between separate instances.
3.4. ONLINE SLAM

3.4.4 Pose

The class extends the Point class to provide heading information. This allows it to be used to represent the robot’s position in Cartesian coordinates as well as the direction in which the robot is facing.

Pose has several more functions for allowing the developer to set the position and heading after the object is instantiated. This is useful for when a Localization algorithm wishes to apply a state update, or when the odometry class calculates the current position of the robot. The class also provides the `toString()` and `equals()` functionality for debugging and value equality checking.

3.4.5 Configuration

This class is a useful tool in the setting up of instances which require information about the robot’s physical properties. It has several variables relevant to the physical setup of the robot, including odometry parameters such as wheel spacing and size and sensor information such as sensor positions relative to the center of the robot.

Figure 3.3: Organization of the CommonUtils package
3.4.6 ConfigurationFetcher

This class allows the developer to specify the robot’s name, which it then uses as a file name to find an ini file which will have the dimensions and other properties of the robot. Once it has opened this file, it parses it using the Python configparser module. The class will then return an instance of Configuration which will contain all of these values.

Both odometry and TBF related classes use this information for their calculations. There are several advantages to using a class such as this. Firstly, it creates a uniform method of passing related variables to different classes within the framework. Secondly, it effectively shortens constructor signatures by needing to only pass a single instance. Finally, it also means that a single file lookup is able to provide all the information that a program is likely to use and makes editing of those settings a simple process of simply editing a configuration file, rather than re-compiling source code.

3.5 Offline SLAM (Java)

The framework built for use in offline SLAM has a much stronger structure than that of the Python equivalent due to the strong typing properties of Java. It also has two additional classes provided for the specific requirements of offline SLAM: the ScanStep class and the Landmark class.

3.5.1 CommonUtils Organization for Offline SLAM

The structure of this version of the framework is shown in Figure 3.4. Note that the Configuration, Point and ScanStep classes only show their member-variables as space does not permit me to give the full UML class diagrams.

3.5.2 Point, Pose and Configuration

These three classes are much the same as they are within the online version of the framework. They differ only in the fact that the variables are strongly typed (due to it being written in Java) and that the classes have accessors for all variables. This allows for the implementation of thread safety constructs. This is important since in the case of
cooperative SLAM, where multiple robots could be implemented as threads and may be accessing the same data.

### 3.5.3 ScanStep

The `ScanStep` class, like the `Configuration` class, is one meant mainly as a container for related information. It holds information regarding relative encoder counts (positive or negative differences from the last time step in bot movement) and sensor readings. It is meant to be held in an `ArrayList<ScanStep>` so that the programmer may move through the steps iteratively, much like the polling of the robot’s sensors and encoders. Like most of the other classes within this version of the framework, it has been implemented with thread safety in mind. Firstly, all the member variables are private and have associated accessors, primarily using the `synchronized` keyword to try and preserve thread safety.

### 3.5.4 Landmark

The `Landmark` class is another class that is implemented in the offline SLAM framework, and is meant to be used, like the `ScanStep` class, within an `ArrayList`. This is one of
the more important classes as it is used to hold information regarding the position of the landmark, as well as the amount of times that that landmark has been observed. This is important since, as is be explained in section 3.7, algorithms such as the TBF cannot keep a record of every single scan from the sensors as this quickly becomes computationally expensive. The array list that holds the landmarks is used for association between new observations and already existing landmarks.

The array list acts as the current map of the observed world from the algorithm’s point of view. The distinction between the observed world and the actual world map is an important one as, at best, any algorithm can only estimate positions of landmarks based on data that has a degree of error. Another reason is that the map stored within this array list is only of landmarks that have been reliably detected as the robot has moved along its path and is by no means the complete world map.
3.6 Odometry Implementation

Odometry, as already mentioned, is one of the most important aspects of the SLAM algorithm, as it serves as an initial guess as to the robot’s current position and heading. It is also highly error prone, which is one of the driving reasons why localization algorithms are needed in the development of mobile robotic applications.

Odometry was the first part of the framework, once the initial over-all design decisions were made, to be implemented. It works by using encoder axle rotations to estimate wheel rotations and subsequently robot movement.

This cannot be done perfectly for several reasons. First are the reasons already discussed such as wheel slippage. Another reason that this cannot be done precisely is that, while counts are a very good indicator of wheel rotations, they do not indicate the direction in which wheels were rotating.

Some encoders may report this information by means of a positive or negative count, but if the wheel changes direction in-between processing of counts, those counts are effectively lost, introducing error. For this reason, some encoders only report a positive number indicating axle rotation in any direction. This means that counts are not lost in the way described above. In this approach, the framework or program instructing the motors to turn is responsible for recording which direction the motors are instructed to rotate. This approach can be error prone though. If the controlling program attempts an odometry update by first checking the direction in which the wheels are rotating, and then querying the encoders, encoder “ticks” in another direction that occur in the latency associated with querying the robot and the receiving of the count will be lost.

There are several different approaches to odometry within the field of robotics. The first approach that was implemented for this research assumes that all movements occur in straight lines, while the second approach is to assume that all movements rotate about a point determined by the rotation of the wheels. Both of these approaches only work acceptably well if odometry updates are done in a very short interval to mitigate errors caused by a change in direction between odometry updates.

Since this research has been undertaken with the use of a differential drive robot, all of the calculations mentioned here will be done with a differential drive setup in mind. With the potential for a different drive setup, different methods of implementation of odometry being needed and/or different types of encoders being used, the OdometryUtils module of the library has two different interfaces for this purpose: Encoder and Odometer.
3.6. ODOMETRY IMPLEMENTATION

For the rest of this chapter, I will be referring to the specific implementation as in the Java framework, as this is a stricter version of the framework and the basic structures are very similar in operation, unless specifically mentioned that the code being referred to is for the Python-based framework.

3.6.1 Straight Line Odometry

As already mentioned, straight line odometry operates under the assumption that a robot only ever moves in a straight line. In other words it cannot rotate, but instead a rotation happens in a triangular fashion.

The only inputs to an odometry algorithm are that of the encoder counts and the direction of wheel rotation, which is supplied by the calling process. Our encoders work by maintaining a count of "ticks" since the last clearing of the counter. If the counter were not reset, the count quickly grows into thousands of ticks over a very short period of time. This meant that before the ticks could be used as a reference of wheel rotation, they had to be reduced to the relative amount of ticks since the last odometry update. The only other inputs supplied to the odometer are the physical properties relating to the setup of the robot.

The first task of the algorithm is to initialize the ratios and other variables needed in the calculations. This is given in figure 3.6.

```java
private void init()
{
    this.ot.configuration.wheel_distance =
    this.axis_wheel_ratio*this.scaling_factor*
    (this.wheel_diameter_left+this.wheel_diameter_right)/2;
    this.ot.configuration.wheel_conversion_left =
    this.wheel_diameter_left*this.scaling_factor*
    Math.PI/this.increments_per_cour;
    this.ot.configuration.wheel_conversion_right =
    this.wheel_diameter_right*this.scaling_factor*
    Math.PI/this.increments_per_cour;
}
```

Figure 3.6: The code to initialize the odometry algorithm

The first operation in the initialization code stores the ratio of angular change that would be incurred during wheel rotation when the wheels are rotating at different speeds. The second and third operations hold the variables that determine the distance traveled by a
wheel through one step of an encoder. Also, within these calculations, a variable called scaling_factor is used. This is a variable for calibration and fine tuning of the algorithm, since measurements of the dimensions of the robot may be inaccurate.

Since the odometry module is created in class form, a pair of member variables were created and initialized to zero. Then, every time an odometry update occurs, the newest value of the encoder count replaces the last value.

The calculations performed using these counts required that the differential number be negative if the wheel is rotating backwards in relation to the robot. So the first task of this, and any other odometry implementation using this kind of encoder, is to correct the count of ticks to be relative to the last time an odometry update occurred. This is done by subtracting the last encoder value from the current encoder value and then setting it to negative if the direction of the axle rotation is negative. The following formula will yield the correct delta value:

$$(\text{Count}_{\text{new}} - \text{Count}_{\text{old}}) \times \text{Direction}$$

Where the variable Direction holds the integer indicating the direction of rotation of the wheel, assuming that $-1$ indicates a negative rotation and $+1$ indicates a positive rotation. Lines 54 and 55 in Figure 3.7 perform the above mentioned calculation.

Figure 3.7: Odometry algorithm

```java
54    double delta_pos_left = so.getLeftCount();
55    double delta_pos_right = so.getRightCount();
56    if (delta_pos_left != 0 || delta_pos_right != 0)
57      {
58        double delta_left = delta_pos_left * this.ot.configuration.wheel_conversion_left;
59        double delta_right = delta_pos_right * this.ot.configuration.wheel_conversion_right;
60        double delta_heading = (delta_right - delta_left)/this.ot.configuration.wheel_distance;
61        double theta = (delta_heading) * 0.5 + this.ot.result.getTheta();
62        double delta_x = (delta_left + delta_right) * 0.5 * Math.cos(theta);
63        double delta_y = (delta_left + delta_right) * 0.5 * Math.sin(theta);
64        this.ot.result.increaseX(delta_x);
65        this.ot.result.increaseY(delta_y);
66        this.ot.result.increaseTheta(delta_heading);
67        if (this.ot.result.getTheta() > Math.PI)
68          this.ot.result.decreaseTheta(2 * Math.PI);
69        if (this.ot.result.getTheta() < Math.PI)
70          this.ot.result.increaseTheta(2 * Math.PI);
```

Line 57 makes sure that at least one of the wheels has made a rotation since the last
3.6. ODOMETRY IMPLEMENTATION

odometry update. This is to help keep computation down by not performing calculations when the robot has not moved.

Lines 59 and 60 calculate the distance traveled by the left and right wheels using the differential count of the encoders since the last update and multiplying it by the size of each step.

Line 61 calculates the change in heading relative to the starting heading. This is done by determining the difference in each wheels rotation and dividing this by the wheel distance variable explained above.

Lines 63 through 66 declare a variable called theta and initialize it to the half-way point between the original heading of the robot and the new heading. This is done so that the change in the $X$ and $Y$ positions of the robot may be calculated from an estimated point between the two headings and as such is a crucial step for straight line odometry. This formula indicates the method of determining the change in $X$ and $Y$ positions:

$$\Delta X = (\Delta L + \Delta R) \times 0.5 \times \cos(\theta)$$
$$\Delta Y = (\Delta L + \Delta R) \times 0.5 \times \sin(\theta)$$

Where $L$ and $R$ indicate the differential encoder counts and $\theta$ indicates the half-way point between the old and new heading of the robot. Being an estimated point, this does introduce a slight error, which is greater, the larger the change in position is. This is why this algorithm needs to be invoked as often as possible to keep the introduced error as low as possible.

Lines 68 through 70 set the new position of the robot to the odometer’s current view of where the robot is. This is important so that the next time the odometer is invoked, it has a record of the last position of the robot and it can continue to use the differential encoder counts.

Lines 72 through 75 are simply tasked with keeping the robot’s heading within the range $-\pi \leq \theta \leq \pi$, because certain mathematical functions require the heading to be within that range.

Other functionality included in the odometry implementation includes getters and setters for the current pose of the robot, allowing the calling algorithms to retrieve the current position without recalculating, as well as allowing the position to be safely corrected by a
localization algorithm. For a full listing of the $FT\_Odometer$ class and its functionality, see the Appendix.

The usage of the $FT\_Odometer$ has been kept clean and simple. It simply needs to be instantiated with a reference to a $Configuration$ instance to provide it with the physical properties of the robot.

### 3.7 Triangulation Based Fusion

Triangulation Based Fusion (TBF) is an algorithm that deals with both landmark extraction and data association when using ultrasonic range finders for landmark detection [26]. The algorithm consists of two main parts; firstly, a triangulation algorithm which triangulates separate sensor scans to try and find an intersection between them and, as a result, extracts a landmark (landmark extraction). The second part is an algorithm that maintains a sliding window of sensor scans to try and use more than two individual scans for triangulation, and then associates the newly extracted landmark with an already existing one.

One of the only requirements for this module ($TBFUtils$) like all the other modules, was ease of use. As a result, the $TBFCalculator$ class was designed in such a way that it only needs to be instantiated with an instance of $Configuration$ (again, just for the physical properties of the robot, such as sensor positioning relative to the center of the robot). For the class to be used, it simply needs an instance of $ScanStep$ to be passed for each new set of information parsed by the calling program. The class maintains its own records of information passed to it in terms of a list of $Landmarks$ extracted is be able to refine these landmarks with every new set of information passed to it.

#### 3.7.1 Triangulation

The first step in implementing the TBF algorithm was to implement a method for efficiently, and as accurately as possible, triangulating a landmark from two separate sensor-scans. This is needed, as already explained, as ultrasonic range finders can only report the range from the nearest object within their field of view as well as having a very wide field of view (60 degrees with our equipment).
3.7. TRIANGULATION BASED FUSION

The private function *getPt()* is used to find valid intersections between two separate ultrasonic scans. It requires that two instances of the class *UltraSonic* be passed to it, as they hold information regarding the position of the sensor, as well as the sensors range reading and its arc width.

The first operation performed by the triangulation algorithm is to find the distance between the two sensors in terms of $\Delta X, \Delta Y$ and the scalar distance. It will also check that the distances between the two scans can produce a valid intersection. This is done by the code given in Figure 3.8

```java
double deltaX = right.position.getX() - left.position.getX();
double deltaY = right.position.getY() - left.position.getY();
double distance_squared = Math.pow(deltaX, 2) + Math.pow(deltaY, 2);
double distance = Math.sqrt(distance_squared);

if (distance > (left.range + right.range) || distance < Math.abs(left.range - right.range))
    return null;
```

Figure 3.8: Code for determining the distance between the two sensors

Figure 3.9 gives the code to determine the angle between the center position of the two sensors and the two points of intersection generated by two circles that form the arcs described by the sensor ranges. It also generates the two points at which these circles intersect.

```java
double k = (Math.pow(distance, 2) + Math.pow(left.range, 2) - Math.pow(right.range, 2))/(2*distance);
double px1 = left.position.getX() + ((deltaX*k)/distance) + (deltaY/distance)*Math.sqrt(Math.pow(left.range, 2) - Math.pow(k, 2));
double py1 = left.position.getY() + ((deltaY*k)/distance) - (deltaX/distance)*Math.sqrt(Math.pow(left.range, 2) - Math.pow(k, 2));
double px2 = left.position.getX() + ((deltaX*k)/distance) - (deltaY/distance)*Math.sqrt(Math.pow(left.range, 2) - Math.pow(k, 2));
double py2 = left.position.getY() + ((deltaY*k)/distance) + (deltaX/distance)*Math.sqrt(Math.pow(left.range, 2) - Math.pow(k, 2));

Point p1 = new Point(px1, py1);
Point p2 = new Point(px2, py2);
```

Figure 3.9: Code for determining angle between the robot and the detected object
The final step in triangulation is to determine which of the two possible points generated by the above code are valid. Valid, in this case means that the point is "visible" to both the sensors given their position. To this end, a function was created called checkWithinArcs() which requires that an instance of UltraSonic be passed as well as the point which is going to be evaluated.

The function works by determining the minimum and maximum viewable angles of the sensor and then determining the angle of the vector between the sensor’s position and the point being tested. The code given in Figure 3.10 performs these operations.

```java
private boolean checkWithinArcs(UltraSonic sensor, Point Pt) {
    double minRange = sensor.position.getTheta() - (sensor.arc_width/2);
    double maxRange = sensor.position.getTheta() + (sensor.arc_width/2);

    double dy = Pt.getY() - sensor.position.getY();
    double dx = Pt.getX() - sensor.position.getX();

    double result = Math.atan2(dy, dx);
    return result >= minRange && result <= maxRange;
}
```

Figure 3.10: Code for checking if a point is actually visible to a specific sensor

With these functions implemented, it is a small matter of passing all triangulated points through the function checkWithinArcs() to determine which points are valid and which are not. It is important to note that these points are newly detected points, but may actually belong to an already existing landmark, with which they should be associated.

### 3.7.2 Associating Points with Landmarks

The second part of the algorithm tries to refine points by associating scans from separate time steps to refine a detected point. It then attempts to associate the refined point with an already existing landmark, and then further refines that landmark by using the number of associated triangulations and the number of triangulations used to define the current point. This is done because a landmark is observed over several time steps, and due to the inaccuracy of the sensors, needs to be updated as more corroborating data is obtained about the true position of the landmark. Further, a landmark is not considered a landmark until it has reached a minimum threshold of triangulations in an attempt to
maintain an acceptable minimum level of accuracy. This however, is not handled by this algorithm, and is up to the calling process to decide.

With this in mind, the point of this part of the algorithm is to work in tandem with the triangulation algorithm to raise the levels of accuracy associated with triangulation using fairly inaccurate range scans. To this end, one of the major concepts implemented within this algorithm is that of the sliding window. This is discussed in detail in [26] but is essentially a \( n \times m \) sized array, where \( n \) is the window size (10 in this implementation) and \( m \) is the number of sensors used by the robot.

Every time a new set of range scan data is supplied to the algorithm it is added to the rightmost column of the array, pushing the leftmost column out of the array so that at any given time there is a maximum record of the last \( n \) scans performed by each sensor, describing an arc from that sensor’s position. This is done so that every time there is a new scan performed, there are at most \( ((n-1)*m)+1 \) arcs that can potentially be triangulated to confirm and refine the position of a detected landmark. The size of the sliding window is set rather than simply growing with every scan to keep the computational complexity of the algorithm low. If the window’s size were unbounded, the calculations for triangulation on every scan would grow rapidly, especially when considering that scans occur at least several times a second.

Firstly, the algorithm checks that the scan data is valid by only accepting scans if they are all non zero and are smaller than the maximum reliable distance of the sensor (four meters in this implementation). The algorithm then determines the position of the sensors using the Configuration instance supplied at the time of instantiation and the current position of the robot supplied by the odometry. The code in Figure 3.11 performs the calculation required to determine the position of the sensor:

Once these positions have been calculated, the algorithm creates new instances of the UltraSonic class and adds them to the sliding windows so that they may be used by the triangulation algorithm.

The next operation performed by the algorithm is to create valid triangulations. This is done by iteratively moving down the last column of the sliding window (each of the newest scans) and attempting to triangulate each scan with the other scans in the window.

The code in Figure 3.12 generates an array of all of the possible valid points determined by the intersections of the scans in the sliding window with the newest scan data. These points are then refined into a single landmark with an associated number of triangulations.
The final step is to either add this landmark as a new landmark in the map, or to associate it with an already existing landmark. This is done by comparing its distance from every other landmark already in the map. If an already existing landmark is within the acceptable range (30 centimeters in this implementation), it is considered to be the same landmark. The landmarks are then combined into one with its location set as the average location determined by the ratio of the triangulations for each individual landmark.

3.8 Python Variations

The Python implementation is very similar to the offline version of the framework. It does, however, lack a few classes that are not needed for the online implementation. Firstly, the \texttt{ScanStep} class is not included since the online version implements these algorithms as threads and as a result, they get passed references to the hardware that they will be using, such as encoders for the odometry and references to the sensors for the TBF algorithm.

The reason for this is that online algorithms work by localizing and mapping as the robot moves through the environment. This means that at any given time, the odometry data as well as the visible landmark data should be as up-to-date as possible and this is only achievable through threading. Other than the extension to a threaded module and the inclusion of references to hardware interfacing classes, the actual algorithms that make up the \texttt{Odometer} and the \texttt{TBFCalculator} are identical save for syntactical and function-name
for(int i = 0; i < 2; i++)
{
    UltraSonic rin = this.scans.get(this.scans.size()-1)[i];
    ArrayList<Point> p = new ArrayList<Point>();
    for(int j = 0; j < this.scans.size()-1; j++)
    {
        for (int k = 0; k < 2; k++)
        {
            UltraSonic rkj = this.scans.get(j)[k];
            Point pt = this.getPt(rkj, rin);
            if(pt != null)
                p.add(pt);
        }
    }
    this.addRefinedPoint(p);
}

Figure 3.12: This method associates as many scans as possible from the sliding window with the current scan
differences between Java and Python.
Chapter 4

Results

4.1 Introduction

In this chapter, the methods used for testing the framework in its capacity to aid the development of both offline and online SLAM implementations are described. This includes the implementation of a few classes which are implementation specific, such as the controlling program, graphical output classes and the accuracy of the algorithms implemented in the Framework.

4.2 Online SLAM

Since the framework specification did not include a method for autonomous navigation, the first step was to create a means by which to control the robot remotely. It was decided that the use of a remote control would be best since it would allow for easy course adjustment and the ability to finely control the robot.

A program was developed in C# to take advantage of the Microsoft Xbox controller, which was perfect for this kind of test as the analog joy sticks could be used to drive the individual tracks of the differential drive robot. The program was designed so that pushing forward on the joysticks would rotate the tracks forward, and pulling back on the joysticks would rotate the tracks in a negative direction.

The program uses the XNA framework supplied for interfacing with the Xbox controller. It converts the values indicating the state of the controllers joystick on the Y-axis (in the
range -1 to 1) and converts it into the range of -512 to 512 which indicates positive or negative rotation and speed in the controller framework. This information, along with controller ID is sent over UDP to the program directly controlling the robot.

The program was built with another class extending the thread class which has references to the controller framework and a UDP socket. This allows the class to interpret control signals received from the UDP socket and to command the encoders to move in any specific direction and speed.

Once this had all been done, the first test to be executed was to test the odometer class. The robot was placed in a known position and then driven in a meter by meter square and the odometry coordinates were checked.

While the distance travelled by the robot was fairly accurate, the direction in which the robot was facing was very wrong. It reported near random results for all tests, with the distance usually being between 10 and 20 percent of the actual distance travelled while the indicated direction was almost random at any speed and any shape of movement path.

The first modification to the framework came after realizing that clearing the encoder count on the controller after every odometry update caused unused encoder ticks to be lost in the latency between sending the command to clear the encoders was sent and the actual clearing being executed as the robot was still moving at this time. This is due to the fact that the robot could only be polled every 12 milliseconds, even though this seems like a very quick execution time. The modification was to keep track of the last encoder count and do a simple subtraction after every update in the odometer’s records. This made a significant improvement to the performance of the algorithm.

The final modification to the odometer algorithm came after realizing that the Xbox controller was causing errors in odometry calculation. Every time the joystick was released it centered itself on its axis, but only after bouncing slightly over in the opposite direction to the last direction of movement. While this slight bounce did not cause the robot to move erratically, the framework recorded a change in direction, which was then used in the odometry calculations. This caused the last update to be in the wrong direction after every release of the joystick. At this point the decision was made to move to hard coded movement paths.

Finally, the robot was moved to a carpeted floor, as it had been, up to this point, tested on a wooden floor which was causing a near imperceptible degree of slip even at low speeds. Once this change was made, the robot’s odometry improved greatly, to well within the
4.3 Offline SLAM

Offline SLAM was tested again, using a simple series of programs. The first was a program written to simply track the encoder ticks and sensor readings for each time step and then, once the program was finished executing, to save this data to a file which could be analyzed by an offline SLAM program. It is important to note that the program running on the robot was not doing any analysis at all, simply recording values. For the complete program to complete this task, refer to the Appendix.

The program for performing the offline SLAM analysis was written to do odometry updates and the TBF algorithm testing. The program itself simply had to load the file containing the data from a specific sensor run, and analyze it on a line by line basis, creating an instance of ScanStep to pass to the various parts of the framework, depending on the required functionality.

First, the odometry algorithm was tested. Again, the robot was moved in an almost square path relating to its starting position, recording its values for analysis. The program for performing this test simply has to initialize an Odometer class, the TBF class, and some classes for displaying the output. Then it creates an ArrayList<ScanStep> to hold the information about the entire run performed by the robot. Finally, it simply has to iterate
through the entire array of recordings, passing the individual \textit{ScanStep} instances to the classes. The code for iteratively moving through this array for performing odometry updates is shown in Figure 4.1.

![Figure 4.1: Simple Offline program analyzing encoder data to perform odometry updates](image)

As seen in this code, the process of developing an algorithm for tracking pose by odometry is simple. In the code above, \textit{sd} is an instance of a class, which shows the output path of the robot according to the information obtained from the odometer class. Figure 4.2 shows the output of the program.

![Figure 4.2: Output of the odometry program](image)

This output is very close to the actual movement path of the robot. The robot which started at the position (0,0,0) actually ended this run at the position (-19, 15.5, -100), which is well within the limits imposed upon odometry error in [19].
The second step was to test the Triangulation based fusion algorithm’s output. But to test the triangulation and the full algorithm separately, the same run was used, with different parts of the algorithm turned off. The diagram in Figure 4.3 shows the run with simple triangulation occurring but with no attempt to associate any of the landmarks and points and no attempt to refine their position.

![Figure 4.3: Offline odometry and triangulation program that does not use data association](image)

The red dots indicate the positions of triangulations and clearly show the need for a data association algorithm to clean up the positions by using multiple triangulations and already existing landmarks to refine landmark positions. The light grey line indicates the last point detected by triangulation. Notice that the bots path overlaps some of the red
The robot did not actually move over or through the landmark, but simply moved close to it, and as such moved through some of the erroneous detections.

The final figure, Figure 4.4, shows the output of the completed TBF triangulation and data association algorithm. Again, it shows the robot’s path, current orientation, landmarks and a line indicating the last landmark detected. Also, the landmarks size is much larger than that of the previous image to indicate that it has had at least 20 triangulations to confirm its position. The size is not an indication of the size of the landmark and as such, the robot’s overlapping path through one of the landmarks is simply an indication that it moved very close to that landmark. Finally, the blue text indicates the number of triangulations associated with each landmark as well as the landmarks position.

4.4 Results

The results of testing the online SLAM framework regarding odometry were positive. They were just as accurate as the offline framework’s odometry implementation which indicates that the computational complexity of calculating odometry is low enough to be executed in real time.

However, the limitations of the hardware platform and the method in which programs run over a Bluetooth connection on the Fischertechnik platform does not lend itself to an online SLAM algorithm as a result of latency and bandwidth limitations. This led to a breakdown in accuracy once the TBF algorithm was used in parallel with odometry updates.

The offline SLAM framework performed significantly better. The odometry was near perfect, but only once precise conditions were met with regard to speed and surface. Even with low speed, tank tracks and a rough carpet to move on, the robot’s odometry still drifts, which clearly indicates the need for a localization algorithm to be implemented to help correct odometry error.
Figure 4.4: Offline odometry and triangulation program using data association
Chapter 5

Conclusion

5.1 Project Summary

The main goal of this research project was to create two separate frameworks to aid in the implementation of SLAM algorithms by providing common functionality, with specific aims of supplying that functionality for Fischertechnik robots such as the ones we have. This was achieved by first implementing several test programs to discover the characteristics of the available robot in terms of hardware capabilities. The second step was to create the framework aimed for use within the online SLAM paradigm, and then to create a second framework in Java for use within offline SLAM.

5.2 Outcomes of Objectives

The goal of the frameworks created here was to create a set of tools for use in both online and offline SLAM. While these goals were met for the offline SLAM variation, the online version did not perform as well as was expected. This was primarily due to the limited capabilities of the Bluetooth connection required by the Fischertechnik controller. Both implementations lack a form of localization algorithm for time constraint reasons and are needed for use in fully implemented SLAM algorithms.

Another, yet equally important, goal of this project was to make use of the functionality provided by this framework as simple and as quick to implement as possible. This was a success shown in the fact that both major algorithmic classes within the framework only
require instantiation with the physical attributes of the robot, and then a simple call to method with data regarding position and sensor information.

5.3 Extensions

The following list describes possible extensions to the framework in both online and offline forms.

**Additional encoder implementations**

The encoder implementation that was successfully tested was that which deals with straight line movement on a differential drive robot. Other implementations may be implemented for use with other robot configurations or odometry movement models.

**Additional landmark extraction models**

Other models could be added to perform landmark extraction such as RANSAC or spike landmarks. These models should also provide functionality for dealing with different sensor types such as laser or contrast sensors.

**Addition of localization algorithms**

The addition of localization algorithms is a pivotal part of this framework, and needs to be added if the framework is to gain any substantial usefulness. Examples would include an implementation of an Extended Kalman Filter or a Monte-Carlo particle filter.

**Addition of a loop-closure algorithm**

Loop closure is another aspect directly involved in increasing the accuracy of a SLAM implementation and should be implemented with the use of a localization algorithm.

**Extensions regarding Cooperative SLAM**

Extensions should be made for use in Cooperative SLAM, such as robot aware odometers and localization algorithms. This would also include the addition of classes to perform tasks such as map merging.


Appendix A

Algorithms

A.1 FT_Odometer

package OdometryUtils;

import CommonUtils.Pose;
import CommonUtils.ScanStep;
import CommonUtils.Configuration;

/**
 * An odometer implementation that uses triangular estimation to
determine the robots current position. The method step should
be used as often as possible to increase the accuracy of this
algorithm.
@author Shaun
*/
public class FT_Odometer implements Odometer
{

    private OdometryTrack ot = new OdometryTrack();

    private int increments_per_tour = 460;
    private double axis_wheel_ratio = 1.695;
    private double wheel_diameter_left = 10.059;
    private double wheel_diameter_right = 10.059;
private double scaling_factor = 1.021;

/**
 * An odometer implementation that uses triangular estimation to determine the robot's current position.
 * The method step should be used as often as possible to increase the accuracy of this algorithm.
 * This default constructor will initialize the robot to the settings for the Fischertechnik track bot.
 */
public FT_Odometer()
{
    this.init();
}

/**
 * An odometer implementation that uses triangular estimation to determine the robot's current position. The method step should be used as often as possible to increase the accuracy of this algorithm.
 * @param config The physical configuration of the robot
 */
public FT_Odometer(Configuration config)
{
    thisincrements_per_tour = config.getIncrements_per_tour();
    this.axis_wheel_ratio = config.getAxis_wheel_ratio();
    this.wheel_diameter_left = config.getWheel_diameter_left();
    this.wheel_diameter_right = config.getWheel_diameter_right();
    this.scaling_factor = config.getScaling_factor();
    this.init();
}

public synchronized Pose step(ScanStep sc)
{
    double delta_pos_left = sc.getLeftCount();
    double delta_pos_right = sc.getRightCount();
if (delta_pos_left != 0 || delta_pos_right != 0)
{
    double delta_left = delta_pos_left *
            this.ot.configuration.wheel_conversion_left;
    double delta_right = delta_pos_right *
            this.ot.configuration.wheel_conversion_right;
    double delta_heading = (delta_right - delta_left)/
            this.ot.configuration.wheel_distance;
    double theta = (delta_heading) * 0.5 +
            this.ot.result.getTheta();
    double delta_x = (delta_left + delta_right) *
            0.5 * Math.cos(theta);
    double delta_y = (delta_left + delta_right) *
            0.5 * Math.sin(theta);
    this.ot.result.increaseX(delta_x);
    this.ot.result.increaseY(delta_y);
    this.ot.result.increaseTheta(delta_heading);
}

return new Pose(this.ot.result);

return this.getPose();
}

public synchronized Pose getPose()
{
    return new Pose(this.ot.result);
}
public synchronized void setPose(Pose p)
{
    this.ot.result = p;
}

private void init()
{
    this.ot.configuration.wheel_distance =
        this.axis_wheel_ratio*this.scaling_factor*
            (this.wheel_diameter_left +
                this.wheel_diameter_right)/2;
    this.ot.configuration.wheel_conversion_left =
        this.wheel_diameter_left*this.scaling_factor*
            Math.PI/this.increments_per_tour;
    this.ot.configuration.wheel_conversion_right =
        this.wheel_diameter_right*this.scaling_factor*
            Math.PI/this.increments_per_tour;
}

private class OdometryConfig
{
    public double wheel_distance = 0.0;
    public double wheel_conversion_left = 0.0;
    public double wheel_conversion_right = 0.0;
}

private class OdometryState
{
    public double pos_left_prev = 0.0;
    public double pos_right_prev = 0.0;
}

private class OdometryTrack
{
    public OdometryConfig configuration = new OdometryConfig();
    public OdometryState state = new OdometryState();
public Pose result = new Pose(0,0,0);

}
Appendix B

Programs

B.1 Sensor Monitor Program

#Python modules
import sys
import threading
import socket
import time

#project modules
from SLAMController import SLAMController
from SLAMSensorMonitor import SLAMSensorMonitor

#framework modules
sys.path.append('lib/Framework')
from FischerTechnik import *
from Movement import Movement
from Sensors import Sensor_Manager

#project utility modules
sys.path.append('lib/SLAM')
from CommonUtils import *
from OdometerUtils import Odometer
from TBFUtils import UltraSonic, TBF

#Defined for framework. Sets up the physical hardware config
#Both bots need to be set up the same way
class Solution_Constants(FT_Constants):
    robotbuild_motor_manager = (Movement, FT_Robot, FT_Constants)
    robotbuild_motors = [(FT_Motor, "left", 1,
                          FT_Constants.orientation_left),
                          (FT_Motor, 'right', 2,
                          FT_Constants.orientation_right)]
    robotbuild_sensor_manager = (Sensor_Manager, FT_Robot,
                                  FT_Constants)
    robotbuild_sensors = [(FT_Distance_Sensor, "left_ultrasonic",
                         2), (FT_Distance_Sensor, "right_ultrasonic", 1),
                         (FT_Counter, "left_motor", 1), (FT_Counter, "right_motor",
                         2)]

#Main program

def main():
    #create an instance of the robot
    track_bot = FT_Robot(FT_Constants ct, Bluetooth, 8)
    #fetch the controller for that robot
    ctrl = track_bot.build_robot(Solution_Constants)
    ctrl["Motor_Manager"].braking = 1
    #create the two main threads
    #A thread to monitor the movement and sensors of the robot
    sm = SLAMSensorMonitor(ctrl)
    #A thread to control the robot over network
    sc = SLAMController(ctrl)
    #start the threads
    sm.start()
    sc.start()
    #wait for user to end program
    print("Press enter to end program")
    end = input()
    #shutdown
    sc.stop()
    sm.stop()

if __name__ == '__main__':
    main()
B.2 Offline SLAM pose and landmark tracking

package slam;

//Common Utilities
import CommonUtils.Point;
import CommonUtils.Pose;
import CommonUtils.ScanStep;
import CommonUtils.Configuration;

//Odometry imports
import OdometryUtils.FT_Curve_Odometer;
import OdometryUtils.FT_Odometer;
import OdometryUtils.Odometer;

//TBF imports
import TBFUtils.TBFCalculator;

//Graphic output imports
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class SLAMBots {

    private Odometer odometer = new FT_Odometer();
    private TBFCalculator tbfc = new TBFCalculator();
    private ArrayList<ScanStep> steps = new ArrayList<ScanStep>();
    private ArrayList<Pose> bot_position = new ArrayList<Pose>();
    private ArrayList<Point> landmarks = new ArrayList<Point>();
    private String data_file = "scan_run3.dat";

    public void start()
    {
        Configuration conf = new Configuration();
        this.init();
    }
}
if (this.steps.isEmpty())
    System.exit(1);
SLAMDisplay sd = new SLAMDisplay();
sd.addWindowListener(new WindowCloser());
sd.setTitle("SLAM\_calculator");
sd.pack();
sd.setVisible(true);
sd.setBot\_position(this.bot\_position);
sd.setLandmarks(this.tbfc.getLandmarks());
for (int i = 0; i < this.steps.size(); i++)
{
    ScanStep sc = this.steps.get(i);
    //odometry
    Pose p = this.odometer.step(sc);
    this.bot\_position.add(p);
    //triangulation
    this.tbfc.setPose(p);
    this.tbfc.calculate(sc);
    sd.update();
    try {Thread.sleep(20);} catch (Exception e)
    {
        System.out.println("Thread\_Exception");
    }
}

private void init()
{
    this.tbfc.setPose(new Pose(0,0,0));
    this.tbfc.setOffset(new Point(8,2.4), new Point(8,-2.4));
    this.readFile();
}

private void readFile()
{
    try
    {
        BufferedReader in = new BufferedReader(
            new FileReader(this.data\_file));
    }
String line = new String();
while((line = in.readLine()) != null && line.length() > 0)
{
    String[] result = line.split("\s");
    int seq = Integer.parseInt(result[0]);
    String bot = result[1];
    int delta_left = Integer.parseInt(result[2]);
    int delta_right = Integer.parseInt(result[3]);
    int range_left = Integer.parseInt(result[4]);
    int range_right = Integer.parseInt(result[5]);
    this.steps.add(new ScanStep(seq, bot, delta_left, delta_right, range_left, range_right));
}

} catch (IOException e)
{
    System.out.println("Failed to open data file :");
    + e.toString());
}

public static void main(String[] args) {
    SLAMBots sb = new SLAMBots();
    sb.start();
}

private static class WindowCloser extends WindowAdapter
{
    @Override
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
Appendix C

Mathematical steps performed by the algorithms implemented

C.1 Odometry

Change in wheel position:

$$\Delta P_k = \Delta C_k \times \frac{d_k \pi}{i}$$

Where $\Delta P$ refers to the change in position, $k$ refers to the specific wheel index, $C$ is the differential encoder count, $d$ is the wheel diameter and $i$ is the number of encoder ticks recorded per full revolution on the wheel.

Change in heading:

$$\Delta \theta = \frac{\Delta r - \Delta l}{ad}$$

Where $\Delta \theta$ refers to the change in heading, $\Delta r$ and $\Delta l$ refer to the change in right and left encoder counts, $a$ is the ratio of the distance between the wheels and the wheel diameter, and $d$ is the wheel diameter.

Average heading:

$$\delta = \Delta \theta \times 0.5 + \theta_{t-1}$$

The average heading is calculated by multiplying half of the change in heading at time $t$ and adding it to the last known heading, that is, the heading at time $t - 1$. 

76
New robot position:

\[ x_t = x_{t-1} + ( (\Delta P_l + \Delta P_r) \times 0.5 \times \cos(\delta) ) \]
\[ y_t = y_{t-1} + ( (\Delta P_l + \Delta P_r) \times 0.5 \times \sin(\delta) ) \]
\[ \theta_t = \theta_{t-1} + \Delta \theta \]

C.2 TBF

C.2.1 Triangulation

Some required values:

\[ \Delta x = S_{rx} - S_{lx} \]
\[ \Delta y = S_{ry} - S_{ly} \]
\[ d^2 = \Delta x^2 + \Delta y^2 \]
\[ d = \sqrt{d^2} \]
\[ k = \frac{d^2 + r_2 - r_1}{2xd} \]

Where \( S \) is the sensor position and \( r \) is the range value of a sensor.

Points of intersection:

\[ P_{ix} = S_l + \frac{\Delta x_k}{d} \pm \frac{\Delta y}{d} \times \sqrt{r_l^2 - k^2} \]
\[ P_{iy} = S_l + \frac{\Delta y_k}{d} \pm \frac{\Delta x}{d} \times \sqrt{r_l^2 - k^2} \]

Two points of intersection may be created and will be checked for validity using the following equations.

Determining the visible point:

For \( \forall P \) as \( (i) \) and \( \forall S \) as \( (k) \):

\[ \Delta x = P_{ix} - S_{kx} \]
\[ \Delta y = P_{iy} - S_{ky} \]
\[ \delta = \arctan(\Delta y, \Delta x) \]
\[ \delta \in (S_{k\theta} - \frac{S_{k\Theta}}{2}, S_{k\theta} + \frac{S_{k\Theta}}{2}) \]

The symbol \( \Theta \) represents the arc width of the sensor. This method checks that the angle between the sensor and the point falls within the viewable area of the sensor.
C.2.2 Point refinement

Calculating the average position of the points:

\[ T_x = 0 \]
\[ T_y = 0 \]
\[ T_t = 0 \]

For \( \forall P \) as \( i \):

\[ T_x = T_x + P_{ix} \]
\[ T_y = T_y + P_{iy} \]
\[ T_t = T_x + 1 \]

The position of the landmark:

\[ x = \frac{T_x}{T_t} \]
\[ y = \frac{T_y}{T_t} \]

\textit{Triangulations} = \( T_t \)