A SIMULTANEOUS LOCALIZATION AND MAPPING IMPLEMENTATION USING INEXPENSIVE HARDWARE

by

TODD MICHAEL AYCOCK

A THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Electrical and Computer Engineering in the Graduate School of The University of Alabama

TUSCALOOSA, ALABAMA

2010
ABSTRACT

Autonomous mobile robots have become more popular over the past few decades, influencing both industry and academia. The strategy of making robots navigate autonomously adds many problems however. Many of these problems are directly related to the robot’s ability to localize and autonomously map its environment. A solution to this problem is called simultaneous localization and mapping (SLAM).

SLAM is the concept of localizing the robot while simultaneously generating a map of the environment, and then using the map in subsequent localization steps. The success of SLAM lies in a filter algorithm. One of the more common and successful filters is the extended Kalman filter (EKF), and there are many different algorithms that could be used to implement this filter. However, the computational complexity and physical cost of implementing the algorithm place the SLAM solution beyond the scope of many low-cost robotics projects.

This thesis analyzes many of these cost issues related to the implementation of SLAM on autonomous robots. First, the types of sensing hardware are discussed, and potential low-cost solutions are suggested. Next, timing aspects of two different methods for data association are examined in order to evaluate tradeoffs between speed and accuracy. Finally, optimizations to the filter’s update step involving matrix multiplication are presented. These three changes are presented as a customized EKF SLAM algorithm, called inexpensive hardware SLAM (IH-SLAM), which is applicable to small-scale robotics applications.
DEDICATION

This thesis is dedicated to everyone who helped me through the process of creating this manuscript, particularly my family, friends, and advisor Dr. Kenneth Ricks.
### LIST OF ABBREVIATIONS AND SYMBOLS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>One dimensional</td>
</tr>
<tr>
<td>2D</td>
<td>Two dimensional</td>
</tr>
<tr>
<td>3D</td>
<td>Three dimensional</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog to digital converter</td>
</tr>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application specific integrated circuit</td>
</tr>
<tr>
<td>BED</td>
<td>Beam edge detection</td>
</tr>
<tr>
<td>cm</td>
<td>Centimeters</td>
</tr>
<tr>
<td>DSM</td>
<td>Decoupled stochastic mapping</td>
</tr>
<tr>
<td>DTMO</td>
<td>Detection and tracking of moving objects</td>
</tr>
<tr>
<td>EKF</td>
<td>Extended Kalman filter</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field programmable gate array</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>HWIL</td>
<td>Hardware-in-the-loop</td>
</tr>
<tr>
<td>IH-SLAM</td>
<td>Inexpensive hardware simultaneous localization and mapping</td>
</tr>
<tr>
<td>INS</td>
<td>Inertial Navigation System</td>
</tr>
<tr>
<td>IR</td>
<td>Infrared</td>
</tr>
<tr>
<td>KF</td>
<td>Kalman filter</td>
</tr>
<tr>
<td>LASER</td>
<td>Light amplification by stimulated emission of radiation</td>
</tr>
<tr>
<td><strong>MATLAB</strong></td>
<td>Matrix Laboratory software</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td><strong>m</strong></td>
<td>Meters</td>
</tr>
<tr>
<td><strong>ms</strong></td>
<td>Milliseconds</td>
</tr>
<tr>
<td><strong>RAM</strong></td>
<td>Random access memory</td>
</tr>
<tr>
<td><strong>RANSAC</strong></td>
<td>Random sampling consensus</td>
</tr>
<tr>
<td><strong>SLAM</strong></td>
<td>Simultaneous localization and mapping</td>
</tr>
<tr>
<td><strong>SONAR</strong></td>
<td>Sound navigation and ranging</td>
</tr>
<tr>
<td><strong>μs</strong></td>
<td>Microseconds</td>
</tr>
<tr>
<td><strong>s</strong></td>
<td>Seconds</td>
</tr>
<tr>
<td><strong>UKF</strong></td>
<td>Unscented Kalman filter</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I am pleased to have this opportunity to thank the many colleagues, friends, and faculty members who have helped me with this research project. I am extremely thankful to Dr. Kenneth Ricks, the chairperson of this thesis and my faculty advisor. His advice and expertise was extremely helpful in the organization and solution of this project. I would also like to thank my other committee members, Dr. Jeff Jackson and Dr. Monica Anderson, for their important input, stimulating questions, and support of both the thesis research and academic class work.

This research would not have been possible without the support of my fellow graduate students and friends. Also, my family provided encouragement and support, and helped tremendously by proofreading the entire document. Finally, I would like to thank the faculty in the Mathematics and Computer Science departments who provided me with the knowledge to understand the complicated matrix calculations and mathematical models, as well as the algorithmic nuances needed to fully comprehend this research.
## CONTENTS

ABSTRACT ................................................................................................................................. ii

DEDICATION .................................................................................................................................. iii

LIST OF ABBREVIATIONS AND SYMBOLS ........................................................................ iv

ACKNOWLEDGMENTS ........................................................................................................... vi

LIST OF TABLES ...................................................................................................................... x

LIST OF FIGURES ................................................................................................................... xi

1. INTRODUCTION ................................................................................................................ 1
   1.1 Introduction to Autonomous Navigation ....................................................................... 1
   1.2 Simultaneous Localization and Mapping Background ............................................... 9
   1.3 Implementations of SLAM .......................................................................................... 15
   1.4 SLAM Example ........................................................................................................ 17
   1.5 Problems with SLAM ............................................................................................... 23
   1.6 Research Contributions ............................................................................................ 24
   1.7 Thesis Outline ......................................................................................................... 25

2. KALMAN FILTER ............................................................................................................... 27
   2.1 Introduction to the Kalman Filter .............................................................................. 27
   2.2 Basic Principles ........................................................................................................ 28
   2.3 Filter Operations ...................................................................................................... 30
   2.4 The Extended Kalman Filter .................................................................................. 33
   2.5 Applications .......................................................................................................... 35
LIST OF TABLES

Table 1. KF predict equations ..................................................................................................... 31
Table 2. KF update equations. .................................................................................................... 31
Table 3. EKF predict equations. .................................................................................................. 34
Table 4. EKF update equations ................................................................................................... 34
Table 5. Laser scanner comparison ................................................................................................. 48
Table 6. Sonar sensor comparison. ................................................................................................. 50
Table 7. IR sensor comparison ..................................................................................................... 52
Table 8. IR sensor layouts ............................................................................................................... 60
Table 9. Sonar sensor layouts. ...................................................................................................... 62
Table 10. Sensor configurations for extraction testing. ................................................................. 70
Table 11. Sensor configurations for extraction testing. ................................................................. 86
Table 12. Standard update function matrix multiplications ......................................................... 95
Table 13. Matrix-chain optimization for update function matrix multiplications. ................. 95
Table 14. Temporary storage, T1, optimization for update function matrix multiplications. ..... 96
Table 15. Temporary storage, T2, optimization for update function matrix multiplications. ..... 96
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>An occupancy grid.</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>A feature map.</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>New York subway topological map.</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>SLAM flowchart.</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>Initial robot state at time $k$.</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>Robot state after first incorrect prediction at time $k + 1$.</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>Corrected robot state after first observation and update at time $k + 1$.</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>Robot state after second incorrect prediction at time $k + 2$.</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>Corrected robot state after second observation and update at time $k + 2$.</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>Underlying model of the Kalman filter.</td>
<td>28</td>
</tr>
<tr>
<td>11</td>
<td>Generic EKF-based SLAM model.</td>
<td>37</td>
</tr>
<tr>
<td>12</td>
<td>MATLAB IH-SLAM simulator GUI.</td>
<td>41</td>
</tr>
<tr>
<td>13</td>
<td>Differential drive robot.</td>
<td>44</td>
</tr>
<tr>
<td>14</td>
<td>Maxbotix MaxSonar-EZ1 beam.</td>
<td>51</td>
</tr>
<tr>
<td>15</td>
<td>Devantech SRF05 beam.</td>
<td>51</td>
</tr>
<tr>
<td>16</td>
<td>Devantech SRF08 beam.</td>
<td>51</td>
</tr>
<tr>
<td>17</td>
<td>Sharp GP2Y0A02YK IR sensor beam.</td>
<td>53</td>
</tr>
<tr>
<td>18</td>
<td>Typical laser scanner data.</td>
<td>56</td>
</tr>
<tr>
<td>19</td>
<td>Test 1 environment.</td>
<td>68</td>
</tr>
<tr>
<td>20</td>
<td>Test 2 environment.</td>
<td>68</td>
</tr>
</tbody>
</table>
Figure 42. Hardware IH-SLAM implementation.................................................. 101
CHAPTER 1
INTRODUCTION

1.1 Introduction to Autonomous Navigation

An autonomous robot is a mechanical, artificial agent that can perform desired tasks without continuous human guidance. Based on this definition, most robotic tasks can be considered autonomous. Generally, for a robot to be considered fully autonomous, it must be able to gain environmental information, perform useful amounts of work for an extended period of time, navigate itself throughout its environment (if it is mobile), and ensure the safety of people, property, and itself (unless it is designed to be harmful). All of these tasks must be performed without human intervention and with respect to time and space constraints.

Autonomy for robots can come in a variety of different forms, but for mobile robots, one of the most important tasks is navigation. To collect information about its environment or perform its specified workload, a mobile robot must be able to navigate effectively. For an autonomous robot, navigation is accomplished by one or more of the following steps: sensing the environment, modeling the environment, planning a path, and acting.

Autonomous navigation has been greatly improved upon over time. Some of the earliest successful applications involved line following. Line following is the process of guiding a robotic vehicle along a fixed path using an observable track that the robot can recognize and
follow. A variation of line following used mainly in indoor applications is wall following. With wall following, the robot follows the edge of a wall instead of a track on the ground.

Line following is extremely simple and requires simple sensor hardware and small amounts of processing. This is desirable in inexpensive applications, but there are some significant disadvantages to this approach. Since the robot can only follow lines or walls, its operational freedom is severely restricted. Also, the robot will not typically track its current position in the environment since it has limited sensory observations.

Another problem with line following is the limitations of using a track, which confines the robot to a limited environment. If the robot becomes disconnected from the track, it becomes instantly lost and can no longer effectively navigate. The track itself can become damaged or ineffective, and if the tracks cross each other the robot can become confused. Line following can also be impractical since the tracks usually must be laid out beforehand, and the flexibility of this method is limited since the path cannot be easily changed. Therefore, the line following technique can be successful in extremely simple applications, but it is also very restrictive and ineffective in more complicated applications.

A robot that is free to move anywhere in its environment requires localization to keep track of its position at all times. Localization is the process of estimating the position (or pose) of a robot in either the local or global frames. In general, there are four main types of localization: dead reckoning, the Global Positioning System (GPS), \textit{a priori} map localization, and SLAM. Each type of localization becomes more complicated but also more accurate.

The most basic form of localization is dead reckoning, which is simply the estimation of a robot’s pose through sensing of its motion. Dead reckoning localization is implemented using some form of proprioceptive sensor, which only senses the position, orientation, speed, or
acceleration local to the robot and does not sense the environment. One of the more common methods for implementing dead reckoning localization is odometry. Odometry is the use of data from the movement of the robot’s actuators to estimate change in position over time. For example, a wheeled robot will measure the distance traveled or velocity of one or more wheels in a given period of time and calculate the change in position accordingly. Another similar form of dead reckoning localization is an Inertial Navigation System (INS). Inertial Navigation Systems measure the forces experienced during acceleration and deceleration of a robot. These forces can be measured using gyroscopes or accelerometers. Inertial Navigation Systems are typically used in systems where odometry is difficult to implement, such as legged and flying vehicles. However, an INS can produce rather noisy measurements, and many of the more reliable systems are very expensive.

Dead reckoning localization suffers from non-deterministic errors. For odometry-based systems, friction inconsistencies between the robot and the ground, i.e. slippage, create many of these errors, leading to inaccurate \([x, y, \theta]\) position calculations. Noise corrupts INS-based systems, and more reliable systems are very expensive. While errors are observed in any navigation system, dead reckoning errors are often amplified due to the fact that they can accumulate over time.

To correct these problems, a robot needs a way to relate its position to something static and known to the robot: the environment. This relationship is developed through the use of exteroceptive sensors (sensors that give information about the surrounding environment). One of the more recent and popular methods for localizing using the environment is GPS.

GPS is a satellite global navigation system originally designed for military use. Now, the system is available to the public and provides reliable positioning and timing services worldwide.
on a continuous basis. The GPS satellites broadcast signals from their orbits, and GPS receivers use these signals to calculate a three-dimensional (3D) location. To effectively calculate this position, a minimum of four satellite signals must be detected at the receiver. Based on the transit time of each received signal and the position of the satellites that transmitted each original signal, trilateration or multilateration techniques can be used to estimate the location of the receiver. Under a clear sky, horizontal position fixes are accurate up to 5m for common commercial GPS receivers. However, GPS does not work in indoor applications, and GPS navigation is best suited for autonomous robots that travel long distances at fast speeds, where the larger resolutions are more acceptable. Errors in GPS-based systems do not accumulate like errors from dead reckoning localization. Nonetheless, for smaller outdoor and indoor applications, GPS is not a feasible solution to localization.

Localization using a map can successfully solve the problem of accumulating errors with dead reckoning while providing much better resolution than GPS. In conjunction with dead reckoning, map-based localization can be very effective in repeatedly minimizing the errors that would normally accumulate through dead reckoning localization. This is because maps provide absolute, static information with which to correct the uncertain measurements from dead reckoning.

To localize with a map, a map must first be built. This process is called map building or mapping, and it involves the integration of environmental information into a model which symbolically depicts the environment. Mapping also involves dealing with the complexity of this model and depicting the environment in an accurate representation that can be easily recognized. This is accomplished by highlighting relationships between elements, such as objects or regions. There are many important aspects to mapping, such as environment
representation and interpretation of observations. The goal is to represent the environment in a manner that is useful to the robot for localization or path planning. There are three main types of maps commonly used in localization systems: occupancy grids, feature maps, and topological maps.

Occupancy grids are maps that are represented by a matrix where each cell describes the probability that a small, metrically defined region of the environment is occupied by an obstacle. Occupancy grids can work well in reasonably small environments, but data association can become too computationally expensive as the map size grows. Figure 1 is an example of an occupancy grid. The color of each cell represents the probability of the cell being occupied by an obstacle, with black representing 100% probability, white representing 0% probability, and grey representing varying degrees of uncertainty.

![Figure 1. An occupancy grid.](image)

Feature maps represent the environment using metrically defined parametric features. These features are also called landmarks. Specifically, landmarks are defined as features which can be easily re-observed and distinguished from the rest of the environment and other landmarks. These landmarks can range from points to lines to geometric shapes. The use of
these landmarks introduces the problem of data association, which is the process of finding the correspondence between elements in two different sets of data. In this case, data association is matching landmarks from a map to landmarks sensed in the environment. An example of a feature map can be seen in Figure 2. The L’s represent landmarks, the blue circle represents the robot, and the black lines represent the current detection of landmarks by the robot’s sensors.

![Figure 2. A feature map.](image)

The determination of landmarks is an important aspect of feature-based map building. Landmarks should be easily re-observable in order to be useful to a robot in the environment. They should be also observable from different positions and different angles. A landmark that is not easily re-observable can turn out to be useless to a robot that is attempting to localize itself via a map. Landmarks should also be unique and distributed widely enough to be easily distinguished from other landmarks. If a robot confuses two different landmarks, the results can
be disastrous. Finally, landmarks must be stationary, so a robot can localize relative to a fixed position.

The final map type, topological, does not make use of metric measurements but instead represents the environment in terms of places and connecting paths. Topological maps are simplified to only contain vital information by removing unimportant details. Typically, a topological map will lack scale, direction, or distance (metric measurements), but the relationships between points are maintained. These types of maps are usually depicted by a graph structure where the graph nodes represent locations in the environment and the graph edges represent procedural information for traveling between nodes. A good example of a topological map is a map of the New York subway system, provided in Figure 3.

![Figure 3. New York subway topological map [1].](image)

With a topological map, navigation is determined by a sequence of transitions between nodes equivalent to path finding in graph theory. This map only works on the assumption that distinctive places are locally distinguishable from surrounding places and that the procedural information given by the map is sufficient to allow the robot to travel safely to the specified location. In topological maps, data association is accomplished using place recognition, where the matching of observations to a map is based on the similarity between the observed
information data set and the graph node description data set. This form of association is functionally different than metric-based association used in feature maps.

Topological maps are attractive in that they efficiently and compactly represent the environment. They also make it very easy to perform tasks like path finding and planning. While topological maps have a number of positive aspects, they also have a few weaknesses. Without the aid of metric measurement, it becomes slightly more difficult to ensure reliable navigation between places, especially in more complex environments. Also, a more critical weakness is place recognition. As with landmarks in feature maps, distinctive places in topological maps must be unique and easily re-observable. These maps contain descriptive information about the distinctive places, and this information must be useful to an exploring robot. A final problem with topological maps lies with their limitations to movement. Typically the robot will only be instructed to follow specific trajectories between places and is therefore not allowed to freely roam within the environment. This restriction is acceptable in some scenarios but is not ideal.

The simplest way to implement localization using a map is to use an a priori map, or a previously known map. This type of localization only works in a previously explored (and mapped) environment. A robot must have a full map to work with before it can navigate autonomously. One benefit of using an a priori map is that the robot is freed from the task of building the map, thereby reducing its computational load. However, environments are not always static over time, and thus, a previously created map can be obsolete at the time of its use in navigation. Also, the map would have to be created in a non-autonomous manner due to the problems with dead reckoning localization.
These problems are the main motivation for simultaneous localization and mapping (SLAM). SLAM overcomes the need for an \textit{a priori} map by physically constructing the map in real-time in conjunction with localization. The combination of these two processes actually allows the flaws of each process to be masked somewhat by redundancy. The actual iterative process of SLAM can be defined in the following manner:

- The robot localizes itself using dead reckoning methods. Errors slowly accumulate over time leading to a growing uncertainty in the robot’s position.
- Simultaneously, as landmarks are observed, the robot builds a map. However, this map is not the same as a normal \textit{a priori} map, because each landmark’s position is only as certain as the robot’s uncertainty at the time the landmark was last observed.
- When landmarks are re-observed, the robot uses \textit{a priori} map based localization on the uncertain map to correct the errors from dead reckoning. Over time, as landmarks are repeatedly re-observed, the robot becomes more confident in its position, and eventually, the map converges into the certainty of an \textit{a priori} map.

In this process, both mapping and localization occur in a single time step, hence the name simultaneous localization and mapping. The goal of the SLAM process is to have the robot autonomously generate a map of an unknown environment into a highly certain \textit{a priori} map that can be used for localization and path planning. This approach does require more computation by the robot since it is responsible for mapping and localization.

1.2 Simultaneous Localization and Mapping Background

The theoretical beginnings of SLAM began at the 1986 IEEE Robotics and Automation Conference. Researchers at the conference, including Peter Cheeseman, Jim Crowley, and Hugh Durrant-Whyte, had been attempting to apply estimation methods to localization and mapping problems [2]. From the conversations at this conference, a number of papers were published that showed a high degree of correlation between landmark location estimates. These papers included research on representation and estimation of spatial uncertainty by R.C. Smith and
Cheeseman [3] and research by Durrant-Whyte into the manipulation of geometric uncertainty [4]. This research combined with research on Kalman filter (KF) navigation algorithms led to the paper by Smith, Self, and Cheeseman [5], which showed that the estimates of observed landmarks were all necessarily correlated because of the common error in the estimated robot location. While most research had been based on either localization or mapping as independent problems, a theoretical breakthrough was made with the realization that the combined mapping and localization problem was convergent, and that as the correlations between landmarks grew, the solution improved. This breakthrough led to the coining of the acronym SLAM and the increased interest in the combination of both localization and mapping problems.

Once the SLAM problem was identified, intensive work began on finding a fully satisfying solution, including [6], [7], [8], [9], and [10]. On a theoretical basis, SLAM is considered a solved problem. However, in practice, a physical SLAM implementation introduces a whole new set of problems, which are identified in Section 1.3.

The SLAM process consists of a number of individual steps. Specifically, the steps are localization and prediction; observation and landmark extraction; data association; filter update; and map insertion. These steps are individual components that are common to most SLAM implementations. An example of the flow of information between the different steps is presented in Figure 4.
Localization and Prediction

The first step in any SLAM approach is the prediction, or estimation, of the robot’s state. This prediction is common to all SLAM implementations independent of the type of filter used. The prediction is based upon known information about the robot’s own motion model combined with the information gathered about the robot’s movement through localization sensors. The robot’s localization system is typically modeled in order to relate it mathematically to whatever filter the SLAM implementation is using. The information gathered about the robot’s movement must also be modeled in order to relate it mathematically to the filter used. This information about the robot’s movement can be gathered using dead reckoning localization or by using the control signals sent to the robot’s localization system.

During the prediction step, the uncertainty of the robot’s position is also updated. The maximum error can be approximated based on the movement of the robot and a predetermined error model. This error accumulates, and the uncertainty of the robot’s position accumulates as well. The manner of computing and representing this uncertainty varies based on the SLAM implementation and the type of filter used, and the error and system dynamic models change.
based on the expected environment and the robot’s specific platform and hardware. In addition to the uncertainty of the robot’s position, changes may be made to the map as well. Depending on the implementation, changes to the uncertainty of the robot’s position or changes to the position itself may affect the stored positions and uncertainties of the landmarks. Here, landmarks can be distinctive places, features, or points in an occupancy grid, depending on the specific type of map used. In all, a number of very important changes occur during the prediction step of any SLAM implementation.

**Observation and Landmark Extraction**

The next step that typically occurs in the SLAM process can actually be broken into two sub-steps: environmental observation and landmark extraction. Both of these sub-steps are entirely dependent on the type of environmental sensor hardware and type of map used in the SLAM implementation. The type of filter used in the SLAM implementation does not typically affect these sub-steps, which only provide data to the next step, data association.

Environmental observation occurs by a robot observing the environment with one or more types of exteroceptive sensors. The type of sensor used can vary from range measurement to vision, and the common factor for all sensors is their ability to make useful observations of the environment suitable for landmark extraction. However, the frequency and types of data gathered can widely vary from sensor to sensor.

Landmark extraction is a very complicated task, and it is not only dependent on the types of sensors used but also on the type of map used in the implementation. Based on the observations gathered from the robot’s sensors, landmark extraction is the process of reliably extracting landmarks from the observations for the purpose of inserting them into a map or matching them with other previously stored landmarks in a map. Because of this, landmark
extraction is vitally important to the success of SLAM. Failure of the landmark extraction step can be destructive to the success of the algorithm, even if all other steps of the process are working correctly. Therefore, the most important characteristic of the landmark extraction step is that it extracts a sufficient amount of quality landmarks, despite the types of landmarks used, the type of map used, or the type of sensor data provided. The overall result from the observation and landmark extraction step should be an accurate list of landmark positions that were extracted from observations of the environment.

**Data Association**

After the observation step is completed, the results are analyzed in the data association step. Data association is the process of matching observed landmarks with those previously stored in the map. This process is also referred to as re-observation of landmarks. Extracted landmarks that are not matched to corresponding landmarks in the map are considered newly observed. These newly observed landmarks are sent to the map insertion process. Associated landmarks are conversely used in the filter update step.

In practice, there are a few problems that can arise during data association. First, an observable landmark may not be matched to its map counterpart. This problem is called a false negative and can occur because the landmark is not extracted from the sensor data or because the extracted landmark cannot be easily matched to the stored landmark. Generally, if this problem occurs, the corresponding landmark is bad and should not be used in the algorithm. Second, a landmark may be observed once and never observed again. This case is troubling in that a useless landmark takes up memory space and affects the execution time of the algorithm. Again, this problem usually indicates a bad landmark that should not be used in the algorithm. These first two problems can be solved by redefining a more suitable landmark extraction policy in
order to minimize the number of bad landmarks extracted. The third problem that can occur is a false positive, where a landmark is wrongly associated to another previously observed landmark. This problem is destructive to the SLAM algorithm, because the robot localizes relative to the wrong stored landmark.

Data association, like landmark extraction, is dependent on the type of map used. Additionally, there are different ways to implement data association. Most algorithms vary in the way they match values in the two data sets.

**Filter Update**

The filter update step is perhaps the backbone to any SLAM implementation. A filter removes unwanted components from a signal. In SLAM, the unwanted component is the error (noise) from using dead reckoning localization. Therefore, the goal of the filter update step is to use the data from the prediction, observation, landmark extraction, and data association steps to remove the errors in the robot’s estimated location.

The filter update step will vary based on the type of filter used. The most common type of filter used in SLAM is the extended Kalman filter (EKF). Other types of filters include the unscented Kalman filter (UKF), the information filter, and various types of particle filters. Regardless of the type of filter used, the update step is important, as it corrects the estimated robot’s position based on the information from the previous steps.

**Map Insertion**

The final step in SLAM is the map insertion step. In this step, unassociated landmarks from the data association step are inserted into the map. Again, this step will vary based on the types of landmarks used and the type of map provided. Also, in this step, initialization information for each landmark is added to the filter, specifically information about the
uncertainty in the landmark’s location. The conclusion of this step finalizes a single iteration of the SLAM process.

While some of these steps are dependent on each other, not all of the steps will occur in a single iteration. Specifically, data association will not occur if the landmark extraction step is unable to extract any landmarks. Likewise, the filter update step will not occur if data association is unable to match any landmarks to the map. Map insertion is also dependent on data association, such that if all extracted landmarks are matched to landmarks in the map, then no new landmarks need to be added.

1.3 Implementations of SLAM

While the concept of SLAM is a unified one, the actual mathematical and physical implementation can vary widely. Mathematically, different types of filters can be used to implement the theoretical SLAM case. Also, the individual filters themselves have a large amount of variation to them. More importantly, SLAM relies on a number of different types of platforms, sensors, and other hardware, and individual systems may alter the SLAM algorithm to work with their individual characteristics. This section will discuss the basic mathematical approaches to SLAM and their differences and will also introduce the main hardware characteristics that can affect the SLAM implementation.

One of the most common statistical solutions to SLAM is the KF, specifically the EKF. Typically, errors from localization are modeled using probabilistically defined noise, and the KF provides a good method for filtering this noise. Most EKF SLAM algorithms are feature based and therefore use feature maps. Also, EKF SLAM algorithms generally use the maximum likelihood algorithm or the nearest-neighbor algorithm for data association, which will be
described in detail in Chapter 6. Since the concept of SLAM was introduced, EKF SLAM has been the de facto statistical method for SLAM implementations. Implementations using the EKF have been proven in practice in a number of cases, including [2] and [11]. A key limitation to EKF SLAM is its computational complexity. For the EKF, the filter update step requires time quadratic in the number of landmarks to compute. This complexity is because of the large covariance matrix used in the EKF algorithm, which must be updated every step through matrix operations.

A newer, more efficient SLAM algorithm is called FastSLAM. FastSLAM uses a modified particle filter, called the Rao-Blackwellized particle filter, instead of the EKF. Each particle possesses a certain number of EKF’s that estimate specific landmark locations. Therefore, only the specifically observed landmarks are updated during the filter update step. This is in contrast to the EKF SLAM algorithm, where every landmark is updated during the filter update step. Specifically, EKF SLAM has a complexity of $O(K^2)$ where $K$ is the number of landmarks, while FastSLAM has a complexity of $O(M \log K)$ where $M$ is the number of particles [12]. As a result, FastSLAM typically executes faster than EKF SLAM algorithms. However, statistically, FastSLAM suffers from degeneration problems due to its inability to forget previous information, and it is overly optimistic in its pose estimation. Also, FastSLAM is more complicated to implement since it requires the maintenance of many separate EKF’s. Additionally, FastSLAM discards correlations between landmarks that exist in the standard EKF SLAM algorithm, making it harder for a robot to recognize a location that it has already seen (called closing the loop in robotics) with the FastSLAM algorithm. Nonetheless, FastSLAM has been implemented successfully in [12].
SLAM has been implemented in a variety of different domains using different types of maps. These domains include indoors [13], [14], outdoors [15], [16], airborne [17], and submarine [18], [19]. SLAM has been implemented using topological maps using consistent pose estimation in [20]. Also, SLAM has been implemented using occupancy grid based maps [21].

Versions of EKF SLAM and FastSLAM are the most common forms of SLAM implementations. Other types of filters have been used with SLAM including the UKF [22] and the information filter [23]. Since EKF SLAM is the most common and stable method for implementing SLAM, it is the method used by the implementation presented in the remainder of this thesis.

1.4 SLAM Example

The following example provides a graphical representation of how SLAM works conceptually. The figures in this example are not drawn to actual scale, but the concept behind SLAM is demonstrated. Also, while the robot is shown moving large distances and stopping between observations, this is not usually the case. Typically, a robot will not stop to make observations and will also make many observations per second. Therefore, this example is for representational purposes only and should not be considered accurate in a metric (distance) or time scale.

Take the example of a robot placed in a totally unknown environment. To start, the robot has no map to use for navigation. The robot’s goal is to navigate through the environment with a map that can be used for path planning or localization. SLAM attempts to bind the processes of localization and mapping in a loop that supports the contiguity of both aspects in separated
processes. The redundancy and iterative feedback from both processes enhance the results of both processes individually.

First, the robot's state is initialized as shown in Figure 5. At this initial state, the robot's position in its local frame (relative to the robot) is known, but its position in the global frame (relative to the environment) is not known. The true state is unknown to the robot, while the estimated state is where the robot thinks it is. This example is based on a feature map, although any map type would work with these same concepts applied.

At time $k$, the robot makes an observation of its environment. The black lines represent the robot’s observation of the three independent landmarks. After this observation, the robot adds the three measured landmark positions to its map and also records the uncertainty of each landmark’s position. At this time step, the robot’s position is completely certain since it has not moved yet, but there will be small uncertainties for each landmark based on errors in the measurement hardware.
After making this observation, the robot moves forward and predicts its new state. The prediction is based on information from dead reckoning localization. Figure 6 shows the predicted and actual state of the robot at time step $k + 1$, along with the corresponding environmental observations of the three landmarks. Obviously the prediction was incorrect, as there is a moderate difference between the robot’s true and predicted positions. Therefore, the robot’s position is now highly uncertain and is decidedly less certain than the stored landmarks.

![Figure 6. Robot state after first incorrect prediction at time $k + 1$.](image)

After making an observation at time $k + 1$, the robot can then compare these observations with the landmarks stored in its map. There are a variety of methods for performing data association, but this example will assume that the robot correctly associates all three observed landmarks with the corresponding landmarks in the map. Therefore, the robot is able to note a discrepancy between all three landmark positions. Specifically, the errors between the three observed and stored landmarks are all necessarily correlated with the error between the
true and estimated robot positions. Therefore, through use of filtering, the robot can adjust its estimated position. This adjustment is shown in Figure 7.

It is important to note that this adjustment is not perfect. This imperfection is expected since the corrected state is still only an estimate based on noisy data. Since there are errors in measurements of both the global environment and the robot’s movement, the system will not have complete confidence in either one of the measurements. Typically, the robot will be more confident in its environmental observation measurements than its localization measurements, because observation errors do not accumulate. Balancing the uncertainty of both measurements is important for the filter.

In the filter, the robot gives the three observations more weight than its original incorrect prediction and thus moves the corrected prediction towards the unknown true state. Since three directly related errors were observed in the observations, the observation measurement is given a much higher weight than the simple position estimate. However, due to the possibility of errors
in the observation measurements, some weight is given to the original localization measurements.

After making the first correction, the algorithm repeats. Again, the robot moves forward, and a prediction is made based on dead reckoning localization with a moderate amount of error between the robot’s estimated position and the robot’s actual position. This new time step $k + 2$ is shown in Figure 8.

![Figure 8. Robot state after second incorrect prediction at time $k + 2$.](image)

At time step $k + 2$, the robot has now observed all three landmarks twice, therefore decreasing the uncertainty in their estimated positions in the map. After making another observation for the current time step, the robot has observed landmark $L_2$ three times, and is therefore highly certain of its stored position. Again, there is a noticeable difference between the stored landmark position and the observed landmark position, and this difference is directly correlated with the difference between the true and estimated robot states.
Based on this information, the filter is again used to correct the system state. Even though L₁ and L₃ were unobserved, their positions are slightly adjusted. This adjustment occurs because the landmark positions are correlated to the robot’s position, so when the robot’s position changes, it also affects the landmark locations stored in the map. The final state is presented in Figure 9.

Figure 9. Corrected robot state after second observation and update at time \( k + 2 \).

While the scenario itself is finished, it is important to note two observations. First, the robot’s estimated state at time \( k + 2 \) is more accurate than its estimated state at time \( k + 1 \). Although this could be attributed to corrective noise in the localization measurement error, in this case it is actually due to the multiple observations and the corrections made by the filter. A second important observation is that L₂ has actually been overcorrected. This overcorrection is more common in some types of filters, but in this scenario it is used to show how the corrections made by the filter are still estimations and are not expected to be completely accurate.
After enough time has passed and an acceptable number of re-observations made, the robot should be left with an accurate estimation of its own position as well as an accurate map comparable to an *a priori* map of the environment. Although, this scenario is ideal and does not represent all real-world challenges, with enough knowledge of system dynamics, SLAM can provide very good performance.

1.5 Problems with SLAM

SLAM has been proven theoretically and physically as a useful tool in robotic navigation. While there are many benefits to using the SLAM technique to enhance the accuracy of the robot’s navigation system, there are also drawbacks. Specifically, the main drawbacks to SLAM are the physical cost of hardware required, timing constraints when implemented in real-time, and memory space required to store the map.

Many SLAM implementations use laser scanners to observe their environment. Laser scanners provide a wealth of valuable information for a SLAM implementation, but commercially available laser scanners are also very expensive. Therefore, the basic laser implementation is currently only feasible for large budget projects, eliminating it from use with most commercial robots, hobby robots, and other generally smaller budget robotics projects.

Many of the functions involved with the SLAM algorithm include numerous steps that require matrix multiplication, since a covariance matrix is typically used to correlate stored features in the map to extracted features. Naïve matrix multiplication requires $O(n^3)$ time, and in KF-based SLAM, the value of $n$ increases by two for each new landmark observed. Therefore, with many multiplications being performed, the time required for each operation increases exponentially as more landmarks are added to the map. For a large scale project, this
would not be a problem, since the budget would likely allow for the purchase of high end processors to perform these calculations. There are also other algorithms, such as sub-map techniques and FastSLAM, which can help prevent the matrices from growing too large. However, even with these algorithms, SLAM is still computationally intensive. These computational loads are restrictive for many robotics applications that have strict budget, size, and power limitations.

While this computational constraint presents a major problem, the amount of memory available in a system is perhaps even more important. The EKF algorithm requires not only two large matrices to be stored, but also many temporary matrices for intermediate calculations. Because of the nature of matrix multiplications and the flow of data described in Figure 4, many of these operations cannot be pipelined, and thus large temporary matrices have to be stored for future use. For smaller-scale projects this is, once again, a significant concern, since the size and speed of memory is typically limited.

1.6 Research Contributions

This thesis presents a custom EKF SLAM implementation, called inexpensive hardware SLAM (IH-SLAM), which is suitable for small-scale robotic applications. First, landmark extraction techniques are presented for inexpensive sensor hardware. Next, two data association methods are compared with the goal of reducing the computational load of the algorithm. Finally, matrix multiplication optimizations are presented to again reduce the computational load and memory requirements. These three optimizations of typical EKF SLAM implementations are presented with the goal of solving the problems of cost, processing time, and memory
constraints, thereby creating a SLAM implementation suitable for low-cost, small-scale robotics applications.

1.7 Thesis Outline

Chapter 2 introduces the KF, including the history, characteristics, and importance of the KF as it relates to the SLAM technique.

Chapter 3 presents a description of the IH-SLAM algorithm setup. This also includes a description of the MATLAB simulation used to test the IH-SLAM algorithm.

Chapter 4 presents a detailed description of the localization and prediction step of the IH-SLAM implementation. This includes the robot model, motion model, and localization method used in this implementation.

Chapter 5 focuses on the observation and landmark extraction step of IH-SLAM. This includes characteristics of the environmental sensors used and the landmark extraction algorithms for each sensor type. Additionally, experimental results are presented to compare the different sensor types and ascertain the feasibility of using inexpensive sensor hardware.

Chapter 6 focuses on data association in the IH-SLAM implementation. Two of the more popular algorithms are presented. Additionally, experimental results are presented to compare the two association algorithms with respect to timing constraints.

Chapter 7 presents the filter update step of the IH-SLAM implementation. Optimizations for improving matrix operations are also presented.

Chapter 8 presents the map insertion step of IH-SLAM. The map insertion step remains fairly consistent with the standard EKF SLAM implementation.
Chapter 9 presents conclusions including a summary of the research presented in the previous chapters. Finally, future work is suggested based upon the research presented.
CHAPTER 2
KALMAN FILTER

2.1 Introduction to the Kalman Filter

Since one purpose of SLAM is to correct the robot’s position estimate based on its environmental observations, the obvious numerical solution is a filter. A filter is a process that is designed to remove an unwanted component from a signal or system. In this case, the filter attempts to remove the errors that accumulate from the robot’s dead reckoning localization. Some filters are very simple and are not very applicable to SLAM. SLAM requires a recursive filter, or a filter that reuses one or more outputs as an input, and one of the most efficient recursive filters is the KF.

In 1960, Rudolf E. Kalman published a paper describing a recursive solution to the discrete-data linear filtering problem [24]. The solution, now known as the KF, is a set of mathematical equations that provides a recursive method to use noisy estimates to estimate the state of a linear dynamic system while minimizing the mean of the squared error. In control theory, the filter is commonly referred to as linear quadratic estimation.

Since the KF is a discrete recursive estimator, only the estimation from the previous state and the current measurement are required to generate a new estimate for the current state. No history of past observations or estimations is required, which makes the KF much more efficient
than batch estimation techniques. The fact that only the current state is stored minimizes the amount of processing and memory needed.

The KF is important to the solution of the SLAM problem. To work correctly, SLAM must have a way to estimate the current state based on current observations. The KF provides this capability very efficiently. The following sections go into detail about the operation of the KF and also introduce the more advanced version of the KF that is needed to implement SLAM.

2.2 Basic Principles

The main purpose of the KF is to estimate the current state $x$ of a discrete-time controlled process that also contains a measurement $z$. Based on a linear dynamical system which is discretized in the time domain, the filter is modeled on a Markov chain built on linear operators that have been changed by Gaussian noise. The system state is represented as a vector, and at each discrete time step, a linear operator is applied to the state in order to generate a new state. When the linear operator is applied, noise is integrated with information from the controls of the system. Next, another linear operator mixed with more noise generates observable outputs from the hidden, or true, state. A visual representation of the KF model is shown in Figure 10.

![Figure 10. Underlying model of the Kalman filter [25].](image)

From Figure 10, the filter models at time steps $k$, $k - 1$, and $k + 1$ are shown. The squares represent matrices, the ellipses represent multivariate normal distributions (with the
mean listed first followed by the covariance matrix), and the unenclosed values represent vectors. The model can be broken up into three main sections: “Observed”, “Supplied by user”, and “Hidden”. The “Observed” section represents observations or measurements of the current state made by the system. The “Supplied by user” section represents the process models given by the user in order to match the dynamic system to the KF framework. Finally, the “Hidden” section represents the actual states of the system, of which the system and user do not have knowledge. These actual states are what the KF is attempting to estimate.

First, the “Supplied by user” section must be setup by the user. This includes the specification of the following matrices: \( F_k \), \( H_k \), \( Q_k \), \( R_k \), and \( B_k \), which are given for each time step \( k \). The variable \( u_k \) is a vector representing the control signal sent to the system at time step \( k \). Next the model estimates the true state at time \( k \) from the state at time \( k - 1 \), according to Equation 2.1.

\[
x_k = F_k x_{k-1} + B_k u_{k-1} + w_k
\]  

(2.1)

In Equation 2.1, \( F_k \) is the state transition model, \( B_k \) is the control-input model, and \( w_k \) is the process noise, which is a zero mean multivariate normal distribution with covariance \( Q_k \). At time \( k \), an observation \( z_k \) of the true state \( x_k \) is made, according to Equation 2.2,

\[
z_k = H_k x_k + v_k
\]  

(2.2)

where \( H_k \) is the observation model and \( v_k \) is observation noise, which is also a zero mean multivariate normal distribution with covariance \( R_k \).

The random variables \( w_k \) and \( v_k \) represent the process and measurement noise, respectively, and they are assumed to be normal, white noises that are independent of each other. The probability distribution is given by Equations 2.3 and 2.4 for the process and measurement noises.
\[ p(w) \sim N(0, Q) \]  \hspace{1cm} (2.3) \\
\[ p(v) \sim N(0, R) \]  \hspace{1cm} (2.4)

The noises are assumed to be independent of each other due to the inherent nature of the Markov chain.

To implement a KF in a dynamic system, a user must design the aforementioned matrices in order to fit the filter to the system model. One of the more difficult problems with this step is trying to model all system dynamics. Unmodeled system dynamics will cause the estimation algorithm to diverge, or destabilize. As long as there are no unmodeled system dynamics, the filter algorithm should not diverge.

2.3 Filter Operations

The notation \( \hat{x}_{n|m} \) represents the estimate of \( x \) at time \( n \) based on observations up to and including time \( m \). Based on this notation, the state of the system is represented by the following variables: \( \hat{x}_{k|k} \), the \textit{a posteriori} state estimate at time \( k \) given observations up to and including time \( k \); and \( P_{k|k} \), the \textit{a posteriori} error covariance matrix, which measures the estimated accuracy of the state estimate.

The KF has two distinct phases: Predict and Update. The predict phase uses the state estimate from the previous time step to produce an estimate of the current state. The predicted state estimate is known as the \textit{a priori} state, because it does not include observation information from the current time step. In the update phase, the current \textit{a priori} prediction is combined with current observation information to improve the current state estimate. This improved state estimate is known as the \textit{a posteriori} state estimate due to its knowledge of the observation.
information from the current time step. The equations required to generate the new state
estimate are given below, broken into predict and update phases [26].

Table 1. KF predict equations.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{x}_{k</td>
<td>k-1} = F_k \hat{x}_{k-1</td>
</tr>
<tr>
<td>( P_{k</td>
<td>k-1} = F_k P_{k-1</td>
</tr>
</tbody>
</table>

Table 2. KF update equations.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tilde{y}<em>k = z_k - H_k \hat{x}</em>{k</td>
<td>k-1} )</td>
</tr>
<tr>
<td>( S_k = H_k P_{k</td>
<td>k-1} H_k^T + R_k )</td>
</tr>
<tr>
<td>( K_k = P_{k</td>
<td>k-1} H_k^T \Sigma_k^{-1} )</td>
</tr>
<tr>
<td>( \hat{x}<em>{k} = \hat{x}</em>{k</td>
<td>k-1} + K_k \tilde{y}_k )</td>
</tr>
<tr>
<td>( P_{k</td>
<td>k} = (I - K_k H_k) P_{k</td>
</tr>
</tbody>
</table>

The predict phase starts by applying a control signal \( u_k \) to the system. Next, the system
state estimate is predicted based on this control signal. In Equation 2.5, \( \hat{x}_{k|k-1} \) represents the
state of the system before including information from the current time step \( k \). The state
transition model \( F_k \) is multiplied to the previous state \( \hat{x}_{k-1|k-1} \) in order to relate the previous
state to the current state. This result is added to the product of the control-input model and the
current control vector at time step \( k \). Therefore, the result \( \hat{x}_{k|k-1} \) is estimated based on the
previous state \( \hat{x}_{k-1|k-1} \) and the current control signal \( u_k \).

After calculating the predicted state estimate, the corresponding covariance must be
estimated, shown in Equation 2.6. The variable \( P_{k|k-1} \) represents the a priori covariance matrix.
The state transition model \( F_k \) and its transpose are multiplied to the previous covariance matrix
\( P_{k-1|k-1} \), again projecting the previous and current covariance estimates forward from time step
\( k - 1 \) to time step \( k \). The result is added to the process noise covariance matrix in order to
adjust the uncertainty of the current state based on the expected noise.
Once the predict phase is finished, a measurement \( z_k \) is observed by the system. At this point, the update phase begins in order to correct the predicted system state based on the observed measurement. First, the innovation matrix \( \tilde{y}_k \) is calculated by subtracting the product of the observation model \( H_k \) and \( \hat{x}_{k|k-1} \) from \( z_k \) (Equation 2.7). Next, to calculate the innovation covariance matrix \( S_k \), \( H_k \) and its transpose are multiplied to the predicted covariance matrix \( P_{k|k-1} \), and the result is added to the observation noise covariance matrix (Equation 2.8).

The optimal Kalman gain \( K_k \) is the next matrix calculated in Equation 2.9. Here, the inverse of \( S_k \), the transpose of \( H_k \), and \( P_{k|k-1} \) are all multiplied together. The Kalman gain represents the correction needed for the system state based on the observation. From the Kalman gain and the earlier calculated innovation residual, the updated state estimate and covariance can be calculated in Equations 2.10 and 2.11 respectively.

The updated state estimate \( \hat{x}_{k|k} \) is calculated by multiplying \( K_k \) and \( \tilde{y}_k \). This result is added to the previously predicted state \( \hat{x}_{k|k-1} \) to hopefully correct some of the errors added by the process noise. Finally, the updated estimate covariance \( P_{k|k} \) is calculated by first multiplying \( K_k \) and \( H_k \). This result is then subtracted from the identity matrix \( I \) of the same dimensions as the previous result. This result is finally multiplied to the predicted estimate covariance \( P_{k|k-1} \) to create the updated estimate covariance matrix. The matrices \( \hat{x}_{k|k} \) and \( P_{k|k} \) are used in the next time step \( k + 1 \).

The equations given in Table 1 and Table 2 show the basic step by step calculations of the estimated state for a particular time step \( k \). Specific derivations of the equations can be found in [26].
2.4 The Extended Kalman Filter

The KF addresses the problem of estimating the state of a discrete time controlled system governed by a linear stochastic difference equation. However, the basic KF is limited to a linear assumption. This presents a problem, since many real world systems are complex and nonlinear in either the process model, observation model, or both.

The EKF is a nonlinear version of the KF. In the EKF, the state transition and observation models may be linear or differentiable functions. The general form of these functions is given in Equations 2.12 and 2.13,

\[
\begin{align*}
    x_k &= f(x_{k-1}, u_k, w_k) \\
    z_k &= h(x_k, v_k)
\end{align*}
\]

where the function \( f \) is used to calculate the predicted state from the previous state estimate.

The function \( h \) is used to calculate the predicted measurement from the predicted state. Similar to the original KF formulation, the variables \( x_k \) and \( x_{k-1} \) represent the current and previous state estimates, respectively, \( z_k \) represents the measurement, and \( u_k, w_k, v_k \) represent the control signal, process noise, and observation noise, respectively. The noises are identical to the basic KF, assumed to be Gaussian noises with covariances of \( Q_k \) and \( R_k \).

However, the two functions \( f \) and \( h \) cannot be applied directly to the covariance matrix. Instead, a matrix of partial derivatives, called the Jacobian, must be computed. For each time step, the Jacobian is evaluated based on the current predicted states. The process linearizes the nonlinear function around the current estimate.

Table 3 and Table 4 give the complete set of equations needed to calculate the state estimate for the EKF [26].
Table 3. EKF predict equations.

Predicted (a priori) state estimate: 
\[ \hat{x}_{k|k-1} = f(\hat{x}_{k-1|k-1}, u_k, 0) \] (2.14)

Predicted (a priori) estimate covariance: 
\[ P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + W_k Q_k W_k^T \] (2.15)

Table 4. EKF update equations.

Measurement innovation or residual: 
\[ \tilde{y}_k = z_k - h(\hat{x}_{k|k-1}, 0) \] (2.16)

Innovation covariance: 
\[ S_k = H_k P_{k|k-1} H_k^T + V_k R_k V_k^T \] (2.17)

Optimal Kalman gain: 
\[ K_k = P_{k|k-1} H_k^T S_k^{-1} \] (2.18)

Updated (a posteriori) state estimate: 
\[ \hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \tilde{y}_k \] (2.19)

Updated (a posteriori) estimate covariance: 
\[ P_{k|k} = (I - K_k H_k) P_{k|k-1} \] (2.20)

The equations from Table 3 and Table 4 only vary slightly from the equations in Table 1 and Table 2. The main differences are the Jacobians and the functions \( f \) and \( h \). The functions \( f \) and \( h \) were defined earlier, and in this calculation the value 0 is used to estimate the noise. This is because the noise is not actually known in the calculations. The matrices \( F_k, W_k, H_k, \) and \( V_k \) are the Jacobians. For each matrix value, \([i,j]\), the Jacobians are defined in Equations 2.21-2.24.

\[ F_{k[i,j]} = \frac{\partial f[i]}{\partial x[j]} (\hat{x}_{k-1}, u_k, 0) \] (2.21)

\[ W_{k[i,j]} = \frac{\partial f[i]}{\partial w[j]} (\hat{x}_{k-1}, u_k, 0) \] (2.22)

\[ H_{k[i,j]} = \frac{\partial h[i]}{\partial x[j]} (\hat{x}_k, 0) \] (2.23)

\[ V_{k[i,j]} = \frac{\partial h[i]}{\partial v[j]} (\hat{x}_k, 0) \] (2.24)

While the EKF offers a solution to filtering the noise in a nonlinear system, there is an important flaw to note. In the EKF, the distributions of the random variables are no longer normal after going through their nonlinear transformations. Therefore, unlike the linear basic KF, the EKF is not an optimal estimator. It is simply an ad hoc state estimator that only approximates the optimality of Bayes’ rule by linearization [26]. Also, in the EKF, the estimated covariance matrix tends to underestimate the true covariance matrix and risks becoming
inconsistent without the addition of stabilizing noise. However, even with these problems, the EKF can give acceptable results and is used extensively in navigation systems, GPS, and specifically, SLAM. The EKF is also the filter used in the IH-SLAM implementation.

2.5 Applications

The KF is useful for estimating the state of a system based on known controls and observed measurements. A number of applications specifically use the KF in order to work correctly.

One of the more common applications of the KF is to provide accurate, updated information about the position and velocity of an object given a sequence of observations of the object’s position. A radar application uses the KF in this specific manner. Radar attempts to track a target by gathering information about locations, speed, and acceleration of a target at discrete instances of time. Typically, this data is corrupted by a certain amount of noise. The KF uses the trusted model of the dynamics of the target and the potential system noise to remove the effects of the noise and get a good estimate of the location of the target at the current time. The filter can also be used to estimate the location, velocity, and acceleration of the target at a future time step or at a past time step.

Some similar applications of the KF include an autopilot system, attitude and heading reference systems, inertial guidance systems, satellite navigation systems, GPS, and SLAM. Each of these specific applications can use some form of KF to remove noise and estimate position and other important variables.

Kalman filtering is also important to weather forecasting. This application attempts to predict the state of the atmosphere at a future time. In weather forecasting, a large amount of
system noise exists due to the large number of environmental variables that cannot be modeled. Therefore, the KF uses the known dynamics of a weather model and the typical noise parameters to predict the state of the climate at a future time step. Climate observations made by weather balloons and other measuring instruments provide the measurements needed by the KF to closely predict the future weather patterns.

Another application of Kalman filtering is economics. Economics, particularly macroeconomics and econometrics, attempt to predict the state of economies at a future time. Again, the models used in this prediction are very noisy due to the large number of unknown variables (in this case, humans). Therefore, the KF will attempt to remove this noise by considering the predictable nature of the noise (humans’ decisions). The filter will then predict the state of an economy in a future time step based on the typical noise and observations.
3.1 IH-SLAM Conceptual Overview

Before describing the individual steps involved with IH-SLAM, it is important to understand the matrices and models that relate the data between the five basic SLAM steps that were presented earlier. These steps are repeated here for convenience: localization and prediction; observation and landmark extraction; data association; filter update; and map insertion. The flow of data between the steps is presented in Figure 11.

Figure 11. Generic EKF-based SLAM model
From Figure 11, it is easy to see the chain of functions from the localization and observation sensors to map insertion for one time step. Additionally, information is directed from the map insertion function to the prediction function for use in the next time step. In the figure, the ovals represent sensor hardware, directed lines depict the flow and direction of data, and rectangular boxes represent the five independent functions (or steps) in the typical EKF-based SLAM algorithm.

The keys to any EKF-based SLAM implementation are the system state ($X$) and covariance ($P$) matrices. $X$ contains the position of the robot ($x_r, y_r, \theta_r$) and the position of each landmark ($x_l, y_l$). Specifically, $X$ is organized as shown in Equation 3.1.

$$X = \begin{bmatrix}
    x_r \\
    y_r \\
    \theta_r \\
    x_1 \\
    y_1 \\
    \vdots \\
    x_n \\
    y_n
\end{bmatrix} \quad (3.1)$$

The first three rows represent the robot position, while the remaining rows represent each consecutive landmark position, from 1 to $n$, where $n$ is the current number of landmarks. Therefore, the size of $X$ is calculated in Equation 3.2.

$$size(X) = 3 + 2n \quad (3.2)$$

Typically, location values are stored in meters, millimeters, inches, or feet. In IH-SLAM, these values are stored in meters. Additionally, bearing values (such as $\theta_r$) are stored as radians, with range $[0, 2\pi]$.

The covariance matrix, $P$, is probably the most important matrix to an EKF-based SLAM implementation. The covariance of two variables measures how strongly correlated the two variables are. Therefore, the covariance matrix represents the degree of linear dependence...
between variables, specifically the landmarks and robot. It contains the covariance between the robot position, covariance between each landmark’s position, covariance between the robot position and each landmark, and the covariance between different individual landmarks. The covariance matrix itself can be broken down into many smaller sub-matrices as shown Equation 3.3. These sub-matrices are not of equal dimension.

\[
P = \begin{bmatrix}
A & C & \cdots & \cdots \\
B & D & \cdots & F \\
\cdots & \cdots & \cdots & \cdots \\
\cdots & E & \cdots & G
\end{bmatrix}
\]  \hspace{1cm} (3.3)

The first sub-matrix, \(A\), contains the covariance on the robot position and is a 3 × 3 matrix. Sub-matrices \(B\) and \(C\), each the transpose of the other, represent the covariance between the robot state and the first landmark as a 2 × 3 matrix and a 3 × 2 matrix, respectively. These sub-matrices continue down column 1 and across row 1 to the final landmark representing the covariance between the robot state and respective landmark. The sub-matrix \(D\) contains the covariance of the first landmark as a 2 × 2 matrix. These sub-matrices continue diagonally to \(G\), which contains the covariance of the last landmark. Finally, \(E\) and \(F\) are the covariances between the last and first landmarks and between the first and last landmarks, respectively. Again, \(E\) and \(F\) are transposes of each other as 2 × 2 matrices. These sub-matrices also continue and fill up the remaining spots based on the covariances between individual landmarks.

From these sub-matrices, the total size of \(P\) can be calculated according Equation 3.4.

\[
size(P) = (3 + 2n)^2
\]  \hspace{1cm} (3.4)

Coincidentally, the dimensions of \(P\) are 3 + 2n by 3 + 2n. Thus, \(P\) has the same length as \(X\), but \(X\) is simply a vector, and \(P\) is a square matrix.

To use the EKF with SLAM, certain values must be initialized. Specifically, the \(X\) and \(P\) matrices must be populated for the EKF to work properly. In IH-SLAM, \(X\) is set up as a 3 × 1
matrix with all 0’s, and \( P \) is configured as a \( 3 \times 3 \) matrix with all 0’s. Initialization values can vary among implementations, but this organization should provide acceptable results and was chosen because of the robot’s unknown starting position.

### 3.2 IH-SLAM MATLAB Simulator

The IH-SLAM algorithm is simulated in the MATLAB environment. The IH-SLAM MATLAB simulator is built upon a framework laid out by Tim Bailey [27]. Tim Bailey’s simulator was intended for comparison of different map building algorithms and to be useful to the SLAM research community. The specific version that this research is based upon is version 2, which avoids copying overhead by using global variables. The simulator is very useful in that it is a straightforward implementation of EKF SLAM, provides a graphical representation of the system state and map, and allows test environments to be generated easily.

The first alteration made to the simulator is to generate a more accurate representation of landmark extraction. Additionally, sensor data is represented graphically to the user in order to observe the sensor data provided to the system in real-time.

With respect to the graphical user interface (GUI) generated by the original simulator, other alterations were made in order to more efficiently represent the system. To facilitate the new sensor data, a separate axis was added in addition to the original position axis: the sensor axis. This axis displays the current readings from the sensor(s) based on the sensor setup. In Figure 12, the sensor axis is positioned at the top of the GUI, with the y-axis representing distance and the x-axis representing angle from the robot’s center. The sign of the angle is actually opposite of the actual angle sign, in order to display the sensor on the correct side for the display. The position axis is the larger axis in the bottom portion of the GUI. Here, the y-axis
and x-axis represent the y-axis and x-axis of the global frame. The size of the axes is calculated based on the points drawn, so the figure may not be to scale. The blue stars represent landmarks and the red ellipses around the landmarks and the robot’s position (around [2, 0]) show the areas of uncertainty based on the covariance matrix. Also, the thin dotted red, green, and blue lines represent the true, uncorrected, and corrected robot positions, respectively.

The remaining changes to the MATLAB implementation are specifically related to the IH-SLAM implementation and optimizations presented in this thesis. The layout of the main simulator function is changed to run more efficiently and to facilitate collection of important
results for testing. Additionally, the sensor and association types are implemented as modules that are called at the beginning of the simulator function in order to customize a simulation more easily. A configuration file is developed to include additional information while removing other unused information. Finally, a function is developed to run multiple simulations, collect results, perform post processing, and return a formatted array of simulation results. This function will be used extensively in Chapters 5 and 6.
CHAPTER 4
LOCALIZATION AND PREDICTION

4.1 Localization and Motion Model

The first function in the IH-SLAM algorithm is the prediction function. The prediction function must specifically know how to model the robot’s movements based on the control data supplied. Before describing the prediction function, the specific locomotion model used in IH-SLAM is described.

There are many forms of locomotion that can be used to mobilize a robot. Specific locomotion approaches include walking, jumping, running, sliding, skating, swimming, flying, and rolling. The simplest and most popular locomotion mechanism in mobile robotics is the wheel, or the rolling approach. The wheel can achieve good efficiencies with a relatively simple mechanical implementation. Also, balance is typically easier to accomplish in wheeled implementations than in other locomotion methods, as static stability is possible using only two wheels. There are many types of wheel configurations that can be used for rolling robots, but IH-SLAM focuses on a robot that uses a differential drive. This basic setup uses two wheels on opposite sides of the center of the robot to drive the vehicle. In Figure 13, the grey circle represents the frame of the robot platform, with a small white hole representing the front of the vehicle. The thin black line represents the center line across the robot, with the black wheels on either side. Each wheel has its own independently controlled motor, so that the robot can go
straight, backwards, or various turning maneuvers by varying the speeds and direction of each wheel. An advantage to this wheel layout is that it allows the robot a 0° turning radius, which can be a beneficial tool for tight maneuvers in indoor applications.

![Differential drive robot.](image)

In the IH-SLAM implementation, odometry is used for dead reckoning. Specifically, a separate wheel encoder is placed next to each wheel in order to incrementally track the distance each wheel travels. Wheel encoders are sensors that monitor wheel rotation. At each discrete time step of the IH-SLAM algorithm, right and left wheel distances are calculated by the robot’s main controller based on the wheel rotation data supplied by the wheel encoders. The equations for calculating the linear ($d_{linear}$) and angular ($d_{angular}$) displacements needed for the prediction function are given in Equations 4.1 and 4.2,

$$d_{linear} = \frac{d_{rw} + d_{lw}}{2}$$  \hspace{1cm} (4.1)

$$d_{angular} = \frac{d_{rw} - d_{lw}}{d_{wheelbase}}$$  \hspace{1cm} (4.2)
where \( d_{rw} \) and \( d_{lw} \) represent the displacement (or distance travelled in a time period) by the right and left wheels, respectively, since the last time step. Also, \( d_{wheelbase} \) represents the distance between the wheels, measured directly from the center of the treads of each wheel.

### 4.2 Prediction Step

After the linear and angular displacements are calculated, the change in \( x_r, y_r, \) and \( \theta_r \) are calculated for the current time \( k \) using Equations 4.3-4.5, where \( \theta_{r_{k-1}} \) represents the previously calculated angle of the robot (at time \( k - 1 \)) stored at position (3, 1) in the \( X \) matrix.

\[
\theta_{r_k} = \theta_{r_{k-1}} + d_{\text{angular}} \quad (4.3)
\]
\[
\Delta x_r = d_{\text{linear}} \times \cos(\theta_{r_k}) \quad (4.4)
\]
\[
\Delta y_r = d_{\text{linear}} \times \sin(\theta_{r_k}) \quad (4.5)
\]

The results from Equations 4.3-4.5 provide the robot with an estimate of its change in position and are used in a model to predict the new state. The EKF uses matrices 4.6-4.8 to relate the data to the filter for each new time step. First, the matrix \( f \) is a \( 3 \times 1 \) matrix and represents the odometry function, also known as the control vector. For time \( k \), the matrix \( f \) is calculated using Equation 4.6.

\[
f = \begin{bmatrix} x_{r_{k-1}} + \Delta x_r \\ y_{r_{k-1}} + \Delta y_r \\ \theta_{r_k} \end{bmatrix} \quad (4.6)
\]

Next, the Jacobian of \( f \) is generated, resulting in a \( 3 \times 3 \) matrix called \( A \), as shown in Equation 4.7.

\[
A = \begin{bmatrix} 1 & 0 & -\Delta y_r \\ 0 & 1 & -\Delta x_r \\ 0 & 0 & 1 \end{bmatrix} \quad (4.7)
\]
Finally, the noise matrix is generated. There are many ways of generating this noise matrix, but in IH-SLAM, a value $c$ is used as a weight. This variable can be changed based on the environment and hardware to more accurately predict the amount of error in the odometry system. Based on the predefined variable $c$ and the change in time $\Delta t$, the noise matrix, $Q$, is a $3 \times 3$ symmetric matrix shown in Equation 4.8.

$$Q = \begin{bmatrix}
    c\Delta x_r^2 & c\Delta y_r \Delta x_r & c\Delta x_r \Delta t \\
    c\Delta y_r \Delta x_r & c\Delta y_r^2 & c\Delta y_r \Delta t \\
    c\Delta t \Delta x_r & c\Delta t \Delta y_r & c\Delta t^2
\end{bmatrix} \quad (4.8)$$

The final step in the prediction function uses the $f$, $A$, and $Q$ matrices to calculate new $X$ and $P$ matrices for time $k$. During these final calculations, the EKF uses known information about the system and the measured movement (odometry) to predict the robot’s current position. First, only the top three rows in the $X$ matrix are updated, as shown in Equation 4.9.

$$X_{1:3} = f \quad (4.9)$$

It is important to note that the positions of the landmarks are not changed during the prediction step. Next, the upper left $3 \times 3$ sub-matrix of $P$ is updated using Equation 4.10.

$$P_{1:3,1:3} = AP_{1:3,1:3}^T + Q \quad (4.10)$$

This update increases the uncertainty of the robot’s position based on the type of movement compared to the expected error. Finally, the covariances between the robot and landmarks are updated using Equation 4.11, where $end$ represents the entire width of $P$.

$$P_{1:3,4:end} = AP_{1:3,4:end} \quad (4.11)$$

Since $P$ is symmetric, a change must also be made using Equation 4.12.

$$P_{4:end,1:3} = P_{1:3,4:end}^T \quad (4.12)$$

The sub-matrix $P_{1:3,4:end}$ is of dimension $3 \times 2n$, where $n$ is the number of landmarks. This final calculation completes the prediction function.
CHAPTER 5

OBSERVATION AND LANDMARK EXTRACTION

5.1 Observation

Landmark extraction is an important function in any EKF-based SLAM implementation. The correct operation of the filter is entirely dependent on the regular and accurate extraction of observable landmarks. After the prediction function, the IH-SLAM algorithm typically waits on data from the observation sensors. Once this data is received, the landmark extraction function begins. Mathematically, this function is hard to describe since it does not directly operate on the matrices used by the EKF. However, the general process can be defined.

5.2 Sensor Overview

The driving force behind the landmark extraction function is the exteroceptive observation sensor. The observation sensor needs to provide accurate distance and bearing readings at regular intervals in order to provide adequate information for the filter to operate. The typical device used to accomplish this task is a range measurement device. There are numerous types of range measurement devices, including laser, infrared (IR), sonar, and stereo vision sensors. As previously stated, the more expensive laser scanners are typically used due to the amount of data they collect. However, their cost is prohibitive for smaller robotic applications.
This chapter presents the different characteristics of the standard types of distance sensors that could be used for landmark extraction. It also suggests a few extraction techniques that will allow the less expensive sensors to provide more information to the SLAM algorithm. Finally, the different distance sensors and extraction algorithms are simulated using MATLAB in randomly generated indoor environments, and the results are compared to determine the success of a low-cost sensor implementation.

**Laser Scanners**

Laser scanners typically scan many angles over a large field of view at quick intervals. They typically work using the time of flight (TOF) method. TOF works by sending out a signal that can bounce off of objects and measuring the time between signal emission and reflection. This time is used to determine distance to the object. Laser scanners are precise, and provide a wealth of good data for SLAM to use in calculations, but they are also very expensive. A comparison of laser scanners and their cost and statistics is shown in Table 5.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Price</th>
<th>Field of View</th>
<th>Range</th>
<th>Angular Resolution</th>
<th>Accuracy</th>
<th>Scan Time for Entire Field of View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hokuyo URG-04LX</td>
<td>$2,375</td>
<td>240°</td>
<td>2cm to 4m</td>
<td>0.36°</td>
<td>1cm up to 1m, 1% of distance to max</td>
<td>100ms</td>
</tr>
<tr>
<td>Hokuyo URG-04LX-UG01</td>
<td>$1,175</td>
<td>240°</td>
<td>6cm to 4m</td>
<td>0.36°</td>
<td>3cm up to 3m, 3% of distance to max</td>
<td>100ms</td>
</tr>
<tr>
<td>Hokuyo URG-08LX</td>
<td>$3,950</td>
<td>270°</td>
<td>10cm to 8m</td>
<td>0.36°</td>
<td>3cm up to 3m, 3% of distance to max</td>
<td>67ms</td>
</tr>
<tr>
<td>Hokuyo UTM-30LX/LN</td>
<td>$5,590</td>
<td>270°</td>
<td>10cm to 30m</td>
<td>0.25°</td>
<td>5cm up to 10m</td>
<td>25ms</td>
</tr>
<tr>
<td>Hokuyo UBG-04LX-F01</td>
<td>$2,850</td>
<td>240°</td>
<td>6cm to 4m</td>
<td>0.36°</td>
<td>1cm up to 1m, 1% of distance to max</td>
<td>28ms</td>
</tr>
<tr>
<td>SICK LMS100</td>
<td>$5,000</td>
<td>270°</td>
<td>50cm to 20m</td>
<td>0.25°</td>
<td>4cm total range [28]</td>
<td>20ms</td>
</tr>
<tr>
<td>RevoLDS</td>
<td>$30</td>
<td>360°</td>
<td>20cm to 6m</td>
<td>1.00°</td>
<td>3cm total range</td>
<td>100ms</td>
</tr>
</tbody>
</table>

The cost for these sensors is high because of the large field of view, quick scan time, small angular resolution, and the ability to detect multiple objects at different angles in the sensor range during each scan. Also, some of the more expensive laser sensors provide distances up to
30m. The small angular resolution provides a huge amount of digital range data and allows very precise position extraction of landmarks for the SLAM algorithm. Laser scanners have been used successfully in many SLAM implementations [29], [30], [31], [32].

The RevoLDS sensor is the anomaly of this group of laser sensors. This sensor was introduced by Kurt Konolige et al. and uses triangulation instead of TOF [33]. The sensor works in indoor conditions, measures a full 360° field of view, has an acceptable range of 6m, has a reasonable error and angular resolution, and scans at 10Hz. The main attractions of this sensor are its small size, low power (< 2W), and drastically reduced cost compared to the other laser scanners. The sensor is not currently in production, but if the sensor makes it to production it could make laser scanners affordable to low-cost projects without sacrificing sensor capabilities.

For IH-SLAM, the Hokuyo URG-04LX, Hokuyo URG-04LX-UG01, Hokuyo UTM-30LX/LN, and RevoLDS sensors are modeled and simulated for comparison. These sensors were chosen due to their differences in characteristics. The goal is to observe the differences in their ability to provide data to the SLAM algorithm. All four sensors are modeled with the documented distance range, scan rate, field of view, and angular resolution. The accuracy is modeled as uniformly distributed noise with the documented maximum errors.

**Sonar**

Another type of range finding sensor is a sonar sensor. Sonar works by measuring the TOF between emitted and reflected sound waves. Sonar has the advantage of being cheap compared to laser scanners, but the readings are not as accurate as laser scanner readings. This is mainly because the field of view of sonar sensors is much narrower than laser scanners, and while the distance to an object is returned, the specific bearing within the field of view is not known. Laser scanners can provide less than 1° of angular resolution to calculate bearing, while
sonar sensors may provide an angular resolution of 50° or more. This lack of angular resolution is the main problem with using sonar sensors over laser scanners.

Another problem with sonar is that the sound waves travel at the speed of sound, which is much slower than the speed of light. This slower wave speed translates to longer overall scan times, which can also cause problems if the robot is moving between signal emission and detection. In addition, sound waves can also be absorbed by some types of material; however, sound waves are unaffected by lighting and sunlight, which can affect laser and IR sensors. Also, sonar sensors can be negatively affected by false (or ghost) echoes, where a sound wave bounces off of multiple objects before finally returning to the sensor, thus leading to a wildly incorrect reading. Sound waves do travel at different speeds dependent on temperature, which could require the use of a temperature sensor in environments with widely varying temperatures. Finally, sonar sensors can require timing to be performed by the robot’s microcontroller, which can take up processing time needed for other tasks. Nonetheless, sonar is a potential low-cost option and has been used in [34], [35].

Some examples of possible sonar sensors are shown in Table 6. The Devantech and Maxbotix sensors are very comparable in range, field of view, scan time, and price.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Price</th>
<th>Field of View</th>
<th>Range</th>
<th>Angular Resolution</th>
<th>Accuracy</th>
<th>Scan Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Devantech SRF02</td>
<td>$25</td>
<td>45°</td>
<td>15cm to 6m</td>
<td>45°</td>
<td>4cm max</td>
<td>70ms</td>
</tr>
<tr>
<td>Devantech SRF04</td>
<td>$30</td>
<td>45°</td>
<td>3cm to 3m</td>
<td>45°</td>
<td>4cm max</td>
<td>36ms</td>
</tr>
<tr>
<td>Devantech SRF05</td>
<td>$30</td>
<td>45°</td>
<td>3cm to 4m</td>
<td>45°</td>
<td>4cm max</td>
<td>36ms</td>
</tr>
<tr>
<td>Devantech SRF08</td>
<td>$64</td>
<td>55°</td>
<td>3cm to 6m</td>
<td>55°</td>
<td>4cm max</td>
<td>65ms</td>
</tr>
<tr>
<td>Maxbotix MaxSonar-EZ0</td>
<td>$28</td>
<td>50°</td>
<td>15cm to 6.45m</td>
<td>50°</td>
<td>4cm max</td>
<td>50ms</td>
</tr>
<tr>
<td>Maxbotix MaxSonar-EZ1</td>
<td>$25</td>
<td>40°</td>
<td>15cm to 6.45m</td>
<td>40°</td>
<td>4cm max</td>
<td>50ms</td>
</tr>
<tr>
<td>Maxbotix MaxSonar-EZ4</td>
<td>$28</td>
<td>40°</td>
<td>15cm to 6.45m</td>
<td>40°</td>
<td>4cm max</td>
<td>50ms</td>
</tr>
</tbody>
</table>
Sonar sensors typically have cone shaped beams; some examples of the beam shapes are shown in Figures 14-16. While the beams are not perfectly cone shaped, they are modeled that way in the IH-SLAM simulation.

The Devantech SRF05 and Maxbotix MaxSonar-EZ1 are modeled and simulated in the IH-SLAM algorithm. The sensors are very comparable in characteristics, but the Devantech sensor scans at a faster rate. The sensors are modeled as arrays of 3, 5, and 7 sensors to provide a comparable scan area to a laser scanner. The sonar models generate noise using uniformly distributed random numbers.

**Infrared**

One option that does not suffer from some of the problems that sonar sensors experience is infrared (IR) sensors. IR sensors are cheap, and the readings have similar accuracy to sonar. However, the field of view is much smaller than sonar, which can be both problematic and beneficial. The smaller field of view allows for a more precise bearing to be extracted, which will generate a more accurate [x, y] landmark position than a sonar sensor can provide. However, this narrower beam does not allow for much of the environment to be observed and can prevent the landmark extraction algorithm from collecting enough data.

An advantage over sonar is that IR sensors, like laser scanners, operate in the infrared spectrum instead of the sound spectrum, and therefore are not affected by temperature or
material absorbance. However, IR sensors can be negatively affected by the reflectivity of an object and by sunlight. IR sensors do not have to account for the robot travelling between beam emission and reception since IR signals travel at the speed of light.

There are a variety of commercially available IR sensors, but one popular and reliable brand is Sharp. These sensors are under $20, provide good accuracy and scan time, and offer very narrow beams with decent detection ranges, although the detection ranges are noticeably less than sonar and drastically less than laser scanners.

Table 7. IR sensor comparison.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Price</th>
<th>Field of View</th>
<th>Range</th>
<th>Field of View Resolution</th>
<th>Accuracy</th>
<th>Scan Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sharp GP2D120</td>
<td>$12.50</td>
<td>1-2°</td>
<td>4cm to 0.3m</td>
<td>1-2°</td>
<td>unknown</td>
<td>50ms</td>
</tr>
<tr>
<td>Sharp GP2D12</td>
<td>$12.50</td>
<td>1-2°</td>
<td>10cm to 0.8m</td>
<td>1-2°</td>
<td>unknown</td>
<td>50ms</td>
</tr>
<tr>
<td>Sharp GP2Y0A21YK</td>
<td>$12.50</td>
<td>1-2°</td>
<td>10cm to 0.8m</td>
<td>1-2°</td>
<td>unknown</td>
<td>50ms</td>
</tr>
<tr>
<td>Sharp GP2Y0A02YK</td>
<td>$12.50</td>
<td>1-2°</td>
<td>20cm to 1.5m</td>
<td>1-2°</td>
<td>3-4cm</td>
<td>50ms</td>
</tr>
<tr>
<td>Sharp GP2Y0A700K</td>
<td>$19.50</td>
<td>1-2°</td>
<td>1m to 5.5m</td>
<td>1-2°</td>
<td>unknown</td>
<td>50ms</td>
</tr>
</tbody>
</table>

The five sensors listed in Table 7 are similar with the main difference being the minimum and maximum detection ranges, from 4cm to 550cm.

There are a few problems with the Sharp IR sensors that must be considered. First, if an object is closer than the minimum range, the sensor will provide bad readings. This problem can be corrected by using redundant sensors with smaller ranges. Another problem is that due to internal timing circuitry, there is periodic noise that occurs in the voltage output. This noise varies from sensor to sensor and will require an RC filter or software averaging to correct. The Sharp IR sensors do not provide an exact distance value, but instead provide an analog voltage that must be converted to a digital value representing the distance. The distance-voltage relationship is not linear though, so the sensor must be calibrated to determine the best fit polynomial to use.
The final issue that must be considered with the Sharp IR sensors is the beam shape and width. The beam width is very narrow and does not widen much over the full range. More importantly, an object needs to fill the entire beam area to provide an accurate reading. If an object fills only part of the sensor beam, then the sensor reading is unreliable. As the object fills more of the beam, the voltage output slopes linearly from the previous voltage towards the correct voltage corresponding to the actual object distance. The beam width, shape, and this beam edge problem are shown in Figure 17.

![Figure 17. Sharp GP2Y0A02YK IR sensor beam.](image)

The outer red areas represent the edges of the beam, while the inner blue area represents the area where the voltage stabilizes at the correct level. There are several possible solutions to eliminate this problem, such as filtering, scanning using a turret, and software averaging. Another possible solution is to use thresholds. If an object enters the side of the sensor beam, then the changes in distance will be drastic for a short period of time, while the distance adjusts to the new reading. Therefore, an alteration can be made to the IR sensor reading function. Consecutive sensor readings that have changed more than a certain threshold are considered part...
of this bad beam range and are discarded until the sensor distances have stabilized. This
threshold is dependent on the specifications of the IR sensor and on the angular speed of the
robot. Once the voltage has stabilized, the readings are considered valid.

In the IH-SLAM simulation, the GP2Y0A02YK sensor is the only sensor modeled and
simulated. This sensor was selected over the GP2Y0A700K sensor due to its lower cost and
smaller size. Also the GP2Y0A02YK sensor has a minimum range of 20cm compared to the 1m
minimum range of the GP2Y0A700K sensor. The other sensors were not chosen due to their
unacceptably low maximum ranges. The noise for this sensor is modeled using uniformly
distributed noise based on the maximum values accuracy specified in Table 7.

Vision

The final option that has been used successfully is vision. Vision can be appealing as it
replicates the way humans view the world. Traditionally, vision was very computationally
expensive and was prone to changes in natural and artificial light. However, advances within
this field have made vision a potential option for observation and landmark extraction. Vision
has been used successfully in [39].

Vision-based observation suffers from several problems making it inappropriate for low-
cost, small-scale robotics applications. First, vision systems are expensive to implement (in the
hundreds of dollars). Second, a stereo or triclops system must be implemented to measure
distance adding to hardware costs. Finally, manipulation of image data is computationally
intensive requiring significant computational resources. Due to these shortcomings, vision
sensors will not be used in IH-SLAM.
5.3 Landmark Extraction

In IH-SLAM, landmark extraction will provide a $2 \times o$ matrix to the data association function, where $o$ represents the number of extracted landmarks. This matrix, called $z$, is shown in Equation 5.1,

$$
 z = \begin{bmatrix}
 r_{z_1} & r_{z_2} & \cdots & r_{z_o} \\
 b_{z_1} & b_{z_2} & \cdots & b_{z_o}
\end{bmatrix}
$$  \hspace{1cm} (5.1)

where the variable $b_{z_i}$ represents the bearing of the $i$th extracted landmark. The bearing is the direction of the landmark relative to the robot’s local frame. Also, the variable $r_{z_i}$ represents the range of the $i$th extracted landmark, or the distance from the center of the robot to the landmark.

Another $2 \times o$ matrix which will provide useful information in data association is $z'$, shown in Equation 5.2.

$$
 z' = \begin{bmatrix}
 x_{z_1} & x_{z_2} & \cdots & x_{z_o} \\
 y_{z_1} & y_{z_2} & \cdots & y_{z_o}
\end{bmatrix}
$$  \hspace{1cm} (5.2)

This matrix is very similar to $z$, except the global x and y positions are stored for each landmark. These positions are calculated from the range and bearing information from $z$ combined with the information about the robot’s current state, stored in $X$. The individual calculations for the $i$th landmark $[x, y]$ position are given in Equations 5.3 and 5.4.

$$
 x_{z_i} = x_r + \left( r_{z_i} \cos(-b_{z_i} + \theta_r) \right)
$$  \hspace{1cm} (5.3)

$$
 y_{z_i} = y_r + \left( r_{z_i} \sin(-b_{z_i} + \theta_r) \right)
$$  \hspace{1cm} (5.4)

The actual calculation of the $z$ matrix is dependent on the type of landmark extraction algorithm used and on the types of sensors used. In IH-SLAM, a variety of different sensors are used, ranging from complex laser scanners to simple infrared (IR) rangefinders. Therefore, the specific manner for calculating the range and bearing is presented in subsequent sections, where the specific landmark extraction algorithms are discussed.
5.4 Laser Extraction

Laser scanners provide a large amount of data in the form of an array of distances corresponding to specific bearings. Typical data collected by a laser scanner is represented in Figure 18. Laser scanners in IH-SLAM extract landmark info from this array of data using a variation of the SPIKES method [40].

![Figure 18. Typical laser scanner data.](image)

SPIKES is the concept of finding distance spikes in an array of data. This method is commonly used with laser scanners, since these sensors provide large arrays of data with a high angular resolution. To find these spikes, the algorithm iterates through the entire array, and for each value, it checks the values to the left and right for comparison. If the middle distance value is smaller than the right and left values by a specific threshold, then it is extracted as an observable landmark. SPIKES is a simple algorithm and is very computationally efficient. However, SPIKES will fail in various naïve situations. For instance, as a small object gets closer to the sensor, it may take up 2 or more pixel values in the sensor array. Using this naïve version of SPIKES, this landmark will no longer be detected. There is a specific modification to the naïve algorithm that will allow SPIKES to more robustly discover landmarks.
This modification allows the algorithm to discover edges in addition to spikes. Whenever there is a change in distance of a certain threshold between two adjacent pixel values in the sensor array, this will be considered the edge of some object. This provides a much better picture of the environment since most of these edges can be observed by the robot from different perspectives around the map.

Other landmark extraction methods for laser scanners include Random Sample Consensus (RANSAC) and geometric polygon extraction. These methods are much more complicated than SPIKES, so they will not be implemented in IH-SLAM. A full description of RANSAC is presented in [41], while an example of extraction using polygons is presented in [42].

There are two aspects to simulating a laser scanner in the IH-SLAM simulator. First, the laser scanner data must be generated based on the robot’s position and the positions of the landmarks around it. This data must be generated according to the specifications of the specific laser scanner for each particular test. Also, noise must be added to the data according to each particular sensor’s specifications. Second, this sensor data must be interpreted by a landmark extraction function that is specifically designed for a laser scanner. The extracted landmarks need to be provided in [range, bearing] and [x, y] formats. Since spikes (or edges) are the only type of information being extracted, the landmark heading, θ, does not need to be calculated or stored.

For the laser scanner generator function, each landmark is checked to determine if it is in the field of view of the sensor. If so, the particular position in the laser array is calculated by rounding upwards towards the closest value. The bearing for the corresponding landmark is calculated using Equation 5.5.
\[ b = a \left( n - \text{ceiling}(\frac{w}{2}) \right) \frac{\pi}{180} \]  

(5.5)

Equation 5.5 assumes that the sensor array will provide values starting from the left side of the field of view \((-\frac{w}{2})\) and continuing to the far right side of the field of view \(\frac{w}{2}\). The variable \(b\) represents the bearing, \(n\) represents the position of the extracted landmark in the array, \(a\) represents the angular resolution for each array value, and \(w\) represents the total field of view. Then, the distance is calculated using the distance from the center of the robot (where the laser scanner is placed) to the landmarks and adding uniformly distributed noise that ranges from \(-R \leq \text{noise} \leq R\), where \(R\) is the maximum error given in the laser scanner specification.

For the landmark extraction function, a threshold of 20cm is used to only detect edges or spikes. Any distance change on either the right or left greater than the threshold will be extracted as a landmark. In a real 2D environment, the distance needs to change by this threshold on one side, but change by less than a minimum threshold on the other side. This prevents the algorithm from extracting walls that are close to the sensor and provides large distance changes between readings. This implementation only extracts 1D points, so the simple SPIKES algorithm will suffice.

The final calculation needed for the landmark extractor function is to convert the [range, bearing] information to an \([x, y]\) position. This calculation requires a frame transformation using Equations 5.6 and 5.7.

\[ x_l = x_r + r \times \cos(\theta_r - b) \]  

(5.6)

\[ y_l = y_r + r \times \sin(\theta_r - b) \]  

(5.7)

From these two equations, \(x_l\) and \(y_l\) represent the position of the landmark, while \(x_r\), \(y_r\), and \(\theta_r\) represent the estimated position of the robot in its current state. The variable \(r\) is the distance reading from the sensor, and the variable \(b\) is the bearing which is calculated in Equation 5.5.
5.5 IR Extraction

To correctly simulate IR sensors in IH-SLAM, a number of assumptions must be made. First, the problem with periodic sensor noise discussed in Section 5.2 is not modeled, since this problem has been solved successfully using RC and software filtering techniques in [43]. Second, errors from analog-to-digital conversion (ADC) are considered negligible, since a 10-bit ADC will only cause a maximum of 3mV rounding error which corresponds to 0.6% of the range. Third, the errors caused by an object that does not fill the entire sensor field of view are not modeled. It is assumed that these errors are corrected using some form of hardware or software filtering technique or by only using distance values that have stabilized.

The very narrow field of view of the IR sensor is ideal for extracting a precise bearing to the object; however, a single sensor can only sense a tiny fraction of the environment. This problem would most certainly lead to insufficient data collection, and the SLAM algorithm would fail very quickly. There are two potential solutions to solve this problem.

First, more than one sensor can be used. The sensors can be set up in varying arrangements pointing in different directions to cover a greater area. This solution is simple, yet requires more money to cover the cost of the additional sensors. It is also still limited in its view of the environment due to the narrow beam width. Extraction for this solution is nearly identical to SPIKES, except instead of using adjacent array values, the extraction algorithm uses consecutive IR readings from the same sensor. The IR extraction algorithm is implemented similarly to the laser extraction algorithm, with a threshold of 20cm. The closer of the current and previous readings will be extracted as the landmark. To accomplish this, each sensor needs to store its previous distance readings.
The second solution is to use one or more sensors on a turret that rotates to scan the environment. This solution is able to view more of the environment, but requires extra cost to purchase a motor to rotate the turret. It also causes areas of the environment to be unobserved for long periods of time while the turret is pointed in one direction. Some motors are also not very accurate in position, so some sort of optical device (such as a wheel encoder) is needed to determine the exact direction that the sensor(s) are pointing for each scan. This added hardware adds more cost to the system. Therefore, due to these complications, this solution is not modeled in IH-SLAM. Instead, multiple sensors are implemented.

The first step to correctly model the IR sensor data is to determine the number of sensors and their arrangement. In IH-SLAM, three different arrangements are used. These three arrangements are tested in the different environments in order to determine their accuracy and feasibility. The three arrangements are described in Table 8. The angles are relative to the front of the robot.

<table>
<thead>
<tr>
<th>Number of Sensors</th>
<th>Angles</th>
<th>Total Field of View</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-20°, 0°, +20°</td>
<td>around 3°</td>
</tr>
<tr>
<td>5</td>
<td>-40°, -20°, 0°, +20°, +40°</td>
<td>around 5°</td>
</tr>
<tr>
<td>7</td>
<td>-60°, -40°, -20°, 0°, +20°, +40°, +60°</td>
<td>around 7°</td>
</tr>
</tbody>
</table>

The second step in modeling the IR sensor observation system is to model the actual sensor data. To do this, each actual landmark is checked to determine if it is in the field of view of any of the IR sensors. If so, then the distance is calculated from the center of the robot. However, it is assumed that sensors are placed directly on the outside of the circular robot frame, so \( \frac{1}{2} \) of the wheelbase is subtracted from the distance reading. Finally, uniformly distributed noise is added to the sensor readings.
The final step is to create the extraction function. The extraction method used is based on the SPIKES extraction method. The function uses the current sensor data, current robot state, previous sensor data, and previous robot state to produce a [range, bearing] and [x, y] position. The calculations for [range, bearing] and [x, y] are shown in Section 5.3. The only difference in these calculations is the range calculation. The formula for calculating range is no longer simply the distance reading but is now calculated using Equation 5.8,

\[ r = d + \frac{d_{\text{wheelbase}}}{2} \]  

(5.8)

where \( r \) is range, \( d \) is the measured distance, and \( d_{\text{wheelbase}} \) is the distance between the wheels of the robot.

5.6 Sonar Extraction

Sonar sensors have a much wider field of view and angular resolution than IR sensors, but this presents a problem in discerning the precise bearing from a sensor reading. For example, for a sonar sensor having a 45° field of view (and therefore, angular resolution), at the maximum range of the sonar sensors modeled (around 6m), the position error from the middle of the sensor beam would have an upper bound of 2.3m. This amount of uncertainty is unacceptable for the accuracy needed by the extraction algorithm. Another problem, which is also encountered with IR sensors, is the risk of losing data from the environment. This is due to the possibility that numerous landmarks can go unnoticed if there is a closer landmark, since sonar sensors only return the distance of the nearest object detected.

The first step in modeling a sonar-based extraction and observation system involves the number and arrangement of sonar sensors. While a 45° field of view is far greater than the field of view of IR sensors, it is still significantly less than the field of view of laser scanners. Also,
only one object is detectable in this range, greatly limiting the ability of the sensors. Therefore, different combinations of multiple sensors are tested. Since sonar beams are much wider than IR beams and there is a potential for detection of false echoes, a solution to prevent these problems must be found.

One such solution is to set up each sensor well outside of the field of view of the other sensors. This can prevent some false echo problems, but there is still the problem of crosstalk between the sensors. Another solution is to operate the sonar sensors at different intervals to allow the other sensors’ echoes time to dissipate. This solution solves the problem of crosstalk, but wastes a large amount of time that could be used for data collection. A final solution is to operate each sonar sensor at a different sound frequency. This solution can also be implemented by using coded signals and sequences, as implemented in [44] and [45]. However, coded signals add complexity, and it is not easy to change the frequency of common commercial sonar sensors. IH-SLAM assumes that these problems are solved using coded signals, so the sensors are able to operate at their defined frequencies.

In IH-SLAM, three different sensor configurations are chosen. These configurations are given in Table 9 for two different sensors. Again, the angles are relative to the front of the robot. The total fields of view for the SRF05-7 and EZ1-7 sensor configurations are very comparable to that of a laser scanner.

Table 9. Sonar sensor layouts.

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Number of Sensors</th>
<th>Angles</th>
<th>Total Field of View</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRF05</td>
<td>3</td>
<td>-45°, 0°, +45°</td>
<td>135°</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>-90°, -5°, 0°, +45°, +90°</td>
<td>225°</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>-135°, -90°, -5°, 0°, +45°, +90°, +135°</td>
<td>315°</td>
</tr>
<tr>
<td>EZ1</td>
<td>3</td>
<td>-40°, 0°, +40°</td>
<td>120°</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>-80°, -40°, 0°, +40°, +80°</td>
<td>200°</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>-120°, -80°, -40°, 0°, +40°, +80°, +120°</td>
<td>280°</td>
</tr>
</tbody>
</table>
Once the sensors are configured, the sensor data is simulated based on the environment and the robot position. For each reading, individual landmarks are inspected to determine if they lie in the sonar beams. The shapes of the sonar beams for the two modeled sensors are very similar, both being nearly cone shaped. For IH-SLAM, it is assumed that the sensor beams are a perfect cone shape. Similar to the IR sensor data model, the distance is calculated from the center of the robot to the landmarks. Then, ½ of the wheelbase is subtracted to calculate the actual measured distance from the sensor, since the sensors are placed on the outside of the circular platform. Finally, uniformly distributed random noise is added based on each sensor’s specifications.

From the modeled sonar data, landmark extraction is performed. The sonar extraction algorithm is significantly different than the previously described laser and IR extraction algorithms, mostly due to the large uncertainty in the bearing to the landmark. Some techniques for extracting landmarks from sonar sensors include the Hough transform [46] and triangulation [44]. The triangulation extraction method was studied extensively, but due to the small baseline between the sonar sensors, it did not perform effectively. Therefore, a new algorithm is presented, called beam edge detection (BED) landmark extraction.

If the robot is heading straight towards a landmark, the landmark will appear as a steadily decreasing distance over time from just outside of the acceptable sensor distance range. However, any landmark that enters the sensor range from the side of the beam will typically provide a drastic change in distance from one reading to another. Based on this concept, the algorithm proposes the following: If the difference between the current sensor reading \(d_c\) and the previous sensor reading \(d_p\) for a single sensor is greater than some threshold \(d_t\), then the landmark has appeared from the side (or edge) of the sensor beam.
Knowing that a landmark has entered the sensor beam from the side is important for one main reason: it solves the problem of the uncertain landmark bearing without sacrificing the ability to continuously observe landmarks or the ability to view a large chunk of the environment. It is important to note that in this implementation, the sensor beam is assumed to be cone-shaped, which simplifies the math. In reality, sensor beams are not perfectly cone-shaped and can vary from one sensor to another, so exact measurements are needed to implement this method using real hardware. However, a major problem is encountered: with the basic algorithm, the robot will not know which side of the sensor beam the landmark entered.

There are several different methods that can be used to determine which side of the beam is correct, and there are other methods that can give even more information. Each of these methods can be used independently or can be combined together in a single algorithm. The following four methods are presented:

1. Based on observing a similar distance in one of the neighboring sensors recently, if an object appears that is more than \(d_t\) of the minimum range of the sensor and closer than the previous reading by less than \(d_{t_1}\), it just appeared in the edge of the beam adjacent to the neighboring sensor.

2. Based on the linear and angular velocities, the edge of the beam is calculated.

3. If the sensor observes a landmark that was previously detected on the edge of the same sensor beam, then the previously calculated \([x, y]\) position of the landmark is used to calculate a new bearing; then the new distance is used to calculate a new \([x, y]\) position.

4. The sensor can observe when an object leaves the sensor as well (when distance increases by a certain threshold, \(d_{t_3}\)), covering both edges of the sensor beam.
The first method can only be used for multiple sensor configurations, and the entire algorithm is based on consecutive sensor scans using the different sensors. Since the sensor beams are aligned where the edges of the beams are nearly flush, and potentially overlapping, there is a decent chance that within a few time steps, the landmark will disappear from one sensor beam, only to appear in the next, or appear in two neighboring sensors during the same time step. This is detected by using a slightly more demanding threshold, \( d_{t_1} \). In this case, as long as the two readings from neighboring sensors are within \( d_{t_1} \) of each other, it is likely that the landmark has moved from one beam to the other within the time step. Therefore, the far angle of the sensor beam that most recently observed the landmark is used as the bearing in order to calculate the landmark \([x, y]\) position.

The second method expands on the first. Since the robot knows its own movement in the environment, it can determine which side of the sensor beam the landmark has likely entered. Many important movement factors come into play here. First, if the sensor is pointing towards the left or right of center from the robot’s front and the robot is moving directly forward with a very small angular velocity, then if a landmark enters the sensor, it must have entered on the edge of the beam closest to the center of the robot. This first bit of knowledge allows the robot to effectively track landmarks as it passes by them if the robot is travelling straight. Second, if the robot begins to turn, then the sensors opposite the side that the robot is turning can know that if a landmark enters the sensor beam, it has entered from the edge of the beam closest to robot center again. These two simple bits of knowledge provide a wealth of information to the robot. Also, based on the robot’s distance to an object and the linear and angular velocities of the robot, a specific mathematical equation can be derived to determine the likely side of the beam that the landmark entered. However, this solution is problematic, due to the common errors in
odometry. These odometry errors can cause the landmark side to be falsely detected in extreme cases. However, it may also allow for more landmarks to be observed. Due to computational complexity, this solution is not implemented and remains a candidate for future testing.

The third method occurs after either of the first two methods. If a sensor detects a landmark entering its beam, then it will likely gather distance readings to the landmark for many consecutive time steps. This data is effectively lost, yet it contains potentially useful information. This method strives to make use of this data by assuming the original detected landmark position \([x, y]\) after the [range, bearing] conversion. This original \([x, y]\) position is used to calculate the current bearing to the landmark. Once this bearing is calculated, the new distance reading is used to generate a new \([x, y]\). The problem with this method is that the \([x, y]\) is only partially observed, yet the algorithm considers this reading to be of equal importance as a full range and bearing observation, as in an edge detection. This problem may prove to be insignificant, but if it does appear to be problematic, future research could study modifications to the update state (specifically the Kalman gain) that could be made to alter the confidence in these less confident readings.

The final method is a simple extension from the first two methods. While landmarks can be detected when they enter the beam, landmarks can also be detected when they leave the beam. Using identical methods (but reverse) provides roughly twice as many readings, so this final method is very useful. To implement this method, a new threshold, \(d_t_3\), is introduced.

In IH-SLAM, the BED landmark extraction algorithm is implemented with methods 1, 2, and 4, specifically. From initial testing, method 3 actually causes the algorithm to perform worse than methods 1, 2, and 4 alone. Since other extraction algorithms have already been studied and triangulation was proven ineffective for this implementation, these other individual algorithms
are not modeled for this analysis. In the future, these other algorithms and others not discussed here could be added to provide a more comprehensive study.

5.7 Experiments

The purpose of modeling the different sensors and extraction methods in this IH-SLAM MATLAB implementation is to compare the accuracy of the laser, sonar, and IR sensor configurations. Also, the accuracy of each sensor configuration compared to the cost of that configuration is presented. To effectively compare the different sensors’ abilities, four different environments are used with randomly generated robot paths and random numbers and positions of landmarks to represent a number of possible environmental conditions.

These four unique environments are presented in Figures 19-22, where the blue star-shaped points represent landmark points, and the red line represents the robot path. The robot begins at point \((0, 0)\) for each experiment, and the ending position is marked by a small black triangle. The x and y axes are represented in meters. To maintain consistency for the experiments, each sensor configuration is subjected to the same set of experimental conditions and the true robot path in each environment is kept identical through each sensor test. While the estimated robot path is based on randomly generated noise, the robot always has the same opportunity to view landmarks.
Figure 19. Test 1 environment.

Figure 20. Test 2 environment.
Figure 21. Test 3 environment.

Figure 22. Test 4 environment.
For each sensor and environment combination, 100 test trials are presented. For each time step in a trial, the corrected and uncorrected errors are calculated using Equations 5.9 and 5.10, respectively.

\[
error_{\text{corrected}} = \text{abs}(position_{\text{corrected}} - position_{\text{true}})
\]  
\[
error_{\text{uncorrected}} = \text{abs}(position_{\text{uncorrected}} - position_{\text{true}})
\]

At the end of a trial, the average, maximum, and final position errors are stored for both the corrected and uncorrected errors. Finally, at the end of an experiment, the overall averages of the average, maximum, and final errors are calculated.

The experimental results are shown graphically in Figures 23-26. In these four figures, the sensor model is listed on the x-axis with three separate colored bars used to represent the average, maximum, and final improvement values. The percentage values in the figures represent the percentage of improvement from the uncorrected error to the corrected error. Equation 5.11 is used to calculate this improvement value.

\[
improvement = \frac{(error_{\text{uncorrected}} - error_{\text{corrected}})}{error_{\text{uncorrected}}}
\]

This percentage quantifies the effectiveness of the algorithm for a given sensor. Additionally, the sensor configurations are provided again for convenience in Table 10.

<table>
<thead>
<tr>
<th>Sensor Type</th>
<th>Configuration Name</th>
<th>Number of Sensors</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laser</td>
<td>Hokuyo URG-04LX</td>
<td>1</td>
<td>$2,375</td>
</tr>
<tr>
<td></td>
<td>Hokuyo URG-04LX-UG01</td>
<td>1</td>
<td>$1,175</td>
</tr>
<tr>
<td></td>
<td>UTM-30LX</td>
<td>1</td>
<td>$5,590</td>
</tr>
<tr>
<td></td>
<td>RevoLDS</td>
<td>1</td>
<td>$30</td>
</tr>
<tr>
<td>Sonar</td>
<td>SRF05-3</td>
<td>3</td>
<td>$90</td>
</tr>
<tr>
<td></td>
<td>SRF05-5</td>
<td>5</td>
<td>$150</td>
</tr>
<tr>
<td></td>
<td>SRF05-7</td>
<td>7</td>
<td>$210</td>
</tr>
<tr>
<td></td>
<td>EZ1-3</td>
<td>3</td>
<td>$75</td>
</tr>
<tr>
<td></td>
<td>EZ1-5</td>
<td>5</td>
<td>$125</td>
</tr>
<tr>
<td></td>
<td>EZ1-7</td>
<td>7</td>
<td>$175</td>
</tr>
<tr>
<td>IR</td>
<td>IR-3</td>
<td>3</td>
<td>$38</td>
</tr>
<tr>
<td></td>
<td>IR-5</td>
<td>5</td>
<td>$63</td>
</tr>
<tr>
<td></td>
<td>IR-7</td>
<td>7</td>
<td>$88</td>
</tr>
</tbody>
</table>
Figure 23. Landmark extraction results from Environment 1.

Figure 24. Landmark extraction results from Environment 2.
Figure 25. Landmark extraction results from Environment 3.

Figure 26. Landmark extraction results from Environment 4.
Additionally, Figure 27 shows the average of the final, maximum, and average results from all four test environments, with individual contributions displayed by color.

![Figure 27. Landmark extraction results averaging the four environments.](image)

Based on results from the first environment in Figure 23, it is apparent that the laser scanners (specifically the URG-04LX, URG-04LX-UG01, and RevoLDS sensors) are very accurate. However, it is interesting to note that all six sonar sensor combinations provided nearly equal results to the laser scanners. The UTM-30LXLN and IR-5 sensors had sub-par results while the IR-3 sensor actually performed worse than the uncorrected position.

Observing the second environment in Figure 24, the URG-04LX, URG-04LX-UG01, UTM-30LXLN, SRF05-5, and SRF05-7 sensors provided the best improvement. The RevoLDS, SRF05-3, EZ1-5, and EZ-7 sensors provided moderate improvement, while the EZ1-3 and IR sensors provided poor results. A pattern appears to be forming from the first two data sets in which the laser scanners and the SRF05-5 and SRF05-7 sonar combinations performed very well, and the IR sensors provided sub-par results.
From the third environment in Figure 25, the laser scanners again show good performance, but the three SRF05 sensor configurations are very comparable. In fact, SRF05-5 and SRF05-7 have better improvement values than the laser scanners. This is a very interesting fact considering the difference in cost between the two types of sensors. The EZ1 sensors showed increasing performance as the number of sensors increased, but the IR sensors showed a decreasing performance as the number of sensors increased, which is an unexpected result. The EZ1 and IR sensor combinations gave moderate improvement results.

In the final environment, shown in Figure 26, the laser scanners actually provide only a moderate improvement, possibly due to the sparse amount of landmarks. The SRF05, EZ1, and IR sensor combinations showed improvement as the number of sensors increased. The two sonar sensor models had nearly identical performance values between their three sensor combinations, while the IR sensors lagged significantly behind again. The IR-5 and IR-7 combinations provided little improvement while the IR-3 again provided worse performance than the uncorrected errors.

Figure 27 provides an overview of the different sensors and their performance. Again, the four laser scanners provided high improvement ratings, between 70% and 80%. However, the SRF05-5 and SRF05-7 sonar sensor configurations actually provided better performance than the laser scanners. The probable cause for this result is that since laser scanners provide more information, they actually tend to cause the EKF to overcorrect the position error. The BED landmark extraction algorithm used by the sonar sensors does not gain as much information as the SPIKES algorithm used by the laser scanners, but it provides enough accurate data to allow the EKF SLAM algorithm to work very efficiently at correcting odometry errors. The EZ1-7 sonar sensor provides nearly identical performance to the laser scanners, but in comparison to the
SRF05 sensor combinations, the EZ1 sensor combinations are slightly less accurate. This is likely due to a combination of the larger minimum range of the EZ1 sensor and the slightly smaller field of view. The IR sensors do not provide comparable improvement to the laser scanners or the sonar sensors, even with the IR-7 configuration.

The final graph, shown in Figure 28, provides a representation of the percentage of improvement provided by each sensor type compared to the monetary cost of the implementation. It is obvious that the three expensive laser scanners provide very low improvement per dollar cost due to their high cost. It is also apparent that the RevoLDS sensor, if it becomes commercially available for the stated $30, provides the best improvement per cost ratio of all sensor types. From the remaining sensor combinations, the SRF05-3 and SRF05-5 provide the best improvement per cost ratios. Additionally, the IR-3 and IR-5 sensors do not provide very good improvement per cost ratios due to their low improvement ratings.

Figure 28. Percentage of improvement per dollar cost for each sensor combination.
Based on experimental results, it is apparent that the RevoLDS, SRF05, and EZ1 sensor models provide similar performance improvement as the more expensive, commercially available laser scanners, but these sensor configurations are significantly less expensive. While these results are promising, it is important to note that landmark extraction in this simulation is accomplished using 1D points, not 2D surfaces. Therefore, it is necessary to extrapolate the 1D results to 2D space. For the 2D problem, observation and landmark extraction for laser and IR sensors will remain nearly identical based upon the SPIKES extraction algorithm. However, sonar observation is accomplished differently in the 2D environment. Only one side of each sensor beam is able to accurately detect a surface corner, depending on the angle of the corner seen by the robot. This should not affect the amount of data provided by the sonar sensors, since adjacent sensor beams will pick up the corner. Using BED, two adjacent sensors are capable of detecting the same landmark on the edges of both of their beams. However, this advantage is erased since a landmark extracted within the same time step by two different sensors is only extracted once to avoid adding two copies of the same landmark to the map. Therefore, the BED algorithm will not suffer when landmarks are changed from 1D points to 2D surfaces.

For the remainder of the testing of the IH-SLAM algorithm, only the RevoLDS, SRF05, and EZ1 sensor combinations are used. Since simulation results proved that these sensor configurations provide comparable results to commercially available laser scanners at a significantly lower cost, the remainder of this thesis examines other ways to reduce costs without sacrificing accuracy while using these sensor types.
6.1 Motivation

Initial timing tests of the five functions of IH-SLAM are presented in Figure 29. From the figure, it is evident that the extract and predict functions have very little impact on the execution time of the IH-SLAM algorithm, especially when the number of landmarks is large.

Figure 29. Timing of the five steps in the IH-SLAM MATLAB implementation.
However, while the update and augment (map insertion) functions do contribute to the processing requirements of the algorithm, the data association function is the largest timing constraint for IH-SLAM.

6.2 Nearest-Neighbor Approach

While landmark extraction is heavily dependent on the type of sensors used, data association is not. Therefore, the same data association algorithm can be used interchangeably with any landmark extraction algorithm, provided the given map remains consistent. In IH-SLAM, a feature map is used. While some feature maps use geometric shapes or lines as features, this implementation only uses 1D points in the 2D plane.

The nearest-neighbor algorithm is a method for comparing objects (landmarks) based on the closest distances in the feature space (2D plane). Specifically, the data association algorithm compares an extracted landmark’s [x, y] position with the positions of all other landmarks stored in the map. The stored landmark that is the shortest distance from the extracted landmark is selected, and the associated pair of landmark positions are sent to the next part of the data association algorithm.

Not all extracted landmarks can be associated directly with the closest stored landmark. Therefore, a validation gate is used to determine if the associations are actually valid. The addition of this validation gate changes the data association algorithm to the gated nearest-neighbor algorithm. The calculation of this gate is typically based on a minimum and maximum gate. In the minimum/maximum gate method, the features are matched if the distance is less than the minimum threshold. However, the observed landmark is only passed to the map insertion function if the distance is greater than the maximum threshold. The gated nearest-
neighbor approach is the basis for the data association algorithms presented in this thesis, with the minimum and maximum threshold referred to as the rejection and augmentation gate, respectively.

The calculation of the distance in the nearest-neighbor algorithm can also vary. Two specific distances that are commonly used are Euclidean and Mahalanobis distances. A comparison and specific description of both distances can be found in subsequent sections of this chapter.

6.3 Nearest-Neighbor Using Mahalanobis Distance

The Mahalanobis distance is a method for measuring distance, originally introduced by P.C. Mahalanobis in 1936 [47]. The distance is calculated based on correlations between variables involved with the distance. The correlations are used to identify patterns and determine the resemblance of an unknown set of variables to a known set. The calculation of Mahalanobis distance is different from the calculation of Euclidean distance, because it accounts for correlations between different data sets and is independent of the scale of the measurements.

For a random vector \( \mathbf{a} = (a_1, a_2, \ldots, a_n)^T \) with a mean \( \mu = (\mu_1, \mu_2, \ldots, \mu_n)^T \) and covariance matrix \( \mathbf{P} \), the Mahalanobis distance is mathematically defined in Equation 6.1.

\[
d = \sqrt{(a - \mu)^T \mathbf{P}^{-1} (a - \mu)}
\]  

(6.1)

The mean vector, \( \mu \), can also be another random vector with the same probability distribution as \( \mathbf{P} \). If the covariance matrix is the identity matrix, then the Mahalanobis distance simplifies to the Euclidean distance. For a diagonal (but not identity) covariance matrix, the distance is called the normalized Euclidean distance.
There are benefits and disadvantages to using the Mahalanobis distance to calculate the distance for nearest-neighbor. The main benefit is illustrated with Figure 30 and Figure 31.

![Figure 30. Standard Euclidean distance calculation.](image1)

![Figure 31. Mahalanobis distance calculation.](image2)

Using the standard Euclidean distance calculation, it would appear that landmark L₁ is the nearest neighbor to the observed landmark L₀. However, by using the Mahalanobis distance, the nearest neighbor is shown to actually be L₂. Therefore, nearest-neighbor techniques using Mahalanobis distance are much more reliable for data association when the covariance between variables is known.

The data association algorithm presented in this section uses the gated nearest-neighbor approach. These distance and gate calculations are applied to each extracted landmark, and each calculation compares all landmarks stored in the map. The stored landmark positions are found in the X matrix, as described earlier.

First, the algorithm will try to predict the landmark’s [range, bearing] reading using the current estimated robot position \((x_r, y_r, \theta_r)\) and the jth saved landmark position \((x_j, y_j)\).

\[
\begin{bmatrix}
  r_{pj} \\
  b_{pj}
\end{bmatrix} = \begin{bmatrix}
  \sqrt{(x_j - x_r)^2 + (y_j - y_r)^2} \\
  \tan^{-1}\left(\frac{y_j - y_r}{x_j - x_r}\right) - \theta_r
\end{bmatrix}
\]  

(6.2)

This 2 × 1 matrix \(z_p\) is calculated for all extracted landmarks and their corresponding associated landmarks. Next, the algorithm calculates the Jacobian of the measurement model with respect
to the robot’s position. This matrix has dimension $2 \times m$, with $m$ representing the length of the $X$ matrix. The matrix, $H$, is calculated using Equations 6.3 and 6.4.

$$
H_{1:2,1:3} = \begin{bmatrix}
\frac{x_r-x_j}{r_p j} & \frac{y_r-y_j}{r_p j} & 0 \\
\frac{y_j-y_r}{r_p j^2} & \frac{x_r-x_j}{r_p j^2} & -1
\end{bmatrix}
$$

(6.3)

$$
H_{(1:2),(2j+2:2j+3)} = \begin{bmatrix}
\frac{x_j-x_r}{r_p j} & \frac{y_j-y_r}{r_p j} \\
\frac{y_r-y_j}{r_p j} & \frac{x_j-x_r}{r_p j}
\end{bmatrix}
$$

(6.4)

The remaining columns in $H$ are filled with 0’s.

After $z_{pj}$ and $H$ are calculated, the next step is to calculate the innovation matrix, $S$. This is the key to calculating the Mahalanobis distance. First, the difference between predicted and measured landmarks is calculated using Equation 6.5.

$$
v = z - z_p
$$

(6.5)

The innovation matrix $S$ is then calculated using Equation 6.6.

$$
S = HP H^T + R
$$

(6.6)

Since $H$ is sparse, this calculation can technically be optimized in order to avoid unnecessary matrix multiplication steps. All of the unused matrix spaces (filled with 0’s) are not included in the optimized calculation. The resulting matrix, $S$, is a $2 \times 2$ matrix. The matrix $R$ in the equation is the measurement noise covariance. It is also a $2 \times 2$ matrix and is a predefined diagonal matrix used to estimate the amount of noise in the range and bearing readings from the observation sensors. Finally, two variables are calculated for direct use in the gated nearest-neighbor algorithm. These variables, $n_{is}$ and $n_d$, are calculated using Equations 6.7 and 6.8.

$$
n_{is} = v^T S^{-1} v
$$

(6.7)

$$
n_d = n_{is} + \log(\det(S))
$$

(6.8)
The calculations in Equations 6.7 and 6.8 are performed for all extracted/associated landmark pairs. These values represent the Mahalanobis distance between the two landmarks. Each $n_{is}$ value is passed through the rejection gate. If the $n_{is}$ value is less than the rejection gate, then the corresponding landmark pair is matched as a valid association. If multiple $n_{is}$ values pass for a specific landmark, then the value with the smallest distance, $n_{d}$, is used for the association.

Associated landmark pairs are added to a matrix called $z_f$. This matrix is $2 \times a$, where $a$ represents the number of associations. The first and second rows of each column store the measured [range, bearing] of the jth observation, $(r_j, b_j)$, respectively. Also, a matrix $id_f$ is used to store the id of the matched landmark from the map. The id is simply the landmark number, calculated from the position of the landmark in the X matrix.

In addition to the associated landmark matrix $z_f$, data association also returns information regarding the newly found landmarks. However, not all unmatched landmarks are added to the map. Another gate, the augmentation gate, is used to validate a new landmark. Again, the gated nearest-neighbor algorithm uses the $n_{is}$ value to determine the validity of the newly detected landmark. If this $n_{is}$ value is greater than the augmentation gate, the landmark is far enough away from other landmarks and is therefore accepted as a new landmark. New landmarks are added to the new landmark matrix $z_n$. This $2 \times f$ matrix is calculated in the same way that $z_f$ is calculated, storing the measured [range, bearing] of the jth observation, $(r_j, b_j)$, with $f$ representing the number of landmarks found. However, since there are no associations made, there is no matrix used to store ids for new landmarks.
6.4 Nearest-Neighbor Using Euclidean Distance

The Euclidean distance algorithm uses similar techniques as the Mahalanobis distance algorithm. Like the Mahalanobis distance algorithm, this algorithm is also implemented using the gated nearest-neighbor method. The Euclidean distance is also known as the ordinary distance between two points. No external factors, such as feature associations, are considered. The distance is simply calculated using the Pythagorean formula. For two sets of coordinates \(a = (a_1, a_2, ..., a_n)\) and \(b = (b_1, b_2, ..., b_n)\), the Euclidean distance between points \(a\) and \(b\) is calculated using Equation 6.9.

\[
d = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_n - b_n)^2}
\]

(6.9)

This formula, simplified for 2D, is shown in Equation 6.10.

\[
d = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}
\]

(6.10)

From observation, the 2D Euclidean distance formula requires three addition/subtraction operations and three power operations. Also, only four memory reads are required to gather the needed data. This is much more efficient than the Mahalanobis distance calculation, which requires numerous matrix multiplications and a matrix inversion, which become incredibly time intensive as the matrices grow in size.

Each stored landmark is used to calculate a distance based on each current landmark reading. This version of the gated nearest-neighbor data association algorithm uses the Euclidean distance equation from Equation 6.10 to calculate the distance used for the two gates.

6.5 Performance

Data association, like landmark extraction, plays a vital role in the success of the SLAM algorithm. Problems in this step can result in very few landmark re-observations or incorrect
associations. The gated nearest-neighbor data association algorithm based on the Mahalanobis
works very well, but it is computation intensive and may not be appropriate for real-time
applications.

Figure 32 shows test results from timing the Mahalanobis distance gated nearest-neighbor
association algorithm in MATLAB. These results represent the average execution time from
1000 trials using randomly generated data. As the number of landmarks increases, the algorithm
time complexity increases linearly. The algorithmic complexity is effectively $O(n)$, but the
important factor is the linear coefficient, 0.1763. This test shows how an increase in landmarks
can have a large effect on this algorithm’s computation time, with 1000 landmarks requiring
around 180ms. This execution time could be larger in a microcontroller environment, where
direct matrix optimizations may not be available.

![Figure 32. Association timing using Mahalanobis distance in MATLAB.](image)

By simply using the Euclidian distance, the computation time of the data association
algorithm is drastically reduced, especially as the number of landmarks in the map increases.
The asymptotic complexity of this version of the algorithm is still $O(n)$, but the coefficient is
much more respectable, 0.0003. Figure 33 shows a graphical representation of the timing results for the Euclidean distance version of this data association algorithm. Specifically, this algorithm requires around 0.35ms at 1000 landmarks, over 500 times faster than using the Mahalanobis distance. While the timing of this algorithm is more suitable for real-time implementations, the accuracy is lower than the Mahalanobis distance-based algorithm.

![Figure 33. Association timing using Euclidean distance in MATLAB.](image)

While the timing information presented in Figures 32 and 33 is meaningful, it is necessary to insert the Mahalanobis and Euclidean-based data association algorithms into the full IH-SLAM implementation to test which complete implementation is better. The IH-SLAM performance data already presented in Figures 23-26 used the Mahalanobis distance approach. New tests are performed using the Euclidean distance data association algorithm. The original environments shown in Figures 19-22 are used for testing again. Also, the RevoLDS, SRF05, and EZ1 sensor combinations are used for testing due to their high performance to cost ratios.
The results in Figures 34-37 represent the same tests from Section 5.6 executed using the Euclidean distance data association algorithm. It is safe to assume that the Mahalanobis distance association function will provide much better accuracy, so the final observation compares the timing improvement to cost ratios between the two algorithms for each sensor model. The sensor configurations are provided again for convenience in Table 11.

<table>
<thead>
<tr>
<th>Sensor Type</th>
<th>Configuration Name</th>
<th>Number of Sensors</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laser</td>
<td>RevoLDS</td>
<td>1</td>
<td>$30</td>
</tr>
<tr>
<td></td>
<td>SRF05-3</td>
<td>3</td>
<td>$90</td>
</tr>
<tr>
<td></td>
<td>SRF05-5</td>
<td>5</td>
<td>$150</td>
</tr>
<tr>
<td></td>
<td>SRF05-7</td>
<td>7</td>
<td>$210</td>
</tr>
<tr>
<td>Sonar</td>
<td>EZ1-3</td>
<td>3</td>
<td>$75</td>
</tr>
<tr>
<td></td>
<td>EZ1-5</td>
<td>5</td>
<td>$125</td>
</tr>
<tr>
<td></td>
<td>EZ1-7</td>
<td>7</td>
<td>$175</td>
</tr>
</tbody>
</table>

Figure 34. Data association results using Euclidean distance from Environment 1.
Figure 35. Data association results using Euclidean distance from Environment 2.

Figure 36. Data association results using Euclidean distance from Environment 3.
Figure 37. Data association results using Euclidean distance from Environment 4.

From Figure 34, the SRF05 and EZ1 sonar sensors (5 and 7 sensor combinations) provide the best results at around 80% improvement. The sonar sensor configurations consisting of three sensors did not perform well, while the RevoLDS laser scanner provides sub-par improvement values. In Environment 2 (Figure 35), the SRF05-5 and EZ1-5 sonar sensors provide the best results, again at around 80% improvement. Both 7-sensor configurations and the RevoLDS sensor provide moderate improvement while the 3-sensor configurations provide sub-par improvement values. Environment 3, as seen in Figure 36, provides an interesting pattern. The sonar sensor configurations for both sensor models actually provide worse results as the number of sensors increases. Specifically, the SRF05-7 configuration performs worse than the uncorrected position. The RevoLDS and the EZ1-3 and EZ-5 sensor configurations provide the best results, all around 80% improvement. Finally, Environment 4 from Figure 37 provides much more predictable results, with the sonar sensor configurations providing increasingly better
results as the number of sensors increases. In this environment, both 3-sensor configurations provide poor improvement results, both 5-sensor configurations and the RevoLDS provide moderate improvement results, and both 7-sensor configurations provide great results at around 80% improvement.

A comparison between the Mahalanobis and Euclidean distance tests is shown in Figure 38, which averages results from all four environments. In most of the sensor models, the Mahalanobis distance calculation provides better results than the Euclidean distance calculation. However, some sensor configurations, particularly EZ1-3 and EZ1-5, perform better using the Euclidean distance, which is partially due to the subpar performance of these sensor models under the Mahalanobis distance tests. Nonetheless, the Euclidean distance association method provides decent results, particularly with the RevoLDS and EZ1-5 sensor models providing improvement between 70% and 80%.

![Figure 38. Data association results comparing the two distance calculations.](image)
Timing for the algorithms was accomplished via the tic/toc functions in MATLAB, with all experiments performed on the same computer to keep the hardware consistent. Additionally, ten trials are averaged for each sensor/algorithm combination. For additional consistency among tests, the MATLAB process uses a real-time priority under Microsoft Windows XP and is given affinity to only one processor core. Figure 39 presents the timing improvement for each sensor using the Euclidean distance calculation over the Mahalanobis distance calculation.

Figure 39. Euclidean distance timing improvement over Mahalanobis distance.

Figure 39 proves that the RevoLDS showed drastic speedup while the sonar sensor configurations show very little improvement. This is likely due to the fact that the RevoLDS sensor provides much more data, so the data association function is called more frequently. The sonar sensors do show improvement, but this improvement is relatively small since the data association algorithm is not called as often. However, the improvement grows as the number of
sonar sensors grow for each sensor model, because the data association function is called more frequently since more sensors are capable of extracting data.

Based on the experimental results, it appears that the Euclidean distance calculation can be used instead of the Mahalanobis distance to decrease the computational load of the overall SLAM algorithm. However, this improvement is only advantageous when large numbers of landmarks are extracted or when there are a significant number of landmarks stored in the map. In reasonably small or sparse environments, it appears that the Mahalanobis distance calculation is more efficient when sonar sensors are used. However, when the RevoLDS (or any other laser scanner) is used, the Euclidean distance calculation is a viable option.
CHAPTER 7
FILTER UPDATE

7.1 Update Step

The fourth function in IH-SLAM implementation is the filter update step. There are a number of ways to approach the update operation, mostly related to matrix inversion and multiplication. The Kalman gain is calculated using Equation 7.1.

\[ K = PH^T (HPH^T + vRv^T)^{-1} \]  

(7.1)

The calculation performed inside the parentheses, \( HPH^T + vRv^T \), is actually another formulation of the innovation matrix \( S \). The matrix \( v \) is calculated using Equation 6.5. This time, \( v \) is directly incorporated into the calculation of \( S \). The resulting matrix \( S \) and the simpler calculation of \( K \) are provided in Equations 7.2 and 7.3.

\[ S = HPH^T + vRv^T \]  

(7.2)

\[ K = PH^T S^{-1} \]  

(7.3)

The matrix \( K \) indicates how much each landmark position and the robot position should be updated according to a specifically observed landmark. This gain is either calculated in individual steps for each observed landmark or in a batch function that encompasses all landmark pairs from the data association function. In the batch approach, less overhead is needed, but the matrices are larger and more difficult to calculate. IH-SLAM uses a batch update function, which allows for more optimization possibilities.
Based on the Kalman gain, a new state vector is computed, as shown in Equation 7.4.

\[ \mathbf{X} = \mathbf{X} + \mathbf{Kv} \]  

(7.4)

The Kalman gain \( \mathbf{K} \) and the innovation \( \mathbf{v} \) combine to calculate the adjustment needed for the state matrix \( \mathbf{X} \). Also, the covariance matrix \( \mathbf{P} \) must be updated to enhance the correlations between the observed landmarks, unobserved landmarks, and robot position. The calculation for \( \mathbf{P} \) is given in Equation 7.5.

\[ \mathbf{P} = \mathbf{P} - (\mathbf{PH}^T \mathbf{S}^{-1})(\mathbf{PH}^T \mathbf{S}^{-1})^T \]  

(7.5)

The calculation of \( \mathbf{X} \) and \( \mathbf{P} \) conclude the filter update (or state update) step of the IH-SLAM algorithm.

7.2 Optimizations

There are some optimization possibilities in the update function. Some of these optimizations can only be realized when implementing the IH-SLAM algorithm on a microcontroller. These optimizations are based on memory storage and caching potential. One optimization involves the inversion of the matrix \( \mathbf{S} \) using principles from the field of numerical linear algebra. Using a batch function, \( \mathbf{S} \) can grow to a problematic size. The maximum size of \( \mathbf{S} \) using a batch function is \( 2\alpha \times 2\alpha \), where \( \alpha \) represents the number of associated landmarks. Calculating the inverse of a \( 2 \times 2 \) matrix is straightforward, but as the dimensions grow, the difficulty of directly calculating the inverse grows exponentially. Some methods used to help calculate the inverse of large matrices include Gaussian elimination [48], LU decomposition [49], QR factorization [50], and Cholesky decomposition [51]. For IH-SLAM, the Cholesky decomposition method is used to calculate the inverse of \( \mathbf{S} \), due to its efficiency and numerical stability.
Cholesky decomposition is the decomposition of a symmetric, positive-definite matrix into the product of a lower triangular matrix and its transpose. Cholesky decomposition is useful when finding the inverse of $S$, since $S$ is both symmetric and positive-definite. This decomposition is efficient and very numerically stable. The details of the Cholesky decomposition function can be found in [51]. The pseudo-code for Cholesky decomposition is presented in Figure 40.

\[
\begin{align*}
R & = A \\
\text{FOR } k & = 1 \text{ to } m \\
& \quad \text{FOR } j = k + 1 \text{ to } m \\
& \quad \quad R[j,j:m] = R[j,j:m] - (R[k,j:m]*R[k,j])/R[k,k] \\
& \quad \quad \text{END FOR} \\
& \quad R[k,k:m] = R[k,k:m]/\sqrt{R[k,k]} \\
& \quad \text{END FOR} \\
\end{align*}
\]

Figure 40. Pseudo code for Cholesky decomposition.

Another optimization is possible in the update function since multiple matrix multiplications are performed. Because matrix multiplication is associative, the matrix-chain multiplication solution can be used to find an optimal multiplication ordering. Matrix-chain multiplication is an optimization that uses dynamic programming to find the most efficient way to multiply matrices together [52].

The update function is also unique in that there are other optimization possibilities involving the storage of temporary matrices. The following tables consider 1022 landmarks and 5 associated landmark pairs, used to define $n$ and $m$ in Equation 7.6 and 7.7.

\[
\begin{align*}
n & = (1022 \times 2) + 3 = 2047 \quad (7.6) \\
m & = 5 \times 2 = 10 \quad (7.7)
\end{align*}
\]

The originally defined multiplications needed for the update function are defined in Table 12.
Table 12. Standard update function matrix multiplications.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Matrix Sizes</th>
<th>Out</th>
<th>Size</th>
<th>Notes</th>
<th>Loops</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>P x H'</td>
<td>(n x n) x (n x m)</td>
<td>PHt</td>
<td>(n x m)</td>
<td></td>
<td>n^2m</td>
<td>41,902,090</td>
</tr>
<tr>
<td>H x PHt</td>
<td>(m x n) x (n x m)</td>
<td>SCI</td>
<td>(m x m)</td>
<td>symmetric</td>
<td>m^2n</td>
<td>204,700</td>
</tr>
<tr>
<td>PHt x SCI</td>
<td>(n x m) x (m x m)</td>
<td>W1</td>
<td>(n x m)</td>
<td></td>
<td>m^2n</td>
<td>204,700</td>
</tr>
<tr>
<td>W1 x SCI'</td>
<td>(n x m) x (m x m)</td>
<td>W</td>
<td>(n x m)</td>
<td></td>
<td>m^2n</td>
<td>204,700</td>
</tr>
<tr>
<td>W x v</td>
<td>(n x m) x (m x 1)</td>
<td>X</td>
<td>(n x 1)</td>
<td></td>
<td>nm</td>
<td>20,470</td>
</tr>
<tr>
<td>W1 x W1'</td>
<td>(n x m) x (m x n)</td>
<td>P</td>
<td>(n x n)</td>
<td>symmetric</td>
<td>n^2m</td>
<td>41,902,090</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84,438,750</td>
</tr>
</tbody>
</table>

The “Equation” column represents the mathematical calculation of each step of the update function; the “Matrix Sizes” column represents the sizes of both matrices multiplied in the equation; the “Out” column represents the resulting matrix; the “Notes” column represents interesting details about the resulting matrix; the “Loops” column represents the number of loops required to perform the multiplication; and the “Operations” column represents the number of operations required to perform the matrix multiplication, based on using the values of n and m from Equations 7.6 and 7.7.

By adjusting the order of the matrices to achieve an optimal order (using matrix-chain multiplication optimization), the following multiplication process is observed.

Table 13. Matrix-chain optimization for update function matrix multiplications.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Matrix Sizes</th>
<th>Out</th>
<th>Size</th>
<th>Loops</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>P x H'</td>
<td>(n x n) x (n x m)</td>
<td>PHt</td>
<td>(n x m)</td>
<td>n^2m</td>
<td>41,902,090</td>
</tr>
<tr>
<td>H x PHt</td>
<td>(m x n) x (n x m)</td>
<td>SCI</td>
<td>(m x m)</td>
<td>m^2n</td>
<td>204,700</td>
</tr>
<tr>
<td>PHt x SCI</td>
<td>(n x m) x (m x m)</td>
<td>W1</td>
<td>(n x m)</td>
<td>m^2n</td>
<td>204,700</td>
</tr>
<tr>
<td>W1 x SCI'</td>
<td>(n x m) x (m x m)</td>
<td>W</td>
<td>(n x m)</td>
<td>m^2n</td>
<td>204,700</td>
</tr>
<tr>
<td>W x v</td>
<td>(n x m) x (m x 1)</td>
<td>X</td>
<td>(n x 1)</td>
<td>nm</td>
<td>20,470</td>
</tr>
<tr>
<td>W1 x W1'</td>
<td>(n x m) x (m x n)</td>
<td>P</td>
<td>(n x n)</td>
<td>symmetric n^2m</td>
<td>41,902,090</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84,235,250</td>
</tr>
</tbody>
</table>

While the total number of multiplications required for the optimal matrix-chain order is improved, the improvement is small, 0.24%. This difference is even smaller for small numbers of landmarks. However, this is a slight improvement nonetheless in the computational requirements of the algorithm.
The next observation shows that the matrix multiplication chains $\text{SCI} \times \text{SCI}^T$ and $\text{PHt} \times \text{SCI} \times \text{SCI}^T$ are each used twice. Therefore, temporary storage can be used to save multiple identical operations. However, by using temporary matrix storage, some of the effects of matrix-chain optimization are lost. Nevertheless, the first intermediate temporary matrix $T_1$ is calculated using Equation 7.8.

$$T_1 = \text{SCI} \times \text{SCI}^T \quad (7.8)$$

The results from using this temporary matrix as an optimization are presented in Table 14.

Table 14. Temporary storage, $T_1$, optimization for update function matrix multiplications.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Matrix sizes</th>
<th>Out</th>
<th>Size</th>
<th>Loops</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \times H'$</td>
<td>$(n \times n) \times (n \times m)$</td>
<td>PHt</td>
<td>$(n \times m)$</td>
<td>$n^3m$</td>
<td>41,902,090</td>
</tr>
<tr>
<td>$H \times PHt$</td>
<td>$(m \times n) \times (n \times m)$</td>
<td>SCI</td>
<td>$(m \times m)$</td>
<td>$m^2n$</td>
<td>204,700</td>
</tr>
<tr>
<td>$\text{SCI} \times \text{SCI}'$</td>
<td>$(m \times m) \times (m \times m)$</td>
<td>$T_1$</td>
<td>$(m \times m)$</td>
<td>$m^3$</td>
<td>1,000</td>
</tr>
<tr>
<td>$\text{PHt} \times (T_1 \times v)$</td>
<td>$((n \times m) \times (m \times m)) \times (m \times 1)$</td>
<td>X</td>
<td>$(n \times 1)$</td>
<td>$m^2 + nm$</td>
<td>20,570</td>
</tr>
<tr>
<td>$(\text{PHt} \times T_1) \times PHt'$</td>
<td>$(n \times m) \times (m \times n)$</td>
<td>$P$</td>
<td>$(n \times n)$</td>
<td>$n^2m + n^3m$</td>
<td>42,106,790</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>84,235,150</strong></td>
</tr>
</tbody>
</table>

Once again, this is a very slight improvement, by only 100 operations. The next intermediate matrix, $T_2$, is calculated using Equation 7.9.

$$T_2 = \text{PHt} \times \text{SCI} \times \text{SCI}^T \quad (7.9)$$

The results of this optimization are presented in Table 15. With this optimization, there is another slight improvement of 100 operations.

Table 15. Temporary storage, $T_2$, optimization for update function matrix multiplications.

<table>
<thead>
<tr>
<th>Equation</th>
<th>Matrix sizes</th>
<th>Out</th>
<th>Size</th>
<th>Notes</th>
<th>Loops</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \times H'$</td>
<td>$(n \times n) \times (n \times m)$</td>
<td>PHt</td>
<td>$(n \times m)$</td>
<td></td>
<td>$n^3m$</td>
<td>41,902,090</td>
</tr>
<tr>
<td>$H \times PHt$</td>
<td>$(m \times n) \times (n \times m)$</td>
<td>SCI</td>
<td>$(m \times m)$</td>
<td>symmetric</td>
<td>$m^2n$</td>
<td>204,700</td>
</tr>
<tr>
<td>$\text{SCI} \times \text{SCI}'$</td>
<td>$(m \times m) \times (m \times m)$</td>
<td>$T_1$</td>
<td>$(m \times m)$</td>
<td>symmetric</td>
<td>$m^3$</td>
<td>1,000</td>
</tr>
<tr>
<td>$\text{PHt} \times T_1$</td>
<td>$(n \times m) \times (m \times m)$</td>
<td>$T_2$</td>
<td>$(n \times m)$</td>
<td></td>
<td>$n^3m$</td>
<td>204,700</td>
</tr>
<tr>
<td>$(T_2 \times v)$</td>
<td>$(n \times m) \times (m \times 1)$</td>
<td>X</td>
<td>$(n \times 1)$</td>
<td></td>
<td>nm</td>
<td>20,470</td>
</tr>
<tr>
<td>$T_2 \times PHt'$</td>
<td>$(n \times m) \times (m \times n)$</td>
<td>$P$</td>
<td>$(n \times n)$</td>
<td>symmetric</td>
<td>$n^3m$</td>
<td>41,902,090</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>84,235,050</strong></td>
</tr>
</tbody>
</table>
It is evident that by using these optimizations, a small amount of improvement in the number of operations is possible. However, this improvement is only 0.24% for a large number of stored landmarks and a large number of associated landmarks. This improvement will be even smaller for smaller number of landmarks or less associated landmarks. Nonetheless, this optimization does not affect the accuracy of the SLAM algorithm and does provide a slight timing improvement.
8.1 Map Insertion Step

After the update function calculates $X$ and $P$, the system is in an updated state. The only function left to perform is map insertion. The map insertion function is also referred to as augmentation, since the $X$ and $P$ matrices must be augmented in size to accommodate the new landmarks.

The previously calculated matrix $z_n$ is used in this operation, and two final Jacobians are also generated for each new landmark. First, for the $i$th landmark in $z_n$, the landmark position is added to the augmented $X$ matrix using Equation 8.1.

$$X = \begin{bmatrix} x_r + (z_{n_1} \times \cos(\theta_r + z_{n_2})) \\ y_r + (z_{n_1} \times \sin(\theta_r + z_{n_2})) \end{bmatrix}$$  \hspace{1cm} (8.1)$$

Next, the $P$ matrix is updated. The augmentation of $P$ is more complicated, because Jacobians must be calculated to correlate the new landmark with previous landmarks and the robot position. The first matrix $G_v$ is the Jacobian of the landmark prediction model with respect to the robot position. For the $i$th landmark in $z_n$ the matrix is calculated using Equation 8.2.

$$G_v = \begin{bmatrix} 1 & 0 & -z_{n_1} \times \sin(\theta_r + z_{n_2}) \\ 0 & 1 & z_{n_1} \times \cos(\theta_r + z_{n_2}) \end{bmatrix}$$  \hspace{1cm} (8.2)$$
The second matrix, $G_z$, is the Jacobian of the prediction model of the landmark with respect to the measured [range, bearing]. The calculation of this matrix is given in Equation 8.3.

\[
G_z = \begin{bmatrix}
\cos(\theta_r + z_{n_i1}) & -z_{n_i1} \times \sin(\theta_r + z_{n_i2}) \\
\sin(\theta_r + z_{n_i1}) & z_{n_i1} \times \cos(\theta_r + z_{n_i2})
\end{bmatrix}
\]  \quad (8.3)

After calculation of both Jacobians, the final augmented $P$ matrix is updated. The augmentation of $P$ inserts two rows and two columns at the bottom and right of $P$, respectively. Next, the blank, bottom-right $2 \times 2$ matrix, the feature covariance, is calculated using Equation 8.4, with $p$ representing the new dimension of $P$.

\[
P_{(p-1:p),(p-1:p)} = G_v P_{(1:3),(1:3)}^T + G_z R G_z^T
\]  \quad (8.4)

This leads to the calculation of the robot position to feature covariance, shown in Equation 8.5.

\[
P_{(p-1:p),(1:3)} = G_v P_{(1:3),(1:3)}
\]  \quad (8.5)

Again, since $P$ is symmetric, the following representation is made.

\[
P_{(1:3),(p-1:p)} = P_{(p-1:p),(1:3)}^T
\]  \quad (8.6)

Also, the feature to map covariance is calculated.

\[
P_{(p-1:p),(4:p)} = G_v P_{(1:3),(4:p)}
\]  \quad (8.7)

The final calculation is to transpose the feature to map covariance, as shown in Equation 8.8.

\[
P_{(4:p),(p-1:p)} = P_{(p-1:p),(4:p)}^T
\]  \quad (8.8)
CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Results and Conclusions

The IH-SLAM algorithm was developed and tested using MATLAB. A model depicting the MATLAB implementation of the algorithm is presented in Figure 41.

![MATLAB IH-SLAM Algorithm Model](image)

Figure 41. MATLAB IH-SLAM Algorithm Model.

The main loop is controlled by the “Simulation Driver” function. The “IH-SLAM Algorithm” function encompasses the five main steps, or functions, described in this thesis. The “Initialization” file represents control information used to configure the simulation. These configurations include the type of sensor used and the method for data association. The error
results calculated in the “Simulation Driver” function are written to the “Simulation Results” file, which is either viewed by the user at the end of the simulation used during batch testing.

Figure 42 presents an example of how IH-SLAM could be implemented on physical hardware. Two microcontrollers are used in order to reduce the computational load on the robot’s main microcontroller. Since a dedicated SLAM microcontroller is used, the main microcontroller is only required to receive the updated \([x, y, \theta]\) position from the SLAM microcontroller. Therefore, the goal of the SLAM microcontroller is to perform all five steps of the IH-SLAM algorithm before new distance sensor readings are provided. Based on the sensor models used in this thesis, the timing limit for the SLAM microcontroller to complete one iteration ranges from 36-100ms. Future research can port the IH-SLAM algorithm to an inexpensive microcontroller in order to test the timing feasibility of this hardware implementation.

Figure 42. Hardware IH-SLAM implementation.
This thesis provides an overview of SLAM, and the specific IH-SLAM algorithm is presented. In many of the steps of this algorithm, problems are identified related to cost, timing, and memory constraints when implementing SLAM on a real-time system (the robot). Therefore, optimizations are presented to allow the algorithm to operate within these real-time constraints.

Chapter 5 identifies the main cost problem related to implementing SLAM: the sensor hardware. To remedy this problem, low-cost sensor configurations are proposed, and a new landmark extraction technique is presented. This new technique, BED, uses sonar sensors and provides nearly identical results to laser scanners in simulation. Therefore, one of the major costs associated with the SLAM algorithm is reduced considerably. Breaking the SLAM algorithm’s dependence on laser scanners is one of the main reasons for this research.

Chapter 6 identifies one of the major timing problems related to the SLAM algorithm: the data association function. Two solutions (Mahalanobis and Euclidean nearest neighbor) to feature-based data association are presented with each solution providing its own set of tradeoffs. Testing proves that the Euclidean nearest neighbor method has the potential to significantly reduce the algorithm’s timing requirements. However, the difference in timing compared to the difference in accuracy makes the Euclidean-based algorithm a less efficient solution. Nonetheless, SLAM implementations in larger and more cluttered environments that provide timing strains on the microcontroller can sacrifice accuracy to continue to meet real-time requirements by switching to the Euclidean nearest-neighbor method.

Chapter 7 shows the mathematical basis for the timing complexity of the SLAM algorithm, specifically in the update function: matrix multiplication. A few optimizations for
matrix multiplication and theoretical results are presented to show the decrease in operations possible.

This thesis presents a comprehensive IH-SLAM algorithm that could be implemented on a low-cost microcontroller while potentially meeting real-time constraints. The sensor and processing hardware used in the described hardware implementation costs between $30 and $210 to implement and does not affect the robot’s main microcontroller. Therefore, two of the original problems with SLAM (cost and timing) are addressed, and optimizations are provided to effectively implement SLAM using inexpensive hardware.

9.2 Future Work

While this thesis presents a successful approach to implementing SLAM using low-cost hardware, these results can be potentially approved and extended. This thesis presents a number of optimizations that help improve the performance of the IH-SLAM algorithm, but there are a few other options that can provide even better results. This section presents these options and other extensions to the IH-SLAM algorithm.

The IH-SLAM algorithm could be ported to ANSI-C for implementation on a physical microcontroller. Testing of this ANSI-C algorithm would provide validation of the real-time characteristics of IH-SLAM. Additionally, memory constraints could be analyzed to show the relationship between the number of landmarks and required size of the random access memory (RAM) on the microcontroller.

In this thesis, the sensor data is modeled and simulated based on physical testing of sensors and hardware specifications. However, this modeled data is not perfect. So, to get a better estimate of the accuracy of the IH-SLAM algorithm, specifically the BED landmark
extraction algorithm, real sensor hardware could be incorporated into the simulation. The easiest way to do this is to implement IH-SLAM on a microcontroller and attach real hardware to the design, via a hardware-in-the-loop (HWIL) setup.

Another way to improve accuracy is to use better sensor models for simulation. In this thesis, it is assumed that the sensor beams were simple geometric shapes (specifically a cone). However, the actual beams are not perfectly cone shaped in real hardware. Therefore, a higher fidelity sensor model could be incorporated into the simulation.

One optimization that is not implemented in this thesis is to use a better motion model in the BED algorithm. This is originally suggested in Chapter 5. Using a more complicated algebraic formula, a better motion model could be developed to extract more landmarks without extracting any incorrect landmarks. Additionally, this more complicated motion model could be compared to the original motion model in order to determine if there is enough improvement to validate using more complicated equations and the resulting computational requirements.

Another way to achieve greater speedup is to use a field programmable gate array (FPGA) to implement IH-SLAM in hardware. FPGA’s are great for prototyping, and the IH-SLAM algorithm could potentially be designed on an FPGA and later implemented on an application specific integrated circuit (ASIC). However, there are a number of design challenges for an FPGA implementation. Some of these challenges include memory management, matrix operations, and finding parallelization opportunities within each function in IH-SLAM.

The IH-SLAM algorithm is based on EKF SLAM. However, future research could attempt to implement similar optimizations using particle filters or the FastSLAM algorithm. A comparison of IH-SLAM and similar implementations using other types of filters could provide an improved SLAM algorithm with respect to accuracy, processing time, cost, and memory.
Other areas of research within the SLAM community include dynamic environments, sub-maps, closing the loop, multi-robot SLAM, and 3D SLAM. These areas of research could be implemented using the IH-SLAM algorithm as well.

Environments are usually dynamic, and therefore, work has been done to customize SLAM to work in a dynamic environment. The typical approach to this problem is to use filtering techniques to track moving landmarks in much the same way that radar tracks a target. Dynamic environments are also problematic in that static and dynamic landmarks must be differentiated. Typically, this problem is solved using detection of moving objects (DTMO) algorithms. Work on dynamic environment SLAM can be found in [53] and [54].

Another area of research in SLAM is in the partitioning of a single map into sub-maps. The algorithm will optimally fuse sub-maps together into a global map periodically. Sub-mapping in SLAM reduces linearization errors in the EKF and reduces computational complexity similar to the FastSLAM algorithm, but it reduces the amount of correlation between landmarks. Also, the number of landmarks in each sub-map will likely not be equal, therefore making the algorithm less efficient. Specific sub-mapping algorithms include ATLAS [55], hybrid-SLAM [56], decoupled stochastic mapping (DSM) [57], and hierarchical SLAM [58].

One important concept that has been studied extensively in large scale environments is “closing the loop”. Closing the loop is important because even with small numbers of re-observations, the robot will continue to move into new areas and its position error will still accumulate. However, as the robot enters a previously observed area, it should be able to recognize the landmarks in this area and coincidentally reduce its position error back to the position error when it originally observed the landmarks. Methods for closing the loop typically
include map matching (feature matching), the expectation maximization algorithm, and non-linear optimization. Examples of closing the loop can be found in [59] and [60].

Typically SLAM is implemented on a single robot, but the general algorithm can be implemented on multiple robots. Multi-robot SLAM builds on the concepts of team map building, information sharing, and robotic cooperation in order to make more accurate and efficient maps. These implementations are typically represented by a network of robots that share map information with each other. Multi-robot SLAM has been implemented in [61].

SLAM has been introduced in this thesis in a 2D scale. SLAM can be applied in a 3D scale, but this approach becomes much harder when actually implemented. This is because of the larger number of calculations, the larger amount of possible error, and the general larger scale of the problem. Work in this research area includes [17], [62], [63], [64], [65].

Finally, the IH-SLAM algorithm could be tested using a number of different options which are not presented in this thesis. Other types of maps could be used in order to reduce computing time of the algorithm. Additionally, while a number of landmark extraction and data association algorithms are presented in this thesis, there are additional algorithms and methods that could be tested and compared. Also, vision-based sensors could be implemented into the MATLAB implementation’s observation model of IH-SLAM.
REFERENCES


APPENDIX 1

function [errors, data, sensor_data, time] = simulator(wp, lm)

format compact
init; % initialize general variables
SRF05_5; % initialize sensors
mahalanobis_neighbor; % initialize data association method

time = zeros(100,10);

% error check for graphics
if (USE_GRAPHICS == 0)
    DRAW = 0;
end

% setup figure and plots
if (USE_GRAPHICS == 1)
    fig = figure;
    odometry_axes = axes('position',[.1 .1 .8 .6]);
    whitebg('w');
    plot(lm(1,:),lm(2,:), 'b*')
    hold on, axis equal
    plot(wp(1,:),wp(2,:), 'w', wp(1,:),wp(2,:),'g.'),
    h = setup_animations;
    sensor_axes = axes('position',[.1 .8 .8 .2]);
    set(fig, 'name', 'University of Alabama SLAM Simulator')
    veh = [0 -WHEELBASE -WHEELBASE; 0 -WHEELBASE WHEELBASE]; % vehicle animation
    pcount = 0;
end

if (EXTRACTION_TYPE == 2 || EXTRACTION_TYPE == 3) % sonar or IR
    landmark_table = zeros(3,length(SENSORS)); % sonar or IR
    sensor_table = zeros(LANDMARK_COUNT,length(SENSORS));
    state_table = zeros(LANDMARK_COUNT,3);
end

pause(5);

% Main loop ----------------------------------------------------------
while (iwp ~= 0)

    n = (length(X)-3)/2;
    % Compute true data and predict
[lwdist, rwdist, xtrue, xuncorrected, iwp] = odometry_driver(wp, X(1:3),
xtrue, xuncorrected, iwp, control_time, AT_WAYPOINT, V, USE_ODOMETRY_ERROR,
q_max, q_min, WHEELBASE);

% predict function
 tic
 predict(lwdist, rwdist, c, WHEELBASE, control_time);
 time(n+1, 1) = time(n+1, 1) + toc;
 time(n+1, 2) = time(n+1, 2) + 1;

% Compute sensors
 if (EXTRACTION_TYPE == 1) % laser
 sensors = laser_sensor_driver(lm, xtrue, scan_width, angle_resolution,
 max_range, min_range, sigmaR, noise);
 elseif (EXTRACTION_TYPE == 2) % sonar
 sensors = sonar_sensor_driver(lm, xtrue, max_range, min_range,
 beam_width, SENSORS, WHEELBASE, noise, sigmaR);
 elseif (EXTRACTION_TYPE == 3) % ir
 sensors = ir_sensor_driver(lm, xtrue, max_range, min_range, SENSORS,
 WHEELBASE, noise, sigmaR);
 end

% Extract landmarks
 tic
 if (EXTRACTION_TYPE == 1) % laser
 [z, observed_landmarks, landmarks_found] =
 spikes_landmark_extractor(sensors, xtrue, extract_thresh_max, max_range,
 min_range, scan_width, angle_resolution);
 elseif (EXTRACTION_TYPE == 2) % sonar
 [z, landmark_table, sensor_table, state_table, observed_landmarks,
 landmarks_found] = sonar_landmark_extractor(lm, sensors, X(1:3), lwdist,
 rwdist, max_range, min_range, SENSORS, beam_width, LANDMARK_COUNT,
 landmark_table, sensor_table, state_table, WHEELBASE);
 elseif (EXTRACTION_TYPE == 3) % ir
 [z, sensor_table, observed_landmarks, landmarks_found] =
 ir_landmark_extractor(lm, sensors, X(1:3), max_range, min_range, SENSORS,
 LANDMARK_COUNT, sensor_table, WHEELBASE);
 end
 time(n+1, 3) = time(n+1, 3) + toc;
 time(n+1, 4) = time(n+1, 4) + 1;

% only if landmarks are found
 if (landmarks_found > 0)
 % data association function
 tic
 [zf, idf, zn] = data_associate(X, P, z, R, observed_landmarks,
 ASSOCIATION_TYPE, GATE_AUGMENT, GATE_REJECT);
 time(n+1, 5) = time(n+1, 5) + toc;
 time(n+1, 6) = time(n+1, 6) + 1;

 if (USE_SLAM_CORRECTION == 1)
 % update function
 tic
 update(zf, R, idf);
 time(n+1, 7) = time(n+1, 7) + toc;
 time(n+1, 8) = time(n+1, 8) + 1;
 end

end

% map insertion function
tic
augment(zn, R);
n = (length(X)-3)/2;
time(n+1, 9) = time(n+1, 9) + toc;
time(n+1, 10) = time(n+1, 10) + 1;
end

% error calculations-----------------------------------------------

errors(count, 1) = sqrt((X(1)-xtrue(1))^2 + (X(2)-xtrue(2))^2); % corrected error
errors(count, 2) = sqrt((xuncorrected(1)-xtrue(1))^2 + (xuncorrected(2)-xtrue(2))^2)); % uncorrected error
data(count, 1:3) = X(1:3);
data(count, 4:6) = xtrue(1:3);
data(count, 7:9) = xuncorrected(1:3);
data(count, 10) = lwdist;
data(count, 11) = rwdist;
sensor_data(count,:) = sensors;
count = count + 1; % increment counter

% visuals-------------------------------------------------------------

% Plots
if (DRAW == 1 & USE_ODOMETRY_ERROR == 1)
pvcov= make_vehicle_covariance_ellipse(X,P);
set(h.vcov, 'xdata', pvcov(1,:), 'ydata', pvcov(2,:))
end

if (DRAW == 1)
pcount= pcount+control_time;
if (pcount > 0.4)
  pcount=0;
  axes(odometry_axes);
  plot(X(1,1), X(2,1),'b');
  plot(xtrue(1), xtrue(2), 'r');
  plot(xuncorrected(1), xuncorrected(2), 'g');
  axes(sensor_axes);
  % update sensor visuals
  if (EXTRACTION_TYPE == 1)
    plot((-180:179), sensors);
    axis([-180 180 0 max_range+1]);
    axes(odometry_axes);
  elseif (EXTRACTION_TYPE == 2)
    temp = sensors;
    sensors = zeros(1,360);
    for i = 1:length(SENSORS)
      x1 = 181 - (ceil((SENSORS(i) - beam_width/2) * 180/pi));
      x2 = 181 - (ceil((SENSORS(i) + beam_width/2) * 180/pi));
      sensors(x2:x1) = temp(i);
    end
    plot((-180:179), sensors);
  end
end
elseif (EXTRACTION_TYPE == 3)
    temp = sensors;
    sensors = zeros(1,360);
    for i = 1:length(SENSORS)
        sensors(181 - SENSORS(i)*180/pi) = temp(i);
    end
    plot((-180:179), sensors);
    axis([-180,180,0,max_range+1]);
    axes(odometry_axes);
end

if DRAW == 1 % plots related to observations
    set(h.xf, 'xdata', X(4:2:end), 'ydata', X(5:2:end))
    pfcov = make_feature_covariance_ellipses(X,P);
    set(h.fcov, 'xdata', pfcov(1,:), 'ydata', pfcov(2,:))
end
if (DRAW == 1)
    drawnow
end
end % end of main loop

% draw final positions
if (USE_GRAPHICS == 1)
    axes(odometry_axes);
    plot(X(1,1), X(2,1),'kv');
    plot(xtrue(1), xtrue(2), 'mv');
    plot(xuncorrected(1,1), xuncorrected(2,1),'bv');
end

clear global X
clear global P
% end function -----------------------------------------------

% from Tim Bailey
function h= setup_animations()
    h.xt= patch(0,0,'b','erasemode','xor'); % vehicle true
    h.xv= patch(0,0,'r','erasemode','xor'); % vehicle estimate
    h.pth= plot(0,0,'k.','markersize',2,'erasemode','background'); % vehicle path estimate
    h.obs= plot(0,0,'y','erasemode','xor'); % observations
    h.xf= plot(0,0,'r+','erasemode','xor'); % estimated features
    h.vcov= plot(0,0,'r','erasemode','xor'); % vehicle covariance ellipses
    h.fcov= plot(0,0,'r','erasemode','xor'); % feature covariance ellipses
% from Tim Bailey
function p= make_vehicle_covariance_ellipse(x,P)
% compute ellipses for plotting vehicle covariances
    N= 10;
    inc= 2*pi/N;
    phi= 0:inc:2*pi;
    circ= 2*[cos(phi); sin(phi)];
p = make_ellipse(x(1:2), P(1:2,1:2), circ);

% from Tim Bailey
function p = make_feature_covariance_ellipses(x,P)
% compute ellipses for plotting feature covariances
N = 10;
inc = 2*pi/N;
phi = 0:inc:2*pi;
circ = 2*[cos(phi); sin(phi)];

lenx = length(x);
lenf = (lenx-3)/2;
p = zeros (2, lenf*(N+2));
ctr = 1;
for i=1:lenf
    ii = ctr:(ctr+N+1);
    jj = 2+2*i; jj= jj:jj+1;
    p(:,ii) = make_ellipse(x(jj), P(jj,jj), circ);
    ctr = ctr+N+2;
end

% from Tim Bailey
function p = make_ellipse(x,P,circ)
% make a single 2-D ellipse
r = sqrtm_2by2(P);
a = r*circ;
p(2,:) = [a(2,:) + x(2) NaN];
p(1,:) = [a(1,:) + x(1) NaN];

function augment(z,R)
% Tim Bailey 2004.
% add new features to state
for i=1:size(z,2)
    add_one_z(z(:,i),R);
end

function add_one_z(z,R)
global X P

len= length(X);
r= z(1); b= z(2);
s= sin(X(3)+b);
c= cos(X(3)+b);

% augment x
X= [X;
    X(1) + r*c;
    X(2) + r*s];

% jacobians
Gv= [1 0 -r*s;
     0 1  r*c];
Gz = \begin{bmatrix} c & -r*s \\ s & r*c \end{bmatrix};

% augment P
rng = len + 1:len + 2;
P(rng, rng) = Gv * P(1:3, 1:3)' * Gv + Gz * R * Gz'; % feature cov
P(rng, 1:3) = Gv * P(1:3, 1:3); % vehicle to feature xcorr
P(1:3, rng) = P(rng, 1:3)';
if len > 3
 rngm = 4:len;
P(rng, rngm) = Gv * P(1:3, rngm); % map to feature xcorr
P(rngm, rng) = P(rng, rngm)';
end

function [result] = check_view_angle(phi, arc_center, angle)
% determine if an angle is within a specific range
if ((phi < arc_center + angle/2) & (phi > arc_center - angle/2))
    result = 1;
else
    result = 0;
end

function [zf, idf, zn] = data_associate(x, P, z, R, obs, ASSOCIATION_TYPE, GATE_AUGMENT, GATE_REJECT)
zf = [];
zn = [];
idf = [];
if (ASSOCIATION_TYPE == 1) % euclidean
    Nf = (length(x) - 3)/2;
    for i = 1:size(z, 2)
        xa = x(1) + z(1, i) * cos(z(2, i) + x(3));
        ya = x(2) + z(1, i) * sin(z(2, i) + x(3));
        dist = 100;
        found = 0;
        for j = 1:Nf
            j_f = j * 2 + 3;
            xb = x(j_f - 1);
            yb = x(j_f);
            d = sqrt((xa - xb)^2 + (ya - yb)^2);
            if (d < dist)
                dist = d;
                found = j;
            end
        end
        if (dist < GATE_REJECT) % found an association
            zf = [zf ; z(:, i)];
            idf = [idf ; found];
            % plot(obs(1, i), obs(2, i), 'c.');
        elseif (dist > GATE_AUGMENT) % did not find an association and outside of gate augment
            zn = [zn ; z(:, i)];
            % plot(obs(1, i), obs(2, i), 'm.');
        end
    end
end
elseif (ASSOCIATION_TYPE == 2) % mahalanobis
    Nxv = 3; % number of vehicle pose states
    Nf = (length(x) - Nxv)/2; % number of features already in map

    for i = 1:size(z,2)
        jbest = 0;
        nbest = inf;
        outer = inf;

        % search for neighbors
        for j = 1:Nf
            [nis, nd] = compute_association(x, P, z(:,i), R, j);
            if (nis < GATE_REJECT && nd < nbest) % if within gate, store nearest-neighbor
                nbest = nd;
                jbest = j;
            elseif (nis < outer) % else store best nis value
                outer = nis;
            end
        end

        % add nearest-neighbor to association list
        if (jbest ~= 0)
            zf = [zf z(:,i)];
            idf = [idf jbest];
            %plot(obs(1,i),obs(2,i),'c.);
        elseif (outer > GATE_AUGMENT) % z too far to associate, but far enough to be a new feature
            zn = [zn z(:,i)];
            %plot(obs(1,i),obs(2,i),'m.);
        end
    end

function [nis, nd] = compute_association(x, P, z, R, idf)
    % return normalized innovation squared (ie, Mahalanobis distance) and normalized distance
    [zp,H] = observe_model(x, idf);
    v = z - zp;
    v(2) = pi_to_pi(v(2));
    S = computeS(P,H,R,idf);

    nis = v'*inv(S)*v;
    nd = nis + log(det(S));

function S = computeS(P, H, R, idf)
    % faster computation of S -- H is sparse
    jj = 2 + idf*2;
    ii = [1:3 jj:(jj+1)];
    H = H(:,ii);
    Pt = P(ii,ii);
    S = H*Pt*H' + R;
function [phi] = find_angle(dx, dy, relative_angle)

if (dy > 0) % find angle compared to global
    phi = pi/2 - atan(dx/dy);
elseif (dy < 0)
    phi = (3*pi)/2 - atan(dx/dy);
end
phi = phi - relative_angle; % convert to relative to robot angle
if (phi > pi)
    phi = phi - 2*pi;
elseif (phi < -pi)
    phi = phi + 2*pi;
end

function varargout = frontend(varargin)
%EKF-SLAM environment-making GUI
%
% This program permits the graphical creation and manipulation
% of an environment of point landmarks, and the specification of
% vehicle path waypoints therein.
%
% USAGE: type 'frontend' to start.
% 1. Click on the desired operation: <enter>, <move>, or <delete>.
% 2. Click on the type: <waypoint> or <landmark> to commence the
%    operation.
% 3. If entering new landmarks or waypoints, click with the left
%    mouse button to add new points. Click the right mouse button, or
%    hit <enter> key to finish.
% 4. To move or delete a point, just click near the desired point.
% 5. Saving maps and loading previous maps is accomplished via the
%    <save> and <load> buttons, respectively.
%
% Tim Bailey and Juan Nieto 2004.

% FRONTEND Application M-file for frontend.fig
% FIG = FRONTEND launch frontend GUI.
% FRONTEND('callback_name', ...) invoke the named callback.
global WAYPOINTS LANDMARKS FH

if nargin == 0  % LAUNCH GUI

    %initialisation
    WAYPOINTS= [0;0];
    LANDMARKS= [];

    % open figure
    fig = openfig(mfilename,'reuse');
    hh= get(fig, 'children');
    set(hh(3), 'value', 1)
    hold on
    FH.hl = plot(0,0,'g*'); plot(0,0,'w*')
    FH.hw = plot(0,0,0,0,'ro');
    FH.hln = plot(0,0,0,0,'-pk');
    plotwaypoints(WAYPOINTS);
% Use system color scheme for figure:
set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));
set(fig,'name', 'SLAM Map-Making GUI')

% Generate a structure of handles to pass to callbacks, and store it.
handles = guihandles(fig);
guidata(fig, handles);

if nargout > 0
    varargout{1} = fig;
end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK
    try
        [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
    catch
        disp(lasterr);
    end
end

% --------------------------------------------------------------------
% function varargout = waypoint_checkbox_Callback(h, eventdata, handles, varargin)
%     global WAYPOINTS
%     set(handles.landmark_checkbox, 'value', 0)
%     WAYPOINTS= perform_task(WAYPOINTS, handles.waypoint_checkbox, handles);
%     plotwaypoints(WAYPOINTS);
%
% --------------------------------------------------------------------
% function varargout = landmark_checkbox_Callback(h, eventdata, handles, varargin)
%     global LANDMARKS
%     set(handles.waypoint_checkbox, 'value', 0)
%     LANDMARKS= perform_task(LANDMARKS, handles.landmark_checkbox, handles);
%     plotlandmarks(LANDMARKS);
%
% --------------------------------------------------------------------
% function varargout = enter_checkbox_Callback(h, eventdata, handles, varargin)
%     set(handles.enter_checkbox, 'value', 1)
%     set(handles.move_checkbox, 'value', 0)
%     set(handles.delete_checkbox, 'value', 0)
%
% --------------------------------------------------------------------
% function varargout = move_checkbox_Callback(h, eventdata, handles, varargin)
%     set(handles.enter_checkbox, 'value', 0)
%     set(handles.move_checkbox, 'value', 1)
%     set(handles.delete_checkbox, 'value', 0)
%
% --------------------------------------------------------------------
% function varargout = delete_checkbox_Callback(h, eventdata, handles, varargin)
set(handles.enter_checkbox, 'value', 0)
set(handles.move_checkbox, 'value', 0)
set(handles.delete_checkbox, 'value', 1)

% --------------------------------------------------------------------
function varargout = load_button_Callback(hObject, eventdata, handles, varargin)
    global WAYPOINTS LANDMARKS
    seed = {'*.mat','MAT-files (*.mat)'};
    [fn,pn] = uigetfile(seed, 'Load landmarks and waypoints');
    if fn==0, return, end
    fnpn = strrep(fullfile(pn,fn), '''', '''''');
    load(fnpn)
    WAYPOINTS= wp; LANDMARKS= lm;
    plotwaypoints(WAYPOINTS);
    plotlandmarks(LANDMARKS);

% --------------------------------------------------------------------
function varargout = save_button_Callback(hObject, eventdata, handles, varargin)
    global WAYPOINTS LANDMARKS
    wp= WAYPOINTS; lm= LANDMARKS;
    seed = {'*.mat','MAT-files (*.mat)'};
    [fn,pn] = uiputfile(seed, 'Save landmarks and waypoints');
    if fn==0, return, end
    fnpn = strrep(fullfile(pn,fn), '''', '''''');
    save(fnpn, 'wp', 'lm');

% --------------------------------------------------------------------
function plotwaypoints(x)
    global FH
    set(FH.hw(1), 'xdata', x(1,:), 'ydata', x(2,:))
    set(FH.hw(2), 'xdata', x(1,:), 'ydata', x(2,:))

% --------------------------------------------------------------------
function plotlandmarks(x)
    global FH
    set(FH.hl, 'xdata', x(1,:), 'ydata', x(2,:))

% --------------------------------------------------------------------
function i= find_nearest(x)
    xp= ginput(1);
    d2= (x(1,:)-xp(1)).^2 + (x(2,:)-xp(2)).^2;
    i= find(d2 == min(d2));
    i= i(1);

% --------------------------------------------------------------------
function x= perform_task(x, h, handles)
    if get(h, 'value') == 1
        zoom off
        if get(handles.enter_checkbox, 'value') == 1 % enter points
            [xn,yn,bn]= ginput(1);
            while ~isempty(xn) & bn == 1
                % perform task
            end
        end
    end
x = [x [xn;yn]];  
if h == handles.waypoint_checkbox  
  plotwaypoints(x);  
else  
  plotlandmarks(x);  
end  
[xn,yn,bn]= ginput(1);  
end  
else  
i= find_nearest(x);  
if get(handles.delete_checkbox, 'value') == 1 % delete nearest point  
x= [x(:,1:i-1) x(:,i+1:end)];  
endif get(handles.move_checkbox, 'value') == 1 % move nearest point  
x(:,i)= ginput(1)';  
plot(xt(1), xt(2),'wx', 'markersize',10)  
end  
end  
set(h, 'value', 0)  
end

% init file  
% -------------------------------------------------------------------  
% flags
USE_ODOMETRY_ERROR = 1;     % if 1, odometry noise is added  
USE_SLAM_CORRECTION = 1; % if 1, use EKF updater  
DRAW = 1;             % 1 if redraw, 0 if not  
USE_GRAPHICS = 1;       % 1 if draw graphics, 0 if just output

% Initialize states and other global variables  
global X P  
xtrue = zeros(3,1); % true position  
xuncorrected = zeros(3,1); % uncorrected position  
X = zeros(3,1); % X matrix  
P = zeros(3); % P matrix  
iwp = 1; % index to first waypoint  
count = 1; % count for errors

% control parameters  
V = 0.127; % m/s

% control noises  
q_min = .01379412113; % percentage of min distance error  
q_max = .03745367509; % percentage of max distance error  
if (USE_ODOMETRY_ERROR == 1)  
c = 0.02; % noise guess for predict step  
else  
c = 0;  
end

% data association innovation gates (Mahalanobis distances)  
GATE_REJECT = 4.0; % maximum distance for association
GATE_AUGMENT = 25.0; % minimum distance for creation of new feature
AT_WAYPOINT = 0.05; % meters, distance from current waypoint at which to
switch to next waypoint

% other variables
WHEELBASE = 0.1905; % meters, vehicle wheel base

function [z, sensor_table, landmarks, number_found] =
ir_landmark_extractor(lm, sensor_data, vehicle_state, MAX_RANGE, MIN_RANGE,
SENSORS, LANDMARK_COUNT, sensor_table, WHEELBASE)

count = 1;
landmarks = zeros(2,1); % in case no landmarks are found
z = zeros(2,1);
n = length(SENSORS);

for i = 1:n
    if (sensor_data(i) > MIN_RANGE && ...
        sensor_data(i) < MAX_RANGE && ...
        abs(sensor_data(i) - sensor_table(1,i)) > 0.2)
        theta_o = SENSORS(i); % angle from robot perspective
        theta_r = vehicle_state(3, 1) + WHEELBASE/2; % vehicle angle in
        global view
        z(1:2,count) = [sensor_data(i); pi_to_pi(theta_o)];
        x = sensor_data(i) * cos(theta_o + theta_r); % local x to robot
        y = sensor_data(i) * sin(theta_o + theta_r); % local y to robot
        x = x + vehicle_state(1, 1); % adjust to global x
        y = y + vehicle_state(2, 1); % adjust to global y
        landmarks(:, count) = [x; y]; % add to observed landmark vector
        count = count + 1;
    end
end

for i = 1:n
    sensor_table(1,i) = sensor_data(i);
end

number_found = count - 1;

function [sensor_data] = ir_sensor_driver(lm, xtrue, max_range, min_range,
SENSORS, WHEELBASE, noise, sigmaR)

% initialize sensor vector
for i = 1:length(SENSORS)
    sensor_data(i) = 0;
end

% get observations
for i = 1:size(lm,2)
    dx = lm(1,i) - xtrue(1); % delta x
    dy = lm(2,i) - xtrue(2); % delta y
    phi = find_angle(dx, dy, xtrue(3));
    d = sqrt(dx^2 + dy^2) - WHEELBASE/2; % distance from robot center to
    landmark
if (d < max_range && d > min_range) % find landmarks within the max and min circles
    for j = 1:length(SENSORS)
        if (check_view_angle(phi, SENSORS(j), pi/180) == 1) % find landmarks within angle ranges of a sensor
            if (sensor_data(j) > d || sensor_data(j) == 0) % only adjust the distance if it is a shorter distance away
                sensor_data(j) = d;
            end
        end
    end
end

% add noise to sensor readings
for i = 1:length(sensor_data)
    if (noise == 1)
        sensor_data(i) = sensor_data(i) + (sensor_data(i)/max_range)\(\text{rand}(1) - 0.5)\*2\*sigmaR; % noise is related to the distance of reading
    elseif (noise == 2)
        sensor_data(i) = sensor_data(i) + (rand(1) - 0.5)\*2\*sigmaR; % noise is direct, not related to distance
    end
end

function KF_cholesky_update(v, R, H)
% Tim Bailey 2003
% Adapted from code by Jose Guivant
global X P
PHt = (H*P)'; % Matlab is column-major, so (H*PX)' is more efficient than PX*H' [Tim 2004]
S = H*PHt + R;
S = (S + S')*0.5; % ensure is symmetric
SChol = chol(S);
SCholInv = inv(SChol); % triangular matrix
W1 = PHt * SCholInv;
W = W1 * SCholInv';
X = X + W\*v; % update
P = P - W1\*W1';

function [sensor_data] = sensor_driver(lm, xtrue, scan_width, angle_resolution, max_range, min_range, sigmaR, noise)

n = ceil(scan_width/angle_resolution);

% initialize sensor vector
for i = 1:n
    sensor_data(i) = 0;
end

% get observations
for i = 1:size(lm,2)
    dx = lm(1,i) - xtrue(1); % delta x
    dy = lm(2,i) - xtrue(2); % delta y
    phi = find_angle(dx, dy, xtrue(3));
    d = sqrt(dx^2 + dy^2); % distance from robot center to landmark
    if (d < max_range && d > min_range) % find landmarks within the max and min
        if (check_view_angle(phi, 0, scan_width*pi/180) == 1) % find landmarks within angle ranges of a sensor
            j = ceil((phi*180/pi + scan_width/2)/angle_resolution);
            if (sensor_data(j) > d || sensor_data(j) == 0) % only adjust the distance if it is a shorter distance away
                sensor_data(j) = d;
            end
        end
    end
end

% add noise to sensor readings
for i = 1:length(sensor_data)
    if (noise == 1)
        sensor_data(i) = sensor_data(i) + (sensor_data(i)/max_range)*(rand(1) - 0.5)*2*sigmaR; % noise is related to the distance of reading
    elseif (noise == 2)
        sensor_data(i) = sensor_data(i) + (rand(1) - 0.5)*2*sigmaR; % noise is direct, not related to distance
    end
end

sensor_data = fliplr(sensor_data);

function [z,H]= observe_model(x, idf)
% Tim Bailey 2004.

Nxv= 3; % number of vehicle pose states
fpos= Nxv + idf*2 - 1; % position of xf in state
H= zeros(2, length(x));

% auxiliary values
dx= x(fpos) -x(1);
dy= x(fpos+1)-x(2);
d2= dx^2 + dy^2;
d= sqrt(d2);
xd= dx/d;
yd= dy/d;
xd2= dx/d2;
yd2= dy/d2;

% predict z
z= [d;
    atan2(dy,dx) - x(3)];

% calculate H
H(:,1:3) = [-xd -yd 0; yd2 -xd2 -1];
H(:,fpos:fpos+1)= [ xd yd; -yd2 xd2];
function [lwdist_error, rwdist_error, actual_state, xuncorrected, iwp] = odometry_driver(wp, observed_state, actual_state, xuncorrected, iwp, dt, wpdist, v, USE_ODOMETRY_ERROR, q_max, q_min, WHEELBASE)

% determine if current waypoint reached
w = wp(:,iwp); % current waypoint
d2 = sqrt((cwp(1) - actual_state(1))^2 + (cwp(2) - actual_state(2))^2);
if (d2 < wpdist)
    iwp = iwp + 1; % switch to next
    if (iwp > size(wp,2)) % reached final waypoint, flag and return
        iwp = 0;
        lwdist_error = 0;
        rwdist_error = 0;
        return;
    end
    cwp = wp(:,iwp); % next waypoint
end

% calculate new steering
angle = pi_to_pi(atan2(cwp(2) - actual_state(2), cwp(1) - actual_state(1)) - actual_state(3));

% calculate new l/r wheel speeds
if (angle > 0.001)
    rspd = v;
    lspd = v/(1 + abs(angle)); % slow the left wheel
elseif (angle < -0.001)
    rspd = v/(1 + abs(angle)); % slow the right wheel
    lspd = v;
else
    rspd = v; % both wheels at max speed
    lspd = v;
end

% set lwdist and rwdist
lwdist = lspd*dt;
rwdist = rspd*dt;

% introduce control noise
if (USE_ODOMETRY_ERROR == 1)
    lwdist_error = lwdist - (rand(1)*q_max + q_min)*lwdist; % uniform distribution based on distance travelled and max error
    rwdist_error = rwdist - (rand(1)*q_max + q_min)*rwdist;
else
    lwdist_error = lwdist;
    rwdist_error = rwdist;
end

% calculate actual state
linear_displacement = (rwdist + lwdist)/2;
angular_displacement = pi_to_pi((rwdist - lwdist)/WHEELBASE);
f = [actual_state(1) + linear_displacement*cos(angular_displacement + actual_state(3))];
actual_state(2) + linear_displacement*\sin(\text{angular_displacement} + 
actual_state(3));
pi_to_pi(actual_state(3) + angular_displacement)];
actual_state = f;

% setup uncorrected
linear_displacement = (rwdist_error + lwdist_error)/2;
angular_displacement = pi_to_pi((rwdist_error - lwdist_error)/WHEELBASE);
theta = pi_to_pi(xuncorrected(3,1) + angular_displacement);
deltax = linear_displacement*\cos(theta);
deltay = linear_displacement*\sin(theta);
f = [xuncorrected(1,1) + deltax; % f function
    xuncorrected(2,1) + deltay;
    theta];
xuncorrected = f;

function angle = pi_to_pi(angle)
% Tim Bailey 2000

% Avoid mod() as angles are (nearly) always bounded by -2*pi <= angle <= 2*pi
% and, also, the mod() function is very slow. [Tim Bailey 2004]
i= find(angle<-2*pi | angle>2*pi); % replace with a check
if ~isempty(i)
    warning('pi_to_pi() error: angle outside 2-PI bounds.')</n    angle(i) = mod(angle(i), 2*pi);
end

i= find(angle>pi);
angle(i)= angle(i)-2*pi;

i= find(angle<-pi);
angle(i)= angle(i)+2*pi;

function [] = predict(lwdist, rwdist, c, dist_bt_wheels, dt)

global X P

linear_displacement = (rwdist + lwdist)/2;
angular_displacement = pi_to_pi((rwdist - lwdist)/dist_bt_wheels);
theta = pi_to_pi(X(3,1) + angular_displacement);
deltax = linear_displacement*\cos(theta);
deltay = linear_displacement*\sin(theta);
f = [X(1,1) + deltax; % f function
    X(2,1) + deltay;
    theta];
A = [1 0 -deltay; % Jacobian of f
    0 1 deltax;
    0 0 1];
Q = [c*deltax^2 c*deltax*deltay c*deltax*dt; % Q noise matrix
    c*deltay*deltax c*deltay^2 c*deltay*dt;
    c*dt*deltax c*dt*deltay c*dt^2];

X(1:3,1) = f;
P(1:3,1:3) = A*P(1:3,1:3)*A' + Q;
if (size(P,1) > 3)
P(1:3,4:end) = A*P(1:3,4:end);
P(4:end,1:3) = P(1:3,4:end)';
end

function [max_error, average_error, final_error, time] = run_multiple(wp, ln, loops)
count = 1;
for i = 1:loops
tic
    error = simulator(wp, ln);
    n = length(error);
    max_error(count,1) = max(error(:,1));
    min_error(count,1) = min(error(:,1));
    average_error(count,1) = mean(error(:,1));
    final_error(count,1) = error(n,1);
    max_error(count,2) = max(error(:,2));
    average_error(count,2) = mean(error(:,2));
    final_error(count,2) = error(n,2);
    end_time = toc;
    time(count,1) = end_time;
    remaining = loops - count
    remaining_time = remaining*end_time
    count = count + 1;
end

function [z, landmark_table, sensor_table, state_table, observed_landmarks, landmarks_found] = sonar_landmark_extractor(lm, sensors, state, lwdist, rwdist, MAX_RANGE, MIN_RANGE, SENSORS, VIEW_ANGLE, LANDMARK_COUNT, landmark_table, sensor_table, state_table, WHEELBASE)
% initialize return variables
observed_landmarks = [];
landmarks_found = 0;
z = [];

threshold = 0.015;
threshold2 = 0.08;
threshold3 = 0.1;
min = MIN_RANGE - threshold2;
max = MAX_RANGE - threshold2;
linear_displacement = (rwdist + lwdist)/2;
angular_displacement = pi_to_pi((rwdist - lwdist)/WHEELBASE);

if (mod(length(SENSORS), 2) == 1) % if odd number of sensors, there is a
    mid_sensor = ceil((length(SENSORS))/2);
else % even number of sensors
    mid_sensor = length(SENSORS)/2;
end

for i = 1:length(SENSORS)
    current = sensors(1,i);
    previous = sensor_table(LANDMARK_COUNT,i);
    previous_state = state_table(LANDMARK_COUNT,:);
    if (previous == 0)
        break;
    end
x = landmark_table(1,i);
y = landmark_table(2,i);
type = landmark_table(3,i);
found = 0;
if (i < mid_sensor) % sensor is on left side
    if (angular_displacement < 0 && linear_displacement > 0) % vehicle moving forward
        if (within_range(current, min, max) == 1 && previous - current > threshold2)
            % object just appeared in sensor MUST be from the right
            r = current + WHEELBASE/2;
b = (SENSORS(i) - (VIEW_ANGLE)/2);
            landmarks_found = landmarks_found + 1;
z(1:2,landmarks_found) = [r; b];
x = r * cos(b + state(3)); % local x to robot
y = r * sin(b + state(3)); % local y to robot
x = x + state(1); % adjust to global x
y = y + state(2); % adjust to global y
            landmark_table(1:3,i) = [x; y; 1];
            found = 1;
            if (check_landmarks(x, y, lm) == 0)
                x;
            end
        elseif (current - previous > threshold3)
            % object just disappeared from sensor
            r = previous + WHEELBASE/2;
b = (SENSORS(i) + (VIEW_ANGLE)/2);
x = r * cos(b + previous_state(3)); % local x to robot
y = r * sin(b + previous_state(3)); % local y to robot
x = x + previous_state(1); % adjust to global x
y = y + previous_state(2); % adjust to global y
            landmark_table(1:3,i) = [x; y; 2];
            found = 2;
            if (check_landmarks(x, y, lm) == 0)
                x;
            end
        end
    elseif (i > mid_sensor) % sensor is on right side
        if (angular_displacement > 0 && linear_displacement > 0) % vehicle moving forward
            if (within_range(current, min, max) == 1 && previous - current > threshold2)
                % object just appeared in sensor MUST be from the left
                r = current + WHEELBASE/2;
b = (SENSORS(i) + (VIEW_ANGLE)/2);
                landmarks_found = landmarks_found + 1;
z(1:2,landmarks_found) = [r; b];
x = r * cos(b + state(3)); % local x to robot
y = r * sin(b + state(3)); % local y to robot
x = x + state(1); % adjust to global x
y = y + state(2); % adjust to global y
                landmark_table(1:3,i) = [x; y; 1];
                found = 1;
                if (check_landmarks(x, y, lm) == 0)
                    x;
                end
            end
        end
    end
end

130
elseif (current - previous > threshold3)
    % object just disappeared from sensor
    r = previous + WHEELBASE/2;
    b = (SENSORS(i) - (VIEW_ANGLE)/2);
    x = r * cos(b + previous_state(3)); % local x to robot
    y = r * sin(b + previous_state(3)); % local y to robot
    x = x + previous_state(1); % adjust to global x
    y = y + previous_state(2); % adjust to global y
    landmark_table(1:3,i) = [x; y; 2];
    found = 2;
    if (check_landmarks(x, y, lm) == 0)
        %
    end
end
end
if (found == 0 && i > 1 && i < length(SENSORS)) % object still not found,
use poorer method
    cur_R = sensors(1,i+1);
    prev_R = sensor_table(LANDMARK_COUNT,i+1);
    cur_L = sensors(1,i-1);
    prev_L = sensor_table(LANDMARK_COUNT,i-1);
    result = find_edges(current, cur_L, cur_R, previous, prev_L, prev_R,
min, max, threshold, Threshold2, Threshold3);
    if (result == 1) % came from left
        r = current + WHEELBASE/2;
        b = (SENSORS(i) + (VIEW_ANGLE)/2);
        landmarks_found = landmarks_found + 1;
        z(1:2,landmarks_found) = [r; b];
        x = r * cos(b + state(3)); % local x to robot
        y = r * sin(b + state(3)); % local y to robot
        x = x + state(1); % adjust to global x
        y = y + state(2); % adjust to global y
        landmark_table(1:3,i) = [x; y; 1];
        found = 1;
        if (check_landmarks(x, y, lm) == 0)
        %
    elseif (result == 2) % came from right
        r = current + WHEELBASE/2;
        b = (SENSORS(i) - (VIEW_ANGLE)/2);
        landmarks_found = landmarks_found + 1;
        z(1:2,landmarks_found) = [r; b];
        x = r * cos(b + state(3)); % local x to robot
        y = r * sin(b + state(3)); % local y to robot
        x = x + state(1); % adjust to global x
        y = y + state(2); % adjust to global y
        landmark_table(1:3,i) = [x; y; 1];
        found = 1;
        if (check_landmarks(x, y, lm) == 0)
        %
    elseif (result == 3) % went to left
    % object just disappeared from sensor
    r = previous + WHEELBASE/2;
    b = (SENSORS(i) + (VIEW_ANGLE)/2);
    x = r * cos(b + previous_state(3)); % local x to robot
\begin{verbatim}
\textbf{y} = r \cdot \sin(b + \text{previous\_state}(3)); \% local y to robot \\
\textbf{x} = x + \text{previous\_state}(1); \% adjust to global x \\
\textbf{y} = y + \text{previous\_state}(2); \% adjust to global y \\
\text{landmark\_table}(1:3,i) = [x; y; 2]; \\
\text{found} = 2; \\
\text{if} (\text{check\_landmarks}(x, y, lm) == 0) \\
\text{x}; \\
\text{end}
\end{verbatim}

\textbf{elseif} (\text{result} == 4) \% went to right \\
\text{r} = \text{previous} + \text{WHEELBASE}/2; \\
\text{b} = (\text{SENSORS}(i) - (\text{VIEW\_ANGLE})/2); \\
\textbf{x} = r \cdot \cos(b + \text{previous\_state}(3)); \% local x to robot \\
\textbf{y} = r \cdot \sin(b + \text{previous\_state}(3)); \% local y to robot \\
\textbf{x} = x + \text{previous\_state}(1); \% adjust to global x \\
\textbf{y} = y + \text{previous\_state}(2); \% adjust to global y \\
\text{landmark\_table}(1:3,i) = [x; y; 2]; \\
\text{found} = 2; \\
\text{if} (\text{check\_landmarks}(x, y, lm) == 0) \\
\text{x}; \\
\text{end}
\end{verbatim}

\textbf{end}

\textbf{if} (\text{within\_range}(\text{current}, \text{min}, \text{max}) == 1 && \text{within\_range}(\text{previous}, \text{min}, \text{max}) == 1) \% if object was in sensor last time \\
\text{\% make sure it is the same object} \\
\text{\% ----------------- this is unstable and NOT USED -------------} \\
\text{\% if (abs(\text{current} - \text{previous}) < \text{threshold}) && type == 1 && 0) \\
\text{r} = \text{current} + \text{WHEELBASE}/2; \\
\text{dx} = x - \text{state}(1); \\
\text{dy} = y - \text{state}(2); \\
\text{b} = \text{find\_angle(dx, dy, state(3));} \\
\text{landmarks\_found} = \text{landmarks\_found} + 1; \\
\text{z}(1:2,\text{landmarks\_found}) = [r; b]; \\
\text{landmark\_table}(1:3,i) = [x; y; 1]; \\
\text{found} = 1; \\
\text{\% debug} \\
\text{xx} = r \cdot \cos(b + \text{state}(3)); \% local x to robot \\
\text{yy} = r \cdot \sin(b + \text{state}(3)); \% local y to robot \\
\text{xx} = xx + \text{state}(1); \% adjust to global x \\
\text{yy} = yy + \text{state}(2); \% adjust to global y \\
\text{if} (\text{check\_landmarks}(xx, yy, lm) == 0) \\
\text{x}; \\
\text{end}
\end{verbatim}

\textbf{end}

\textbf{end}

\textbf{if} (\text{found} == 0) \\
\text{\text{landmark\_table}(1:3,i) = [0; 0; 0];} \\
\text{end}
\end{verbatim}

\textbf{for} i = 1:length(\text{SENSORS}) \% check to make sure there are no duplicates \\
\textbf{if} (\text{landmark\_table}(3,i) == 2) \\
\textbf{for} j = 1:length(\text{SENSORS}) \\
\textbf{\% only compare opposites} \\
\text{x1} = \text{landmark\_table}(1,i); \\
\text{x2} = \text{landmark\_table}(1,j);
y1 = landmark_table(2,i);
y2 = landmark_table(2,j);
if (sqrt((x1 - x2)^2 + (y1 - y2)^2) < threshold2)
    % remove duplicate
    landmark_table(1:3,i) = [0; 0; 0];
end
end
end
end
for i = 1:length(SENSORS) % add non-duplicates
    if (landmark_table(3,i) == 2)
        x = landmark_table(1,i);
        y = landmark_table(2,i);
        dx = x - state(1);
        dy = y - state(2);
        r = sqrt(dx^2 + dy^2);
        b = find_angle(dx, dy, state(3));
        landmarks_found = landmarks_found + 1;
        z(1:2,landmarks_found) = [r; b];
    end
end
end
for i = 2:LANDMARK_COUNT
    sensor_table(i-1,:) = sensor_table(i,:);
    state_table(i-1,:) = state_table(i,:);
end
sensor_table(LANDMARK_COUNT,:) = sensors;
state_table(LANDMARK_COUNT,:) = state;

function [result] = find_edges(cm, cl, cr, pm, pl, pr, min, max, threshold, threshold2, threshold3)
    result = 0;
    if (within_range(cm, min, max) == 1 && pm - cm > threshold2) % object just appeared in sensor
        diffL = abs(pl - cm);
        diffR = abs(pr - cm);
        if (diffL < threshold && diffR > threshold) % object came from left
            result = 1;
        elseif (diffL > threshold && diffR < threshold) % object came from right
            result = 2;
        end
    elseif (cm - pm > threshold3) % object disappeared from sensors
        diffL = abs(cl - pm);
        diffR = abs(cr - pm);
        if (diffL < threshold && diffR > threshold) % object went to left
            result = 3;
        elseif (diffL > threshold && diffR < threshold) % object went to right
            result = 4;
        end
    end

    function [result] = within_range(i, min, max)
        result = 0;
        if (i > min && i < max)
            result = 1;
        end
end
function [result] = check_landmarks(x, y, lm)
    threshold = 0.2;
    result = 0;
    for i = 1:size(lm,2)
        d = sqrt((lm(1,i) - x)^2 + (lm(2,i) - y)^2);
        if (d < threshold)
            result = 1;
            break;
        end
    end
end

function [sensor_data] = sonar_sensor_driver(lm, xtrue, MAX_RANGE, MIN_RANGE, VIEW_ANGLE, IR_SENSORS, WHEELBASE, noise, sigmaR)
    % initialize sensor vector
    for i = 1:length(IR_SENSORS)
        sensor_data(i) = MAX_RANGE + 1;
    end

    % get observations
    for i = 1:size(lm,2)
        dx = lm(1,i) - xtrue(1); % delta x
        dy = lm(2,i) - xtrue(2); % delta y
        phi = find_angle(dx, dy, xtrue(3));
        d = sqrt(dx^2 + dy^2) - WHEELBASE/2; % distance from robot center to landmark
        if (d < MAX_RANGE && d > MIN_RANGE) % find landmarks within the max and min circles
            for j = 1:length(IR_SENSORS)
                if (check_view_angle(phi, IR_SENSORS(j), VIEW_ANGLE) == 1) % find landmarks within angle ranges of a sensor
                    if (sensor_data(j) > d) % only adjust the distance if it is a shorter distance away
                        sensor_data(j) = d;
                    end
                end
            end
        end
    end

    % add noise to sensor readings
    for i = 1:length(sensor_data)
        if (noise == 1)
            sensor_data(i) = sensor_data(i) + (sensor_data(i)/MAX_RANGE)*(rand(1) - 0.5)*2*sigmaR; % noise is related to the distance of reading
        elseif (noise == 2)
            sensor_data(i) = sensor_data(i) + (rand(1) - 0.5)*2*sigmaR; % noise is direct, not related to distance
        end
    end

function [z, landmarks, number_found] = spikes_landmark_extractor(sensor_data, vehicle_state, EXTRACTION_THRESHOLD, MAX_RANGE, MIN_RANGE, scan_width, angle_resolution)
count = 1;

landmarks = zeros(2,1); % in case no landmarks are found
z = zeros(2,1);
n = ceil(scan_width/angle_resolution);

for i = 2:(n-1)
    if (sensor_data(i-1) < MIN_RANGE && ...
        sensor_data(i+1) < MIN_RANGE && ...
        sensor_data(i) < MAX_RANGE && sensor_data(i) > EXTRACTION_THRESHOLD)
        theta_o = ((i) - ceil(n/2))*angle_resolution*pi/180; % angle from robot perspective
        theta_r = vehicle_state(3, 1); % vehicle angle in global view
        z(1:2,count) = [sensor_data(i); pi_to_pi(-theta_o)];
        x = sensor_data(i) * cos(-theta_o + theta_r); % local x to robot
        y = sensor_data(i) * sin(-theta_o + theta_r); % local y to robot
        x = x + vehicle_state(1, 1); % adjust to global x
        y = y + vehicle_state(2, 1); % adjust to global y
        landmarks(:, count) = [x; y]; % add to observed landmark vector
        count = count + 1;
    end
end

number_found = count - 1;

function X = sqrtm_2by2(A)
%SQRTM Matrix square root.
%   X = SQRTM_2by2(A) is the principal square root of the matrix A, i.e. X*X = A.
%
%   X is the unique square root for which every eigenvalue has nonnegative
%   real part. If A has any eigenvalues with negative real parts then a
%   complex result is produced. If A is singular then A may not have a
%   square root. A warning is printed if exact singularity is detected.
%
% Adapted for speed for 2x2 matrices from the MathWorks sqrtm.m implementation.
% Tim Bailey 2004.

[Q, T] = schur(A);        % T is real/complex according to A.
%[Q, T] = rsf2csf(Q, T);   % T is now complex Schur form.
R = zeros(2);
R(1,1) = sqrt(T(1,1));
R(2,2) = sqrt(T(2,2));
R(1,2) = T(1,2) / (R(1,1) + R(2,2));
X = Q*R*Q';

function update(z, R, idf)

global X
lenz = size(z,2);
lenx = length(X);
H = zeros(2*lenz, lenx);
v = zeros(2*lenz, 1);
RR = zeros(2*lenz);

for i=1:lenz
    ii = 2*i + (-1:0);
    [zp,H(ii,:)] = observe_model(X, idf(i));
    v(ii) = [z(1,i)-zp(1);
             pi_to_pi(z(2,i)-zp(2))];
    RR(ii,ii) = R;
end

KF_cholesky_update(v, RR, H);
APPENDIX 2

Mahalanobis Distance Results from Environment 1:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Cost</th>
<th>Corrected</th>
<th>Uncorrected</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ave</td>
<td>Max</td>
<td>Fin</td>
<td>Ave</td>
</tr>
<tr>
<td>URG-04LX</td>
<td>$2,375</td>
<td>13.7</td>
<td>47.2</td>
<td>7.8</td>
</tr>
<tr>
<td>URG-04LX-UG01</td>
<td>$1,175</td>
<td>10.1</td>
<td>21.4</td>
<td>5.4</td>
</tr>
<tr>
<td>UTM-30LX</td>
<td>$5,590</td>
<td>31.7</td>
<td>86.8</td>
<td>9.0</td>
</tr>
<tr>
<td>RevoLDS</td>
<td>$30</td>
<td>13.7</td>
<td>47.2</td>
<td>7.9</td>
</tr>
<tr>
<td>SRF05-3</td>
<td>$90</td>
<td>13.7</td>
<td>47.0</td>
<td>7.9</td>
</tr>
<tr>
<td>SRF05-5</td>
<td>$150</td>
<td>13.7</td>
<td>47.2</td>
<td>7.9</td>
</tr>
<tr>
<td>SRF05-7</td>
<td>$210</td>
<td>13.6</td>
<td>46.8</td>
<td>7.9</td>
</tr>
<tr>
<td>EZI-3</td>
<td>$75</td>
<td>13.7</td>
<td>47.1</td>
<td>7.9</td>
</tr>
<tr>
<td>EZI-5</td>
<td>$125</td>
<td>13.7</td>
<td>47.3</td>
<td>7.8</td>
</tr>
<tr>
<td>EZI-7</td>
<td>$175</td>
<td>13.7</td>
<td>47.0</td>
<td>7.9</td>
</tr>
<tr>
<td>IR-3</td>
<td>$38</td>
<td>93.2</td>
<td>287.1</td>
<td>102.8</td>
</tr>
<tr>
<td>IR-5</td>
<td>$63</td>
<td>38.7</td>
<td>88.5</td>
<td>27.1</td>
</tr>
<tr>
<td>IR-7</td>
<td>$88</td>
<td>10.7</td>
<td>33.0</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Mahalanobis Distance Results from Environment 2:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Cost</th>
<th>Corrected</th>
<th>Uncorrected</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ave</td>
<td>Max</td>
<td>Fin</td>
<td>Ave</td>
</tr>
<tr>
<td>URG-04LX</td>
<td>$2,375</td>
<td>18.3</td>
<td>64.5</td>
<td>7.1</td>
</tr>
<tr>
<td>URG-04LX-UG01</td>
<td>$1,175</td>
<td>15.2</td>
<td>49.4</td>
<td>3.2</td>
</tr>
<tr>
<td>UTM-30LX</td>
<td>$5,590</td>
<td>15.3</td>
<td>49.4</td>
<td>3.1</td>
</tr>
<tr>
<td>RevoLDS</td>
<td>$30</td>
<td>61.6</td>
<td>113.6</td>
<td>8.5</td>
</tr>
<tr>
<td>SRF05-3</td>
<td>$90</td>
<td>39.2</td>
<td>129.6</td>
<td>30.2</td>
</tr>
<tr>
<td>SRF05-5</td>
<td>$150</td>
<td>19.0</td>
<td>52.4</td>
<td>8.8</td>
</tr>
<tr>
<td>SRF05-7</td>
<td>$210</td>
<td>24.4</td>
<td>69.6</td>
<td>13.4</td>
</tr>
<tr>
<td>EZI-3</td>
<td>$75</td>
<td>101.2</td>
<td>272.5</td>
<td>101.0</td>
</tr>
<tr>
<td>EZI-5</td>
<td>$125</td>
<td>43.1</td>
<td>104.2</td>
<td>31.4</td>
</tr>
<tr>
<td>EZI-7</td>
<td>$175</td>
<td>28.7</td>
<td>86.3</td>
<td>16.9</td>
</tr>
<tr>
<td>IR-3</td>
<td>$38</td>
<td>64.7</td>
<td>188.6</td>
<td>85.7</td>
</tr>
<tr>
<td>IR-5</td>
<td>$63</td>
<td>124.1</td>
<td>444.0</td>
<td>187.3</td>
</tr>
<tr>
<td>IR-7</td>
<td>$88</td>
<td>86.8</td>
<td>228.1</td>
<td>79.0</td>
</tr>
</tbody>
</table>
### Mahalanobis Distance Results from Environment 3:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Corrected</th>
<th>Uncorrected</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Ave</td>
<td>Max</td>
</tr>
<tr>
<td>URG-04LX</td>
<td>$2,375</td>
<td>10.8</td>
<td>23.4</td>
</tr>
<tr>
<td>URG-04LX-UG01</td>
<td>$1,175</td>
<td>10.9</td>
<td>31.9</td>
</tr>
<tr>
<td>UTM-30LX</td>
<td>$5,590</td>
<td>10.0</td>
<td>23.1</td>
</tr>
<tr>
<td>RevoloDS</td>
<td>$30</td>
<td>4.6</td>
<td>16.0</td>
</tr>
<tr>
<td>SRF05-3</td>
<td>$90</td>
<td>9.5</td>
<td>41.0</td>
</tr>
<tr>
<td>SRF05-5</td>
<td>$150</td>
<td>4.4</td>
<td>16.1</td>
</tr>
<tr>
<td>SRF05-7</td>
<td>$210</td>
<td>4.8</td>
<td>10.3</td>
</tr>
<tr>
<td>EZI-3</td>
<td>$75</td>
<td>25.4</td>
<td>106.6</td>
</tr>
<tr>
<td>EZI-5</td>
<td>$125</td>
<td>22.7</td>
<td>62.4</td>
</tr>
<tr>
<td>IR-3</td>
<td>$175</td>
<td>13.5</td>
<td>35.8</td>
</tr>
<tr>
<td>IR-5</td>
<td>$338</td>
<td>18.5</td>
<td>46.5</td>
</tr>
<tr>
<td>IR-7</td>
<td>$863</td>
<td>18.7</td>
<td>44.4</td>
</tr>
<tr>
<td>IR-7</td>
<td>$888</td>
<td>25.5</td>
<td>57.6</td>
</tr>
</tbody>
</table>

### Mahalanobis Distance Results from Environment 4:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Corrected</th>
<th>Uncorrected</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Ave</td>
<td>Max</td>
</tr>
<tr>
<td>URG-04LX</td>
<td>$2,375</td>
<td>21.5</td>
<td>45.9</td>
</tr>
<tr>
<td>URG-04LX-UG01</td>
<td>$1,175</td>
<td>17.8</td>
<td>40.3</td>
</tr>
<tr>
<td>UTM-30LX</td>
<td>$5,590</td>
<td>9.4</td>
<td>18.9</td>
</tr>
<tr>
<td>RevoloDS</td>
<td>$30</td>
<td>16.3</td>
<td>37.6</td>
</tr>
<tr>
<td>SRF05-3</td>
<td>$90</td>
<td>32.0</td>
<td>79.6</td>
</tr>
<tr>
<td>SRF05-5</td>
<td>$150</td>
<td>13.8</td>
<td>35.3</td>
</tr>
<tr>
<td>SRF05-7</td>
<td>$210</td>
<td>8.1</td>
<td>21.4</td>
</tr>
<tr>
<td>EZI-3</td>
<td>$75</td>
<td>31.7</td>
<td>74.1</td>
</tr>
<tr>
<td>EZI-5</td>
<td>$125</td>
<td>15.5</td>
<td>38.2</td>
</tr>
<tr>
<td>EZI-7</td>
<td>$175</td>
<td>8.1</td>
<td>20.7</td>
</tr>
<tr>
<td>IR-3</td>
<td>$338</td>
<td>72.9</td>
<td>201.5</td>
</tr>
<tr>
<td>IR-5</td>
<td>$63</td>
<td>34.7</td>
<td>81.4</td>
</tr>
<tr>
<td>IR-7</td>
<td>$888</td>
<td>33.2</td>
<td>86.9</td>
</tr>
</tbody>
</table>

### Euclidean Distance Results from Environment 1:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Corrected</th>
<th>Uncorrected</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cost</td>
<td>Ave</td>
<td>Max</td>
</tr>
<tr>
<td>RevoloDS</td>
<td>$30</td>
<td>38.7</td>
<td>80.7</td>
</tr>
<tr>
<td>SRF05-3</td>
<td>$90</td>
<td>47.2</td>
<td>111.7</td>
</tr>
<tr>
<td>SRF05-5</td>
<td>$150</td>
<td>11.3</td>
<td>28.1</td>
</tr>
<tr>
<td>SRF05-7</td>
<td>$210</td>
<td>9.5</td>
<td>23.5</td>
</tr>
<tr>
<td>EZI-3</td>
<td>$75</td>
<td>38.1</td>
<td>89.2</td>
</tr>
<tr>
<td>EZI-5</td>
<td>$125</td>
<td>13.0</td>
<td>29.2</td>
</tr>
<tr>
<td>EZI-7</td>
<td>$175</td>
<td>12.6</td>
<td>36.9</td>
</tr>
</tbody>
</table>
Euclidean Distance Results from Environment 2:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Cost</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Improve/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RevoLDS</td>
<td>$30</td>
<td>64.3</td>
<td>118.5</td>
<td>9.7</td>
<td>144.1</td>
<td>380.1</td>
<td>53.8</td>
<td>55.4%</td>
<td>68.8%</td>
<td>82.0%</td>
<td>2.2918%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-3</td>
<td>$90</td>
<td>77.8</td>
<td>206.6</td>
<td>47.6</td>
<td>144.3</td>
<td>377.1</td>
<td>55.0</td>
<td>46.1%</td>
<td>45.2%</td>
<td>13.4%</td>
<td>0.3877%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-5</td>
<td>$150</td>
<td>18.0</td>
<td>51.6</td>
<td>5.8</td>
<td>147.0</td>
<td>382.3</td>
<td>53.3</td>
<td>87.8%</td>
<td>86.5%</td>
<td>89.1%</td>
<td>0.5853%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-7</td>
<td>$210</td>
<td>40.5</td>
<td>100.0</td>
<td>26.4</td>
<td>145.9</td>
<td>379.1</td>
<td>55.0</td>
<td>72.3%</td>
<td>73.6%</td>
<td>52.0%</td>
<td>0.3141%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-3</td>
<td>$75</td>
<td>83.0</td>
<td>224.7</td>
<td>50.4</td>
<td>146.0</td>
<td>380.5</td>
<td>54.3</td>
<td>43.1%</td>
<td>40.9%</td>
<td>7.2%</td>
<td>0.4056%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-5</td>
<td>$125</td>
<td>24.7</td>
<td>59.1</td>
<td>8.5</td>
<td>143.4</td>
<td>375.8</td>
<td>56.0</td>
<td>82.8%</td>
<td>84.3%</td>
<td>84.8%</td>
<td>0.4056%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-7</td>
<td>$175</td>
<td>33.2</td>
<td>97.2</td>
<td>12.8</td>
<td>146.0</td>
<td>379.6</td>
<td>51.2</td>
<td>77.2%</td>
<td>74.4%</td>
<td>75.0%</td>
<td>0.4316%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Euclidean Distance Results from Environment 3:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Cost</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Improve/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RevoLDS</td>
<td>$30</td>
<td>5.5</td>
<td>22.7</td>
<td>3.1</td>
<td>39.5</td>
<td>114.3</td>
<td>67.9</td>
<td>86.2%</td>
<td>80.1%</td>
<td>95.4%</td>
<td>2.9079%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-3</td>
<td>$90</td>
<td>16.0</td>
<td>49.0</td>
<td>21.4</td>
<td>39.8</td>
<td>114.3</td>
<td>68.0</td>
<td>59.7%</td>
<td>57.1%</td>
<td>68.6%</td>
<td>0.6867%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-5</td>
<td>$150</td>
<td>26.5</td>
<td>133.0</td>
<td>71.0</td>
<td>39.7</td>
<td>113.9</td>
<td>68.0</td>
<td>33.3%</td>
<td>-16.7%</td>
<td>-4.4%</td>
<td>0.0269%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-7</td>
<td>$210</td>
<td>66.9</td>
<td>189.1</td>
<td>115.4</td>
<td>39.7</td>
<td>114.8</td>
<td>68.5</td>
<td>-68.7%</td>
<td>-64.8%</td>
<td>-68.6%</td>
<td>-0.3208%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-3</td>
<td>$75</td>
<td>8.7</td>
<td>33.9</td>
<td>7.8</td>
<td>39.8</td>
<td>114.2</td>
<td>68.3</td>
<td>78.1%</td>
<td>70.3%</td>
<td>88.5%</td>
<td>1.0531%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-5</td>
<td>$125</td>
<td>5.4</td>
<td>22.7</td>
<td>5.3</td>
<td>39.8</td>
<td>115.1</td>
<td>68.1</td>
<td>86.4%</td>
<td>80.3%</td>
<td>92.3%</td>
<td>0.6904%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-7</td>
<td>$175</td>
<td>25.7</td>
<td>90.9</td>
<td>39.2</td>
<td>40.1</td>
<td>115.2</td>
<td>68.1</td>
<td>35.9%</td>
<td>21.1%</td>
<td>42.5%</td>
<td>0.1894%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Euclidean Distance Results from Environment 4:

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Cost</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Ave</th>
<th>Max</th>
<th>Fin</th>
<th>Improve/$</th>
</tr>
</thead>
<tbody>
<tr>
<td>RevoLDS</td>
<td>$30</td>
<td>16.3</td>
<td>37.7</td>
<td>10.3</td>
<td>44.0</td>
<td>102.7</td>
<td>65.8</td>
<td>63.0%</td>
<td>63.3%</td>
<td>84.3%</td>
<td>2.3406%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-3</td>
<td>$90</td>
<td>36.0</td>
<td>77.6</td>
<td>51.5</td>
<td>44.0</td>
<td>103.5</td>
<td>66.4</td>
<td>18.2%</td>
<td>25.0%</td>
<td>22.5%</td>
<td>0.2435%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-5</td>
<td>$150</td>
<td>18.3</td>
<td>45.1</td>
<td>20.3</td>
<td>44.4</td>
<td>102.1</td>
<td>65.5</td>
<td>58.8%</td>
<td>55.8%</td>
<td>69.0%</td>
<td>0.4081%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SRF05-7</td>
<td>$210</td>
<td>10.4</td>
<td>25.0</td>
<td>9.5</td>
<td>43.8</td>
<td>102.2</td>
<td>65.5</td>
<td>76.2%</td>
<td>75.5%</td>
<td>85.4%</td>
<td>0.3765%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-3</td>
<td>$75</td>
<td>33.2</td>
<td>79.9</td>
<td>52.4</td>
<td>43.9</td>
<td>100.3</td>
<td>64.4</td>
<td>24.3%</td>
<td>20.4%</td>
<td>18.7%</td>
<td>0.2814%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-5</td>
<td>$125</td>
<td>20.0</td>
<td>50.5</td>
<td>19.9</td>
<td>44.1</td>
<td>101.5</td>
<td>65.2</td>
<td>54.7%</td>
<td>50.3%</td>
<td>69.4%</td>
<td>0.4650%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EZ1-7</td>
<td>$175</td>
<td>12.8</td>
<td>33.1</td>
<td>11.4</td>
<td>44.2</td>
<td>102.5</td>
<td>66.0</td>
<td>71.0%</td>
<td>67.7%</td>
<td>82.7%</td>
<td>0.4217%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>