THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENTS OF ELECTRICAL ENGINEERING AND
MECHANICAL AND NUCLEAR ENGINEERING

USING MONTE CARLO METHODS FOR ROBOT LOCALIZATION

ZACHARY A. CORRELL

SPRING 2009

A thesis
submitted in partial fulfillment
of the requirements for a baccalaureate degree
in Electrical Engineering
with interdisciplinary honors in Electrical Engineering
and Mechanical Engineering

Reviewed and approved* by the following:

Sean Brennan
Assistant Professor of Mechanical and Nuclear Engineering
Thesis Supervisor

Karl Reichard
Assistant Professor of Acoustics
Research Associate
Applied Research Laboratory
Thesis Supervisor

Sven Bilén
Associate Professor of Electrical Engineering
Honors Adviser

Mary Frecker
Professor of Mechanical and Nuclear Engineering
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

We approve the thesis of Zachary A. Correll:

Date of Signature

_____ _____

Sean Brennan
Assistant Professor of Mechanical and Nuclear Engineering
Thesis Supervisor

_____ _____

Karl Reichard
Assistant Professor of Acoustics
Research Associate
Applied Research Laboratory
Thesis Supervisor

_____ _____

Sven Bilén
Associate Professor of Electrical Engineering
Honors Adviser

_____ _____

Mary Frecker
Professor of Mechanical and Nuclear Engineering
Honors Adviser

ABSTRACT

Accurate localization is necessary in order to autonomously navigate and control a robot. This thesis focuses on the implementation of a robust localization algorithm for robot navigation. This algorithm is a modified version of the Monte Carlo Localization algorithm. The objectives of this thesis are: to explain the need for accurate localization, to explain the original Monte Carlo Localization algorithm, and to present the results of the modified version of the Monte Carlo Localization algorithm.

for my parents, Samuel and Laurel Correll

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Dr. Sean Brennan and Dr. Karl Reichard for providing supervision and guidance for my thesis. I have learned a great deal about research and robotics from both of them. They have helped me capitalize on all of the opportunities available to enhance my education.

I would also like the Intelligent Vehicles and System Group who provided me with support throughout my thesis. I would like to thank Pramod Vemulapalli for assisting me with writing the code for the algorithm and for helping me to prioritize my tasks. Also, I would like to thank Richard Mattes for helping me collect data and teaching me a great deal about robotics.

Most importantly, I am grateful for the support of my parents. Throughout all of my years as a student, you have given me advice on how to excel as both a student and a person. Through all of the highs and lows of life, I know that I have you by my side. I am blessed to have loving parents like you.

This material is based upon work supported by the Applied Research Laboratory Undergraduate Honors Program.

Chapter 1

Introduction

The main objective of this thesis is to show how to accurately localize and track a robot without using dedicated global positioning system (GPS) sensors. This thesis focuses on fusing a laser range finder and odometer to implement a modified version of the Monte Carlo Localization algorithm on a mobile ground robot.

1.1 Motivation

Localization is necessary for most robotics applications. When a remote-controlled robot navigates out of the operator's eyesight, the operator must know the position of the robot with respect to its environment in order to successfully drive the robot. Likewise, most autonomous navigation algorithms operate by setting a goal location and setting intermediate goal locations in order for the robot to reach its final destination. Without localization, the robot would not be aware of its progress to the final goal and therefore could not make intelligent path planning decisions.

This work is motivated by the shortfalls of current localization equipment and sensor fusion methods for localization. Due to the complexity of environments that a mobile robot will encounter, no single sensor is accurate or dynamic enough to operate in all environments. Currently, many methods exist for localizing or tracking mobile robots, but few data fusion methods exist to accomplish both tasks in a practical manner. Robot localization and tracking are two challenges that can be treated distinctly or combined into one problem. Localizing a robot requires global information and

typically requires more information about the environment and a more complex algorithm. Tracking a robot only requires local information about the robot since its initial position is already established. In this case, the change in position and orientation have to be estimated.

1.1.1 Motivation for Research Due to Sensor Limitations

Localization sensors are primarily limited to GPS (Global Positioning System) and Inertial Measurement Units (IMUs). In an outdoor environment with full coverage, a high end GPS can provide accurate and precise localization and tracking of a mobile robot. However, an accurate GPS is an expensive sensor for mid and small sized robots. Also, GPS does not provide full coverage in all environments. Obstructions such as trees and tall buildings can significantly degrade and even eliminate GPS coverage. Likewise, GPS is useless when navigating through any type of structure such as buildings or tunnels. In a previous paper by C. Boucher, A. Lahrech, and J. C. Noyer , Particle filters have successfully fused odometry from wheel encoders with GPS in order to localize a robot when the GPS signal is lost **[2]**.

IMUs use gyros to detect changes in the pitch, yaw, and roll of the robot. The main problem with using IMUs for localization is that the sensor's localization continuously increases with time. As time progresses and the robot travels further from its operator or starting position, localization accuracy becomes more important for navigation.

Another sensor used for localization is a wheel encoder. Wheel encoders count the number of revolutions of the wheels. This information can be used to calculate the change in displacement and bearing of the robot. Similar to the IMU, wheel encoders have errors that continually increase as time progresses. Figure **1.1** shows the odometry data collected from wheel encoders along a hallway. In an ideal environment after travelling in a straight for 60 m, the robot already has 1 m error in the linear *x* direction. Wheel encoders have more error when the robot is operating in uneven terrain and when the robot is turning. When turning, the robot's track or wheels on one side move significantly more than the other side. The odometry error in this situation is highly dependent on the sensor and the motion model. Robots with tank treads increase the encoders' error through track slippage.

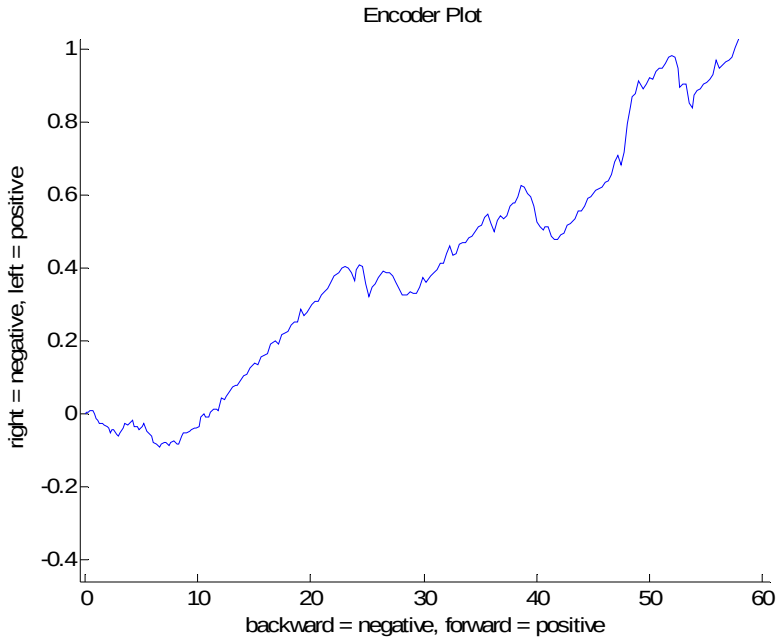

Figure **1.1**: Position Plot from Encoder Data

1.1.2 Motivation for Research Due to Existing Data Fusion Methods

Other than Monte Carlo Localization, which is an implementation of a particle filter, there are three other main areas of mobile robot localization or tracking: Kalman Filters, Markov Localization, and position probability grids. Kalman Filters are ideal for tracking mobile robots due the linear nature of the tracking problem **[1]**. Localizing a robot is a non-linear process due to its unknown position, which could be anywhere on a map. However, tracking is a linear process since once the robot is localized, only the robot's movement from its current location must be estimated. Different forms of the Kalman Filter, such as the Unscented Kalman Filter, can represent non-Gaussian distributions; however, other forms of Kalman Filters cannot re-localize the robot's position once the filter believes that it has the proper position **[4]**. The Original Kalman Filter and the Extended Kalman Filter linearize the estimation problem and only consider local rather than global information, which prohibits them from relocalizing the robot.

Another technique, Markov Localization, is ideal for localization but requires large amounts of computations and memory **[3]**. This computing limitation would force the robot to navigate at a slower than maximum speed in order to collect enough sensor readings **[3]**. Slowing down a robot in order to collect more sensor readings is an unacceptable compromise in most applications. Many Markov Localization Methods require the starting location of the robot to be known, which is impractical and at times impossible **[5]**. In a combat, covered, or previously unmapped environment the exact location might not be known or no time may be available to compute the exact location.

Position probability grids are another method used to localize and track a mobile robot. However, pre-analysis of the map is necessary in order to localize quickly **[6]**. The probability for each sensor reading must be stored for every position on the map increasing the amount of storage required to run the algorithm **[6]**. Since computation grows linearly as a function of the number of cells in the grid, another drawback of position probability grids is the computation time for large maps **[6]**. A map's precision is defined by the number of pixels per area. Since precision is a desired localization characteristic, a large map is usually necessary and sacrificing precision is often not a viable option.

1.2 Outline of Remaining Chapters

The remainder of this thesis describes the implementation and testing of an improved Monte Carlo Localization algorithm for a ground robot. Chapter 2 describes the robot's hardware and software architecture that were used to implement and test the localization and tracking method. Chapter 3 describes the Monte Carlo Localization algorithm. Chapter 4 describes the results for localizing and tracking the mobile robot indoors and outdoors. Finally, Chapter 5 presents the conclusions of this thesis and lists future work in this area of research.

Chapter 2

Robot Architecture

The mobile robot used to test the Monte Carlo Localization algorithm is the "Tankbot" developed by The Pennsylvania State University's Intelligent Vehicles and Systems Group and modified by the Penn State Robotics Club. The hardware and software were designed to be easily configurable for use in research and competition. Figure **2.1** shows the exterior view of the robot.

Figure **2.1**: Exterior View of "Tankbot"

2.1 Tankbot Hardware

The Tankbot's hardware is divided into three main areas: the power system, the control hardware, and the sensors. Figure **2.2** displays a block diagram of the Tankbot's setup required for testing.

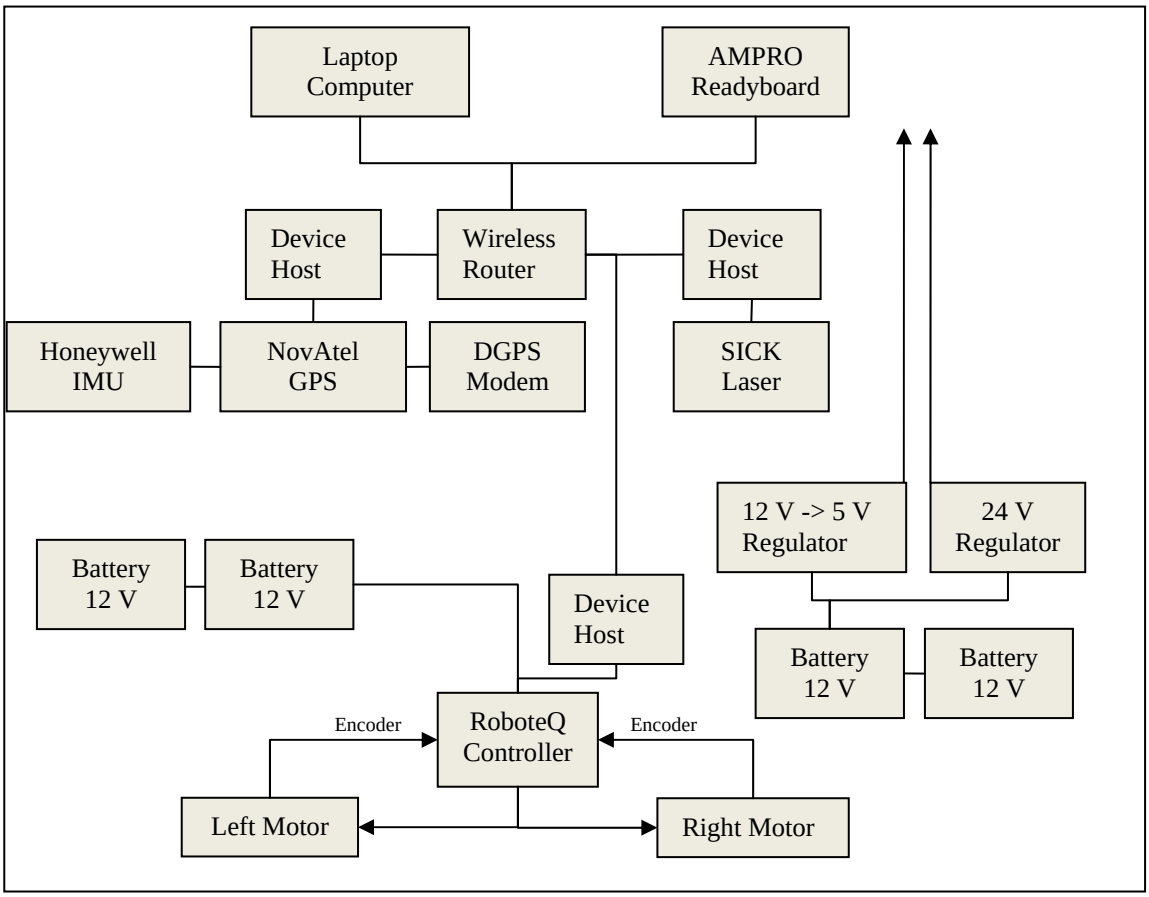

Figure **2.2**: Connectivity Diagram of the Tankbot's Hardware

2.1.1 Power Hardware

The lower bay of the robot includes the power system, remote control receiver and motor controller. Two 12 V, 18 Ah lead acid batteries are connected in series to provide 24 V to the motor controller. Similarly, another set of batteries is connected to provide 24 V to the sensors and auxiliary hardware. A power distribution box provides regulated 24 V, 12 V, and 5 V to the sensors and control equipment.

2.1.2 Control Hardware

A RoboteQ AX2850 motor controller provides up to 120 A maximum per channel to the drive system. The drive system consists of two 350 W motors with a 10:1 gear reduction. Each motor individually powers its respective side's track. The robot has a wheel base of 0.5 m and a drive wheel circumference of 0.4084 m. A remote control receiver allows the robot to operate autonomously and remotely.

2.1.3 Sensor Hardware

The Tankbot has a wide array of sensors that are used when testing the algorithm developed in this thesis. U.S. Digital optical wheel encoders are attached to each drive shaft. An Arduino microcontroller records the odometry from the wheel encoders. Although the wheel encoders are accurate, track slippage when driving straight and especially when turning can negatively affect the accuracy of the sensors.

A SICK LMS-200 Light Detection and Ranging (LIDAR) sensing system provides the distance measurements for the Tankbot. The LIDAR has a range of 80

meters and operates at an 80 Hz scan rate. The LIDAR has 0.5 degrees resolution for its 180° coverage area. Data is transferred via an RS-422 connection to a wireless router and then to an Ampro PC/104 board.

In order to test the accuracy of the localization and tracking algorithm, a GPS/IMU was used as the sensor. A NovAtel DL4plus OEM4 dual-frequency GPS receiver is combined with a Honeywell HG1700 military tactical-grade IMU to provide the true outdoor position of the robot. When utilizing the base station, the GPS receives position data accurate within two centimeters and bearing data accurate within 0.005 degrees. The IMU has a drift bias of 10 degrees/hour, an acceleration bias of three milli-g, and a sampling rate of 600 Hz. The Novatel Span system samples the system at 100 Hz and relays the data through RS-232 to a wireless router. An Ampro PC/104 Board on the robot records and processes the data.

2.2 Tankbot Software

The Tankbot's software is designed around the Player/Stage environment **[8]**. This freeware runs on Linux and provides abstraction layers for the robot's software development. In this configuration, sensor code is completely separated from the control code. Player/Stage allows developers to easily visualize sensor data and simulate sensor data through artificial environments.

In order to collect data to test the implementation of this thesis, Player/Stage was used to log and timestamp encoder, LIDAR, and GPS (position and bearing) data. The

readings were saved to a text file and converted to MATLAB's *.mat file extension. The remaining steps to process the data and implement the Monte Carlo Localization are done offline in MATLAB.

Chapter 3

Monte Carlo Localization

This chapter describes the original Monte Carlo Localization algorithm in detail. The algorithm's implementation and experimental results are also presented.

3.1 Monte Carlo Localization Theory

The Monte Carlo Localization algorithm from "Monte Carlo Localization for Mobile Robots" starts with a static map and *N* random, evenly weighted particles that are randomly distributed throughout the map **[1]**. Each particle has an *X*, *Y* and *Θ* associated with it in order to place it within the two-dimensional map.

Monte Carlo Localization uses a particle filter to implement Sequential Importance Sampling **[7]**. After the initial particle placement, the algorithm loops through the following steps. First, the position estimates *X* and *Y* are calculated by

$$\boldsymbol{X}_p^k = X_p^{k-1} + dX + w \qquad \textbf{(3.1)}$$

where dX is the change in the robot's position according to the wheel encoders and w is the Gaussian white noise variance *Q*. This equation is applied to both the *X* and *Y* components of the total displacement. *Q* is equal to the variance associated with the wheel encoders. A static model to estimate the change in displacement from the wheel encoders is calculated by

$$dD = \frac{C_w(dR+dL)}{2 \cdot T_R} \qquad \textbf{(3.2)}$$

where dD is the total displacement of the robot, C_w is the circumference of the drive wheel, dR and dL are the change in the encoder readings for the individual encoders, and T_R is the number of encoder ticks per wheel revolution. The model for the change in bearing is given by

$$d\Theta = \frac{(dL - dR)}{B_R \cdot T_R} \quad \textbf{(3.3)}$$

where $d\Theta$ is the change in bearing and B_R is the width of the robot base. Next, the weights for each particle are updated by comparing the robot's ranging information (SONAR or LIDAR scan) with the individual particle's simulated ranging information that is calculated from the map of the environment. Particles' ranging information that more closely matches the robot's ranging information will receive more weight than dissimilar particles. Then, the particles are randomly resampled from the weighted set of particles. Resampling allows particles with higher weights to receive more particles to search its area. The process of updating the particle's location and resampling is repeated until the robot is localized and can be continued through tracking as well.

3.2 Original Monte Carlo Localization Experiment

Monte Carlo Localization was first used to localize a robot in an indoor environment in 1999 **[1]**. The robot used a Sound Navigation and Ranging (SONAR) system along with odometry. This localization was successfully implemented and tested in an office at The University of Bonn, Germany as shown in Figure **3.1**. The robot's path throughout the office is displayed by a line in the figure.

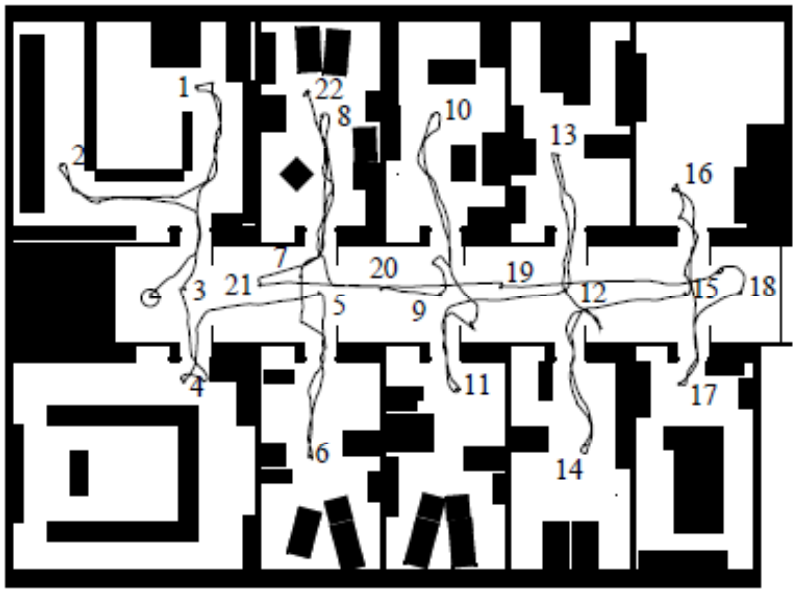

Figure **3.1**: Map of a Lab at the University of Bonn, Germany (Courtesy of "Monte Carlo Localization for Mobile Robots")

For this experiment, 20,000 particles were used to locate the robot within the office. Figure **3.2** shows the particles uniformly and randomly distributed throughout the map. Figure **3.3** shows the how the particles have gathered into two primary locations with a few minor clusters present after the algorithm has been running for many iterations. Finally, Figure **3.4** shows all of the particles gathered in one location, which is the correct location of the robot.

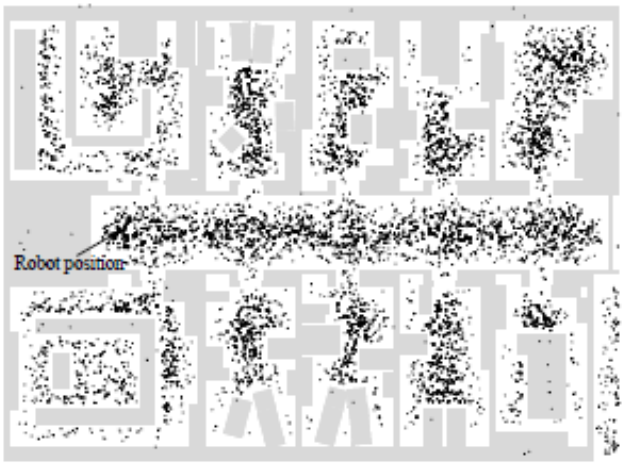

Figure **3.2**: Particles Uniformly Distributed Throughout Lab (Courtesy of "Monte Carlo Localization for Mobile Robots")

Figure **3.3**: Particles Begin to Cluster (Courtesy of "Monte Carlo Localization for Mobile Robots")

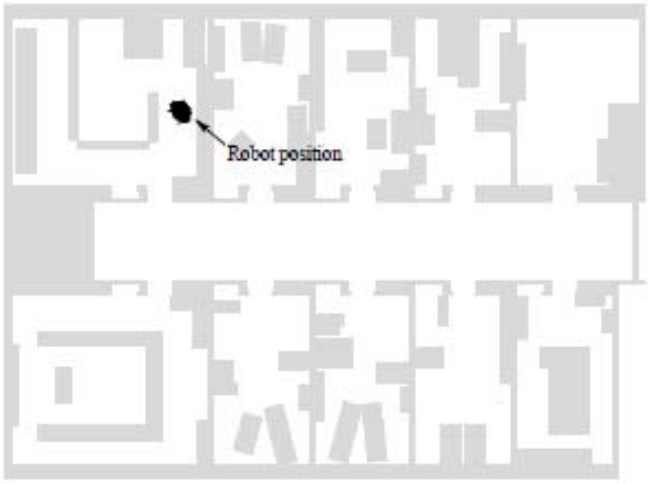

Figure **3.4**: Particles Properly Localize the Robot (Courtesy of "Monte Carlo Localization for Mobile Robots")

Figure **3.3** displays a property that will appear later in this thesis. Since the environment consists of uniformly spaced offices, the robot's position is initially ambiguous due to the symmetry of the offices. However, as Figure **3.4** shows, after many more iterations through the algorithm, the particles find the robot's location.

Another experiment was performed to test the ability of the algorithm to track a robot once its position is known. For this experiment, a robot was remotely controlled throughout the Smithsonian museum **[1]**. 5000 samples were required to track the robot for 75 minutes and 2,200 meters while never losing track of the robot **[1]**.

Chapter 4

Modified Monte Carlo Localization Implementation

In this chapter, theory for the version of the Monte Carlo Localization algorithm presented in this thesis is detailed. The algorithm was first tested using a simulated environment. After the simulation, the algorithm was tested in an outdoor environment.

4.1 Modified Monte Carlo Localization Theory

For this thesis, the Monte Carlo Localization algorithm is implemented using MATLAB and is processed off-line. The algorithm follows the same motion model from Eqns. **3.2** and **3.3** in order to update the particles' location. To compare the robot's LIDAR scan to the simulated scans from each particle, the standard weighting function for a Gaussian probability distribution function was used **[9]**:

$$q_i^k = \eta^{-1}\exp\left(-0.5 \cdot R_p^{-1} \cdot \left[(X_a^k - X_{p,i}^k)^2 + (Y_a^k - Y_{p,i}^k)^2 + (\Theta_a^k - \Theta_{p,i}^k)^2\right]\right) \quad \textbf{(2.4)}$$

where R_p is the measurement of the noise variance on the LIDAR scan, $(X_a^k - X_{p,i}^k)$ is the difference between the measured X component of the LIDAR scan and the i^{th} particle throughout the map, and η is the normalizing factor that is equal to the sum of weights q_i^k. In the original Monte Carlo Localization algorithm the type of comparison was not specified. Examples of an actual LIDAR scan and a simulated LIDAR scan from the map are displayed in Figures **4.1** and **4.2** below.

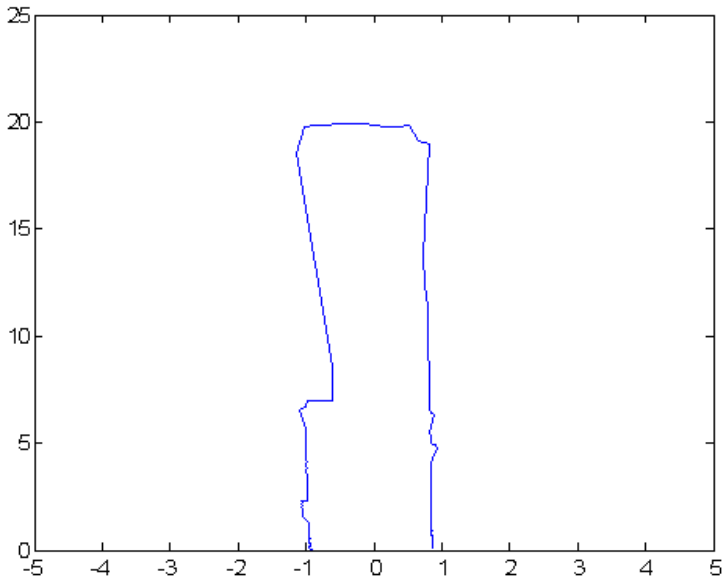

Figure **4.1**: Actual LIDAR Scan (distances in meters)

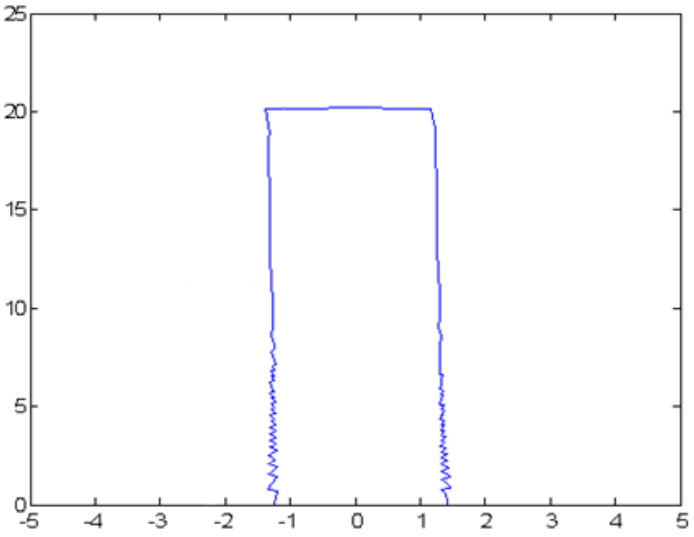

Figure **4.2**: Simulated LIDAR Scan (distance in meters)

Eqn. **2.4** is critical to the convergence of the particles and the success of the algorithm. The value of R_p, which is the measurement of the noise variance, directly affects the distribution of weight throughout all of the particles. A large R_p stretches the Gaussian curve horizontally, which leads to particles receiving similar weights regardless of their comparison with the robot's LIDAR scan. However, a small $R_p$value decreases the width of the Gaussian curve and gives particles with a close match to the LIDAR scan large weights while particles that do not have a close match will receive low or no weight. This scenario will have detrimental effects at the start of the algorithm when none of the particles have correctly identified the location of the robot. The particles will have a greater change of converging to a location that is not the robot's location.

Unlike the original Monte Carlo Localization algorithm, the modified version does not resample the particles after every iteration. Algorithm 2 from **[7]** describes the process where the particles are resampled after a set number of iterations to remove particles with small weights and duplicated particles with large weights. For this algorithm the particles were resampled after every 10 sensor readings. This value was determined by observing the behavior of the system during simulations.

The resampling steps are given below **[9]**:

$$
\begin{aligned}
&c = \text{cumsum}(q^k) \\
&u_1 = \text{rand}(1) \cdot N^{-1} \\
&i = 1 \\
&\text{for } j = 1{:}\,N \\
&\quad u_j = u_1 + (j-1) \cdot N^{-1} \\
&\quad \text{while } u_j > c_i \\
&\qquad i = i + 1 \\
&\quad \text{end} \\
&X_{p,j}^k = X_{p,i}^k \\
&\text{end}
\end{aligned}
\tag{2.6}
$$

$\text{rand}(1)$ is a random number uniformly distributed between (0,1) and $\text{cumsum}()$ is defined by

$$c_i = \sum_{m=1}^{i} q_m^k \tag{2.7}$$

The resampling algorithm requires $O(N)$ time where N is the number of particles to resample. By not resampling the particles after every iteration, the particles are allowed to search for the position of the robot. When the particles are resampled early in the process, the particles with a higher weight are not always the correct particles, which could leave to an incorrect solution.

One drawback of not resampling after every iteration is that the particles can take longer to converge to the right location. While this is a substantial concern, allowing the particles to search for a longer period of time provides a more robust solution.

Finally, the robot's position and bearing are estimated at every iteration by choosing the position and bearing of the "best particle" or the particle with the highest weight. Choosing the best particle rather than the mean of the particles will provide a more accurate estimate for the position of the robot **[9]**.

4.2 Experimental Results

The Modified Monte Carlo Localization algorithm was tested in two distinct phases. First, the algorithm was developed and tested in a simulated environment. After the algorithm worked in simulation, the algorithm was tested in an outdoor environment where a ground truth is available to test the algorithm's accuracy.

4.2.1 Simulation Testing

Developing the algorithm in a simulated environment rather than a real one allowed more focus to be placed on the algorithm rather than external parameters such as non-ideal characteristics of sensors and environmental disturbances. The algorithm accurately localized a robot in two different environments.

The algorithm was first tested on a model of a hallway in The Pennsylvania State University's Leonhard Building. The simulated map, with randomly distributed and evenly weighted particles, shown in Figure **4.3**. Figure **4.4** displays the particles converging towards the robot and Figure **4.5** shows the particles converged onto the robot.

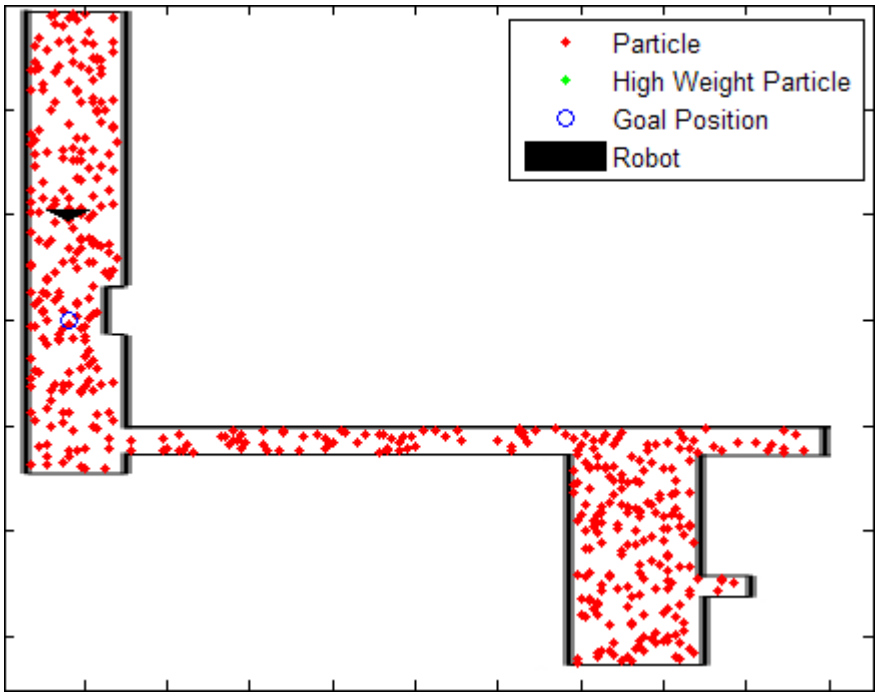

Figure **4.3**: Leonhard Hallway Map with Initial Particle Positions

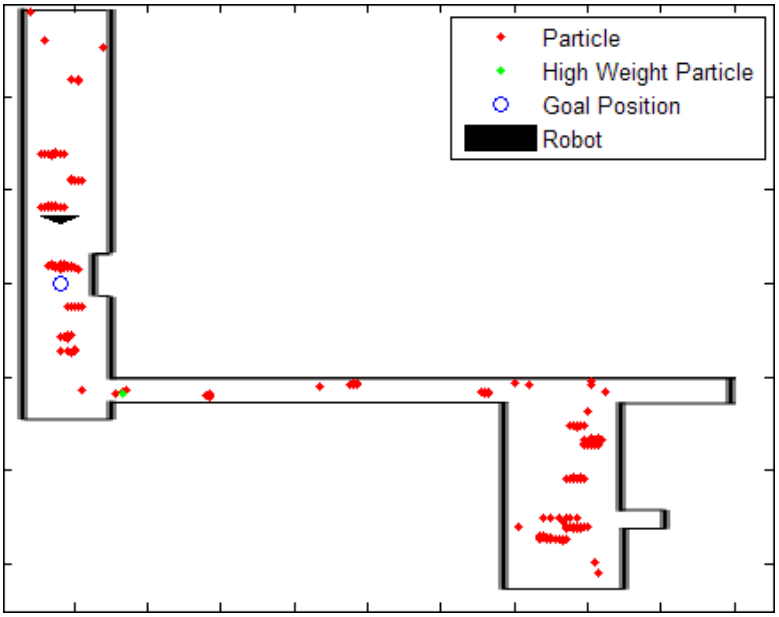

Figure **4.4**: Leonhard Hallway with Converging Particles (1)

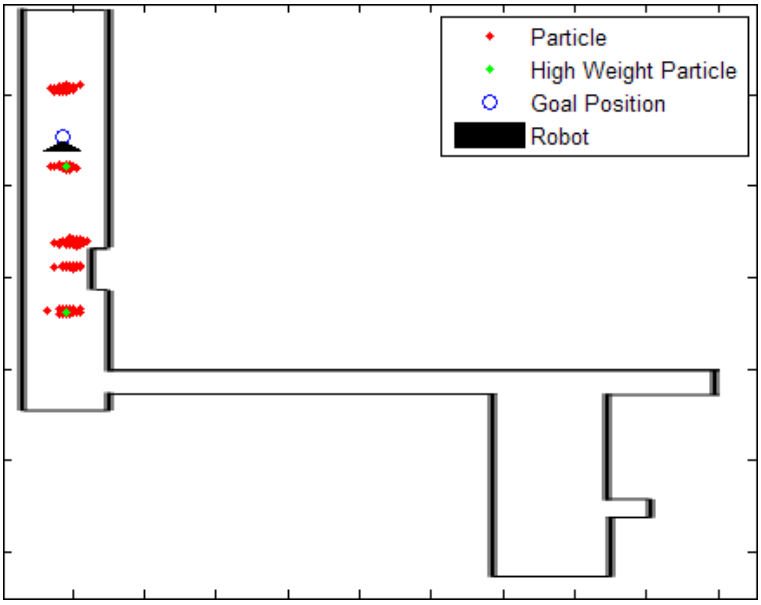

Figure **4.5**: Leonhard Hallway with Converging Particles (2)

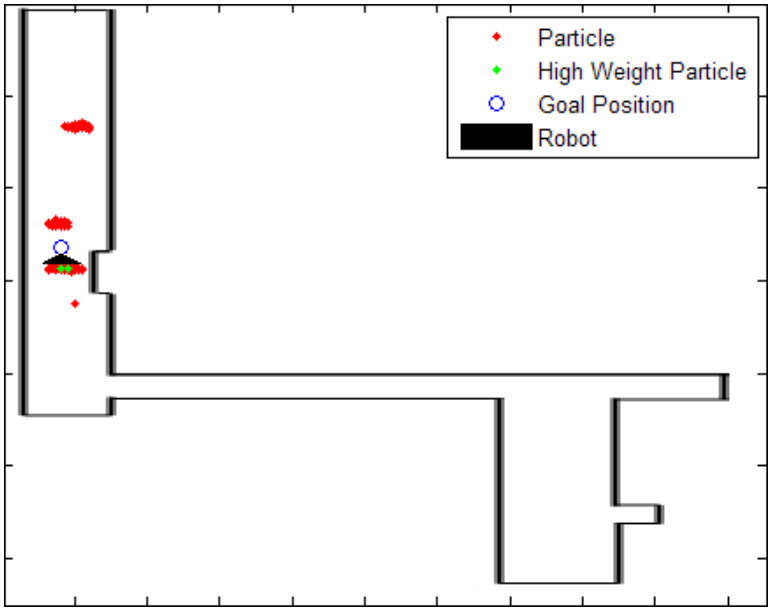

Figure **4.6**: Leonhard Hallway with Converging Particles (3)

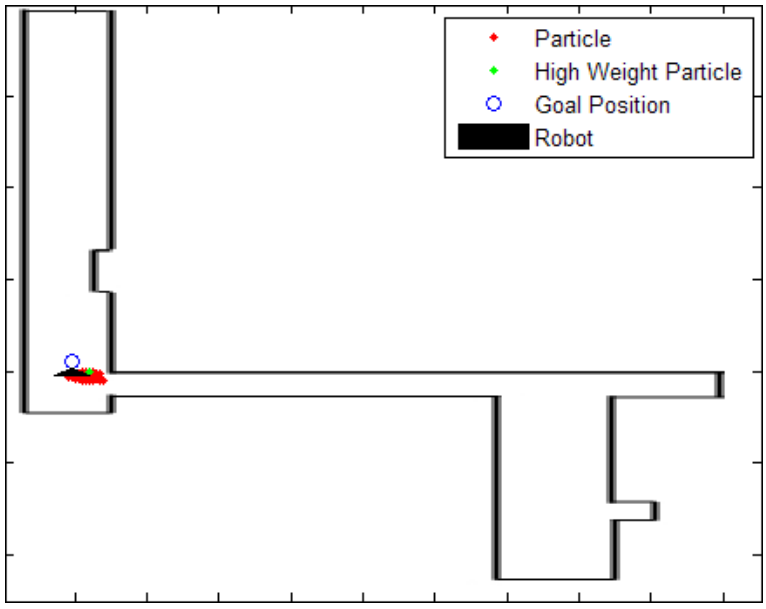

Figure **4.7**: Leonhard Hallway with Robot Localized

Although the particles eventually converged onto the robot's location, the robot had to travel 65 m. In the next series of figures (Figures **4.8** to **4.11**), an environment with more features than long, narrow hallways was used and the algorithm converged after 33.5 m, supporting the theory that the more unique of an environment in which the robot travels, the more likely the particles will converge onto the robot's location.

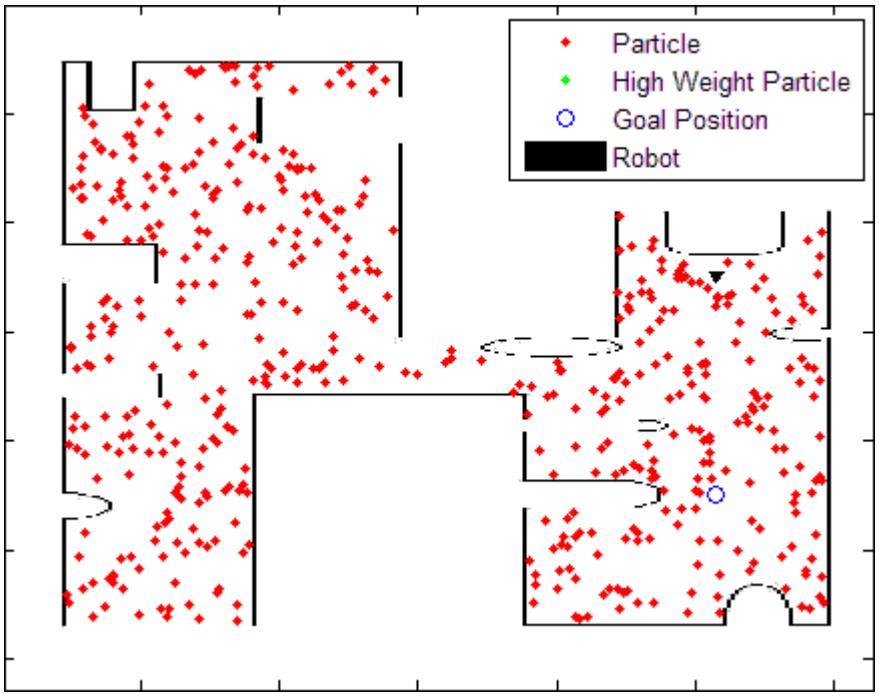

Figure **4.8**: Detailed Map with Initial Particle Positions

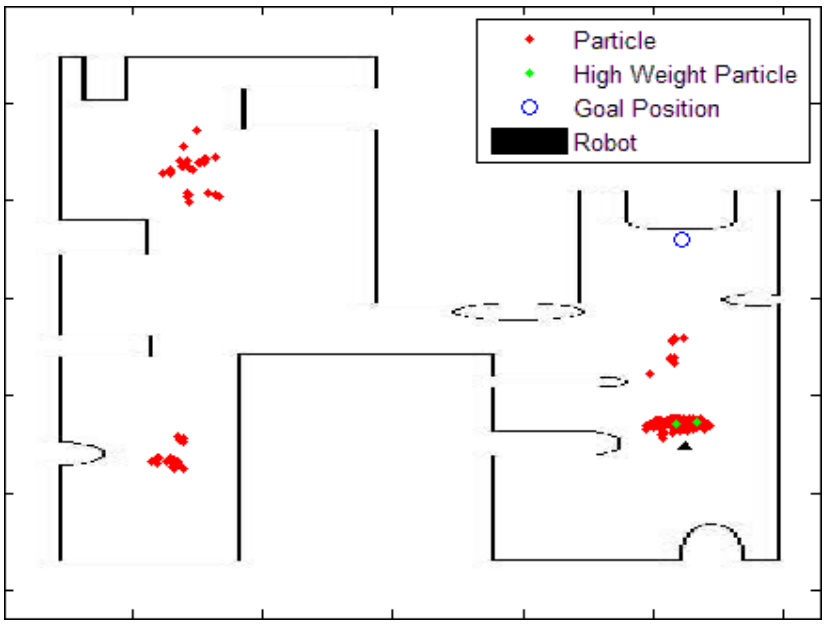

Figure **4.9**: Detailed Map with Converging Particles (1)

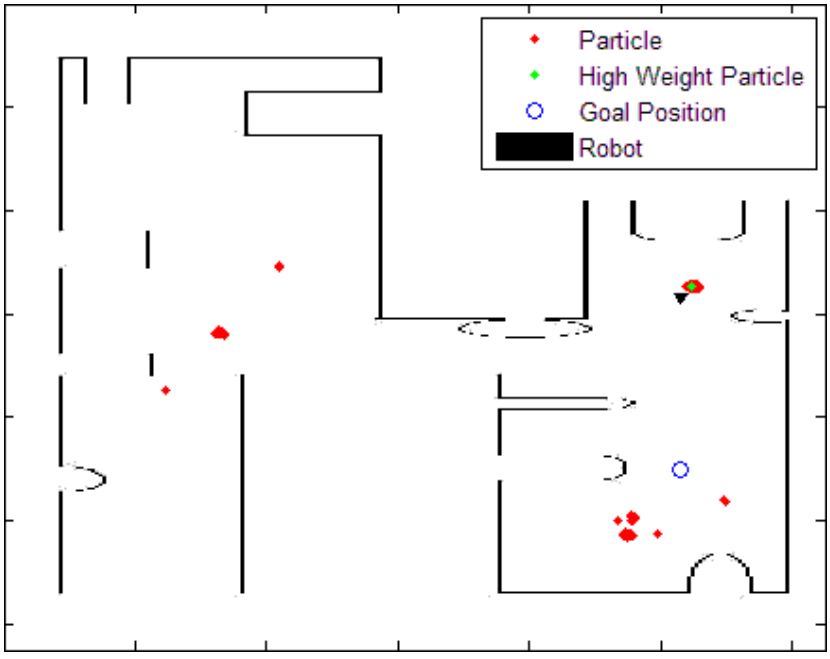

Figure **4.9**: Detailed Map with Converging Particles (2)

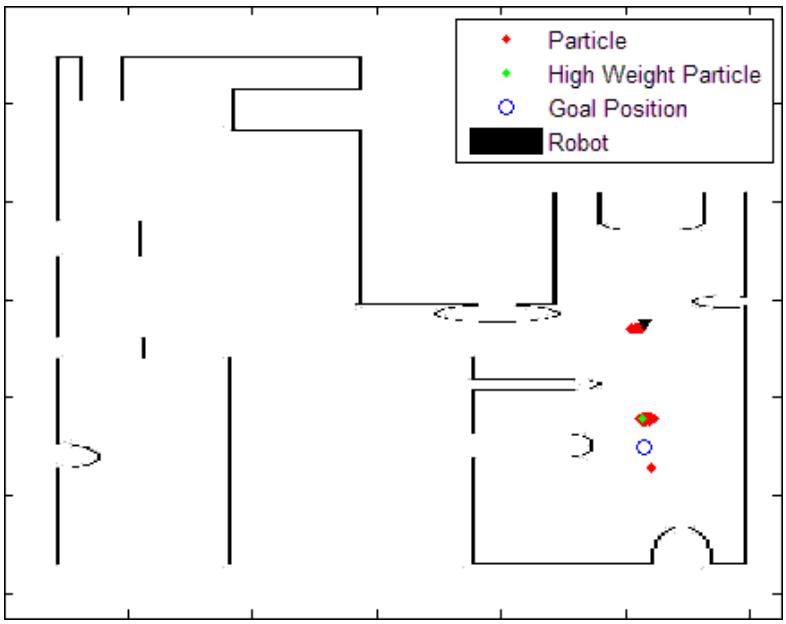

Figure **4.10**: Detailed Map with Converging Particles (3)

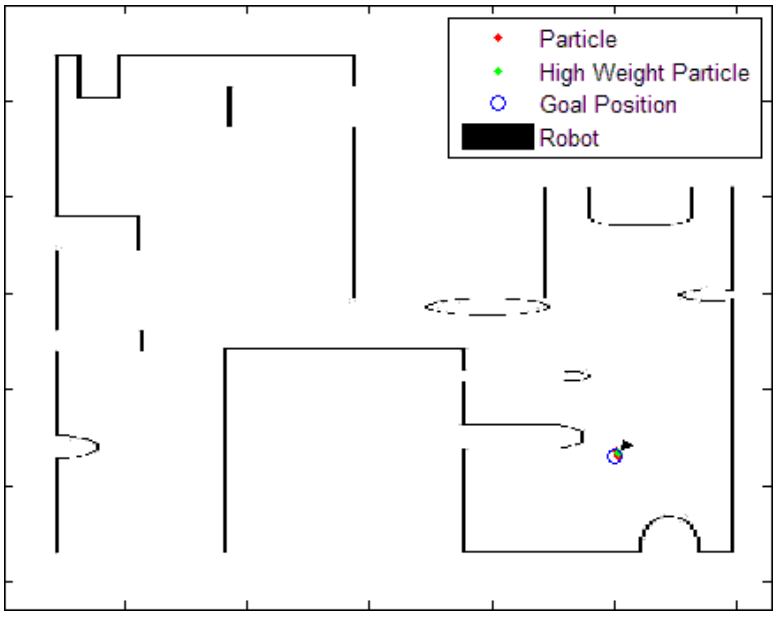

Figure **4.11**: Detailed Map with Robot Localized

All environments will have a faster convergence if a unique area of the map is available. The operator or control algorithm can identify an area where the LIDAR scans have a wide range of values and continually circle that particular area rather than traversing the entire map for localization.

By first programming the algorithm for a simulated environment, the effects of the number of particles is apparent. The number of particles has positive and adverse effects on the algorithm. The more particles that are used for localization, the more likely the particles will converge on the correct location of the robot. However, as stated in Section 4.1, resampling has a processing time of $O(N)$. Therefore, more particles will cause a linear increase in the amount of processing required by the robot. Linear growth is fast for a real-time algorithm and is a limiting factor for the practical implementation of the algorithm.

Another advantage of programming a simulated environment was the ability to identify the proper amount of information to consider when comparing LIDAR scans. The simulated LIDAR, like real LIDARs, had a 180° coverage area with 0.5° increments between scans. The algorithm was first developed using all 361 scans. However, after monitoring the particle's clustering and the speed of convergence onto the robot's location, the number was reduced by an order of magnitude. Using 361 scans caused all of the particles to have similar weights rather than better particles receiving significantly more weight. Limiting the number of scans to compare also

reduced the algorithm's runtime since comparing the robot's LIDAR scan to the particles scans is one of the most processor consuming parts of the algorithm.

4.2.2 Outdoor Testing

After successful simulation results, the algorithm was tested with data collected from the Tankbot. A testing area was established outside of Penn State's Leonhard Building. Figure **4.12** shows the constructed area that consisted of cones, traffic barrels, and tables.

Figure **4.12**: Outdoor Testing Area

A map of this area was created using the LIDAR scans collected while testing. Figure **4.13** shows the map generated from the GPS and LIDAR data. The objects appear larger than their actual size due to small amounts of error in the GPS and LIDAR readings. Figure **4.14** displays the map which was used to generate the particle's LIDAR scans. The obstacles are located in the locations given from the actual readings however the sizes of the obstacles are their measured dimensions.

Figure **4.13**: Map from GPS and LIDAR Data

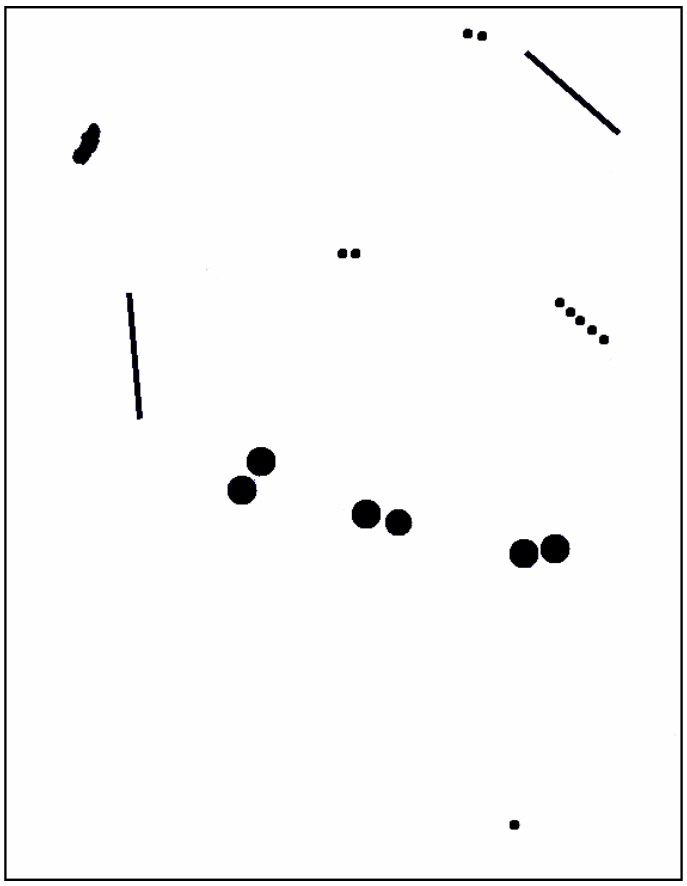

Figure **4.14**: Map of Test Environment

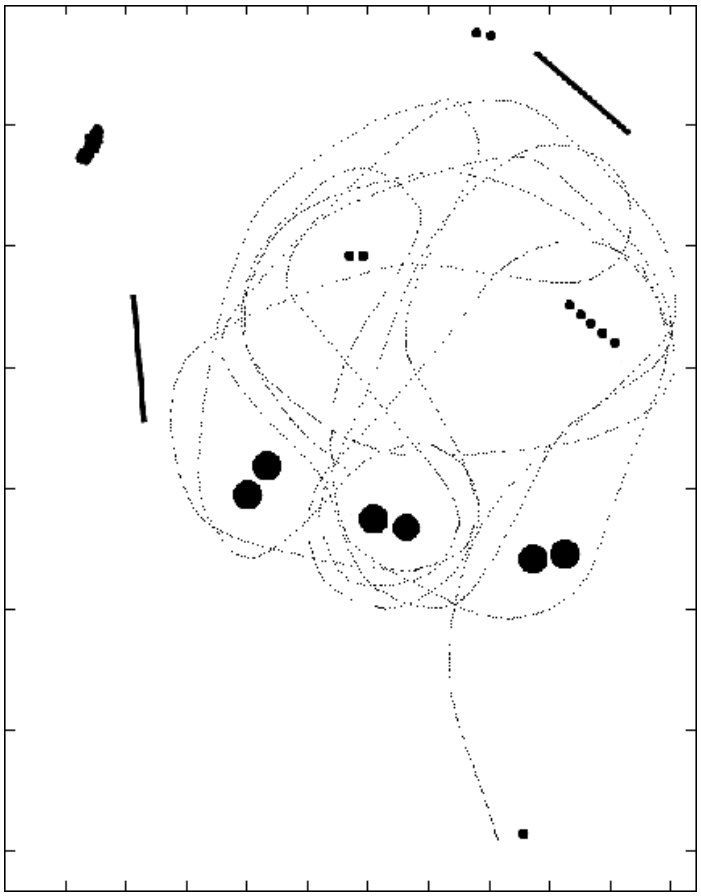

Figure **4.15**: Path of Robot during Testing

LIDAR, GPS and wheel encoder were collected as the robot drove around the obstacles. Figure **4.15** shows the path of the robot.

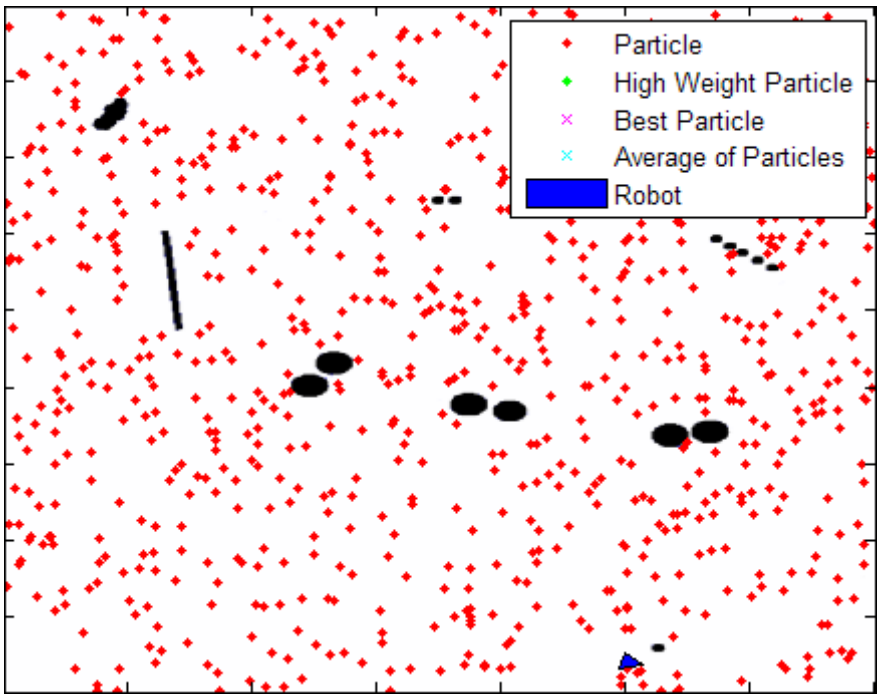

Figure **4.16**: Initial Particle Distribution During Outdoor Testing

For the given map, 800 particles and 100 of the 361 LIDAR scans were used. A measurement noise value of 5000 was used to control the rate of the particle's convergence. Figure **4.16** shows particles distributed throughout the map.

At the start of the algorithm, every particle has a weight of 0.00125. As the algorithm progresses, the particles begin to converge to specific areas as shown in Figure **4.17** to Figure **4.19**.

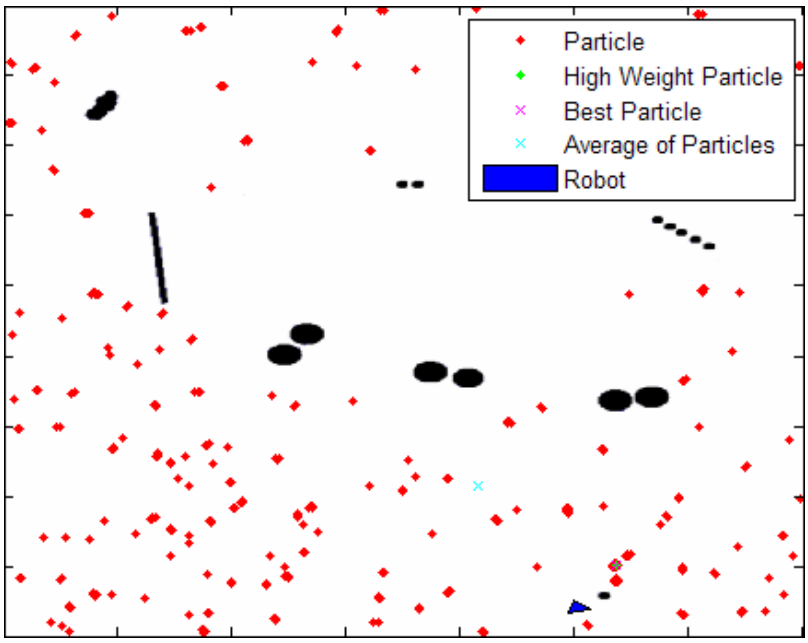

Figure **4.17**: Particles Begin to Converge to Specific Areas (1)

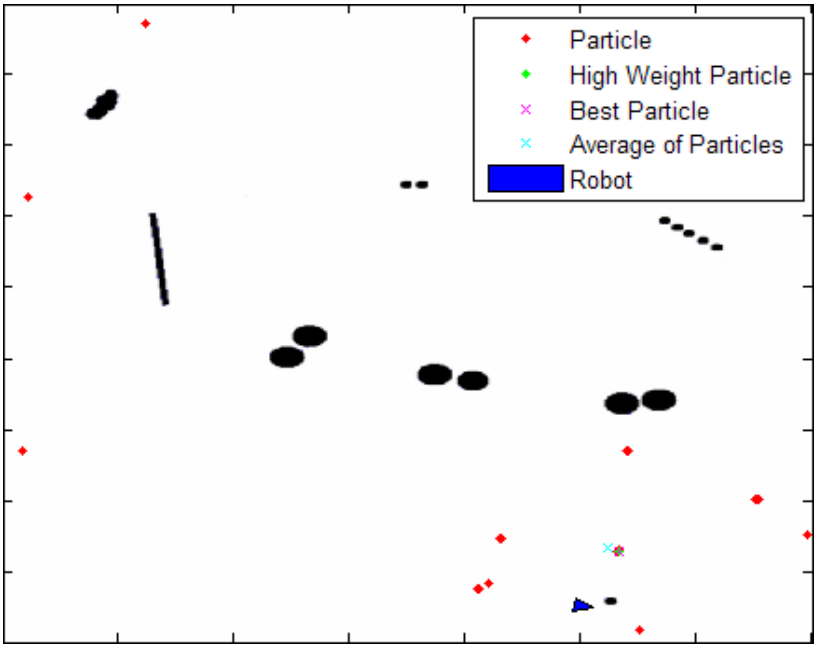

Figure **4.18**: Particles Begin to Converge to Specific Areas (2)

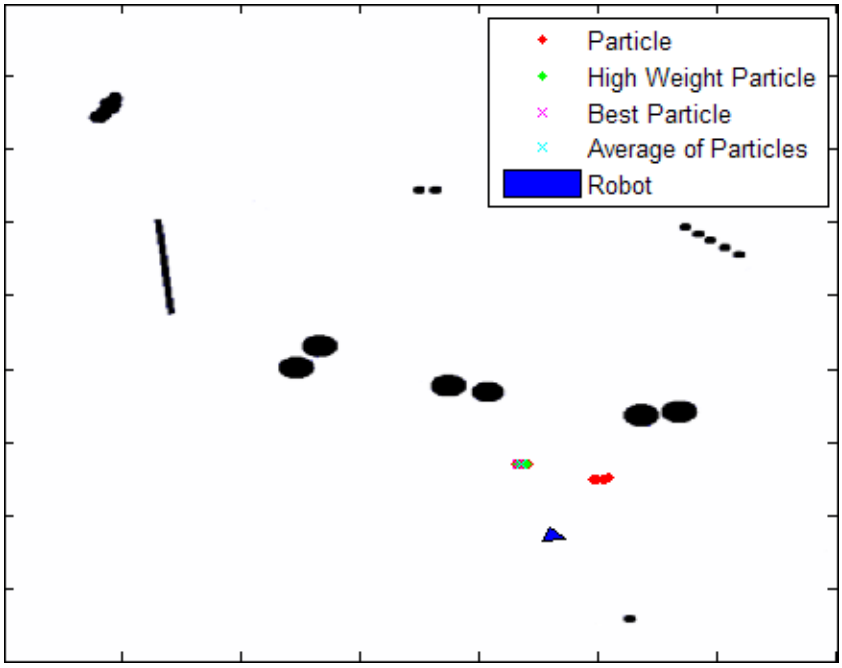

Figure **4.19**: Particles Begin to Converge to Specific Areas (3)

After resampling the particles are distributed at the same location as the highly weighted particles. This causes fewer particles to appear in the image as seen in Figure **4.18**. Before the robot moves, the particles converge to within 1 m of the robot's location as shown in Figure **4.20**. The particles converge quickly because the data was collected at a rate of 100 Hz.

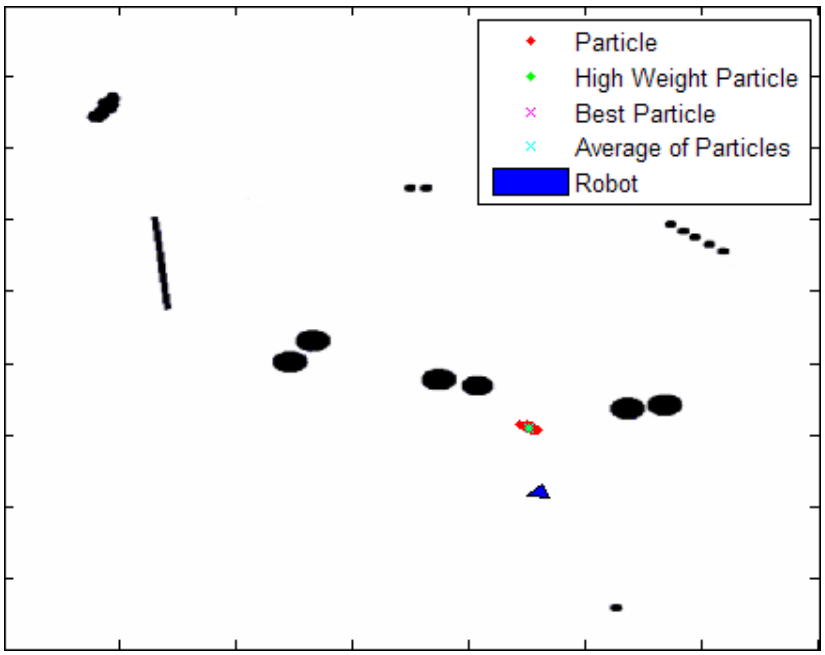

Figure **4.20**: Particles Converge to within 1m of the Robot

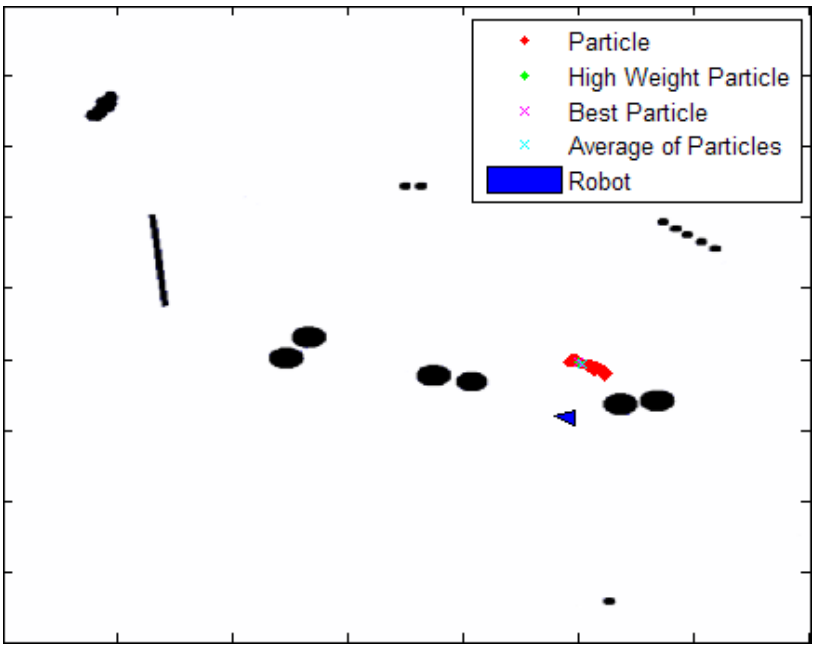

Figure **4.21**: Particles Track the Robot

In Figure **4.21**, the particles track the location of the robot within 1m.

As previously mentioned the best particle and the average location of the particle can be used to approximate the location of the robot. Figure **4.22** and Figure **4.23** show the localization error for the best particle and average of particles respectively.

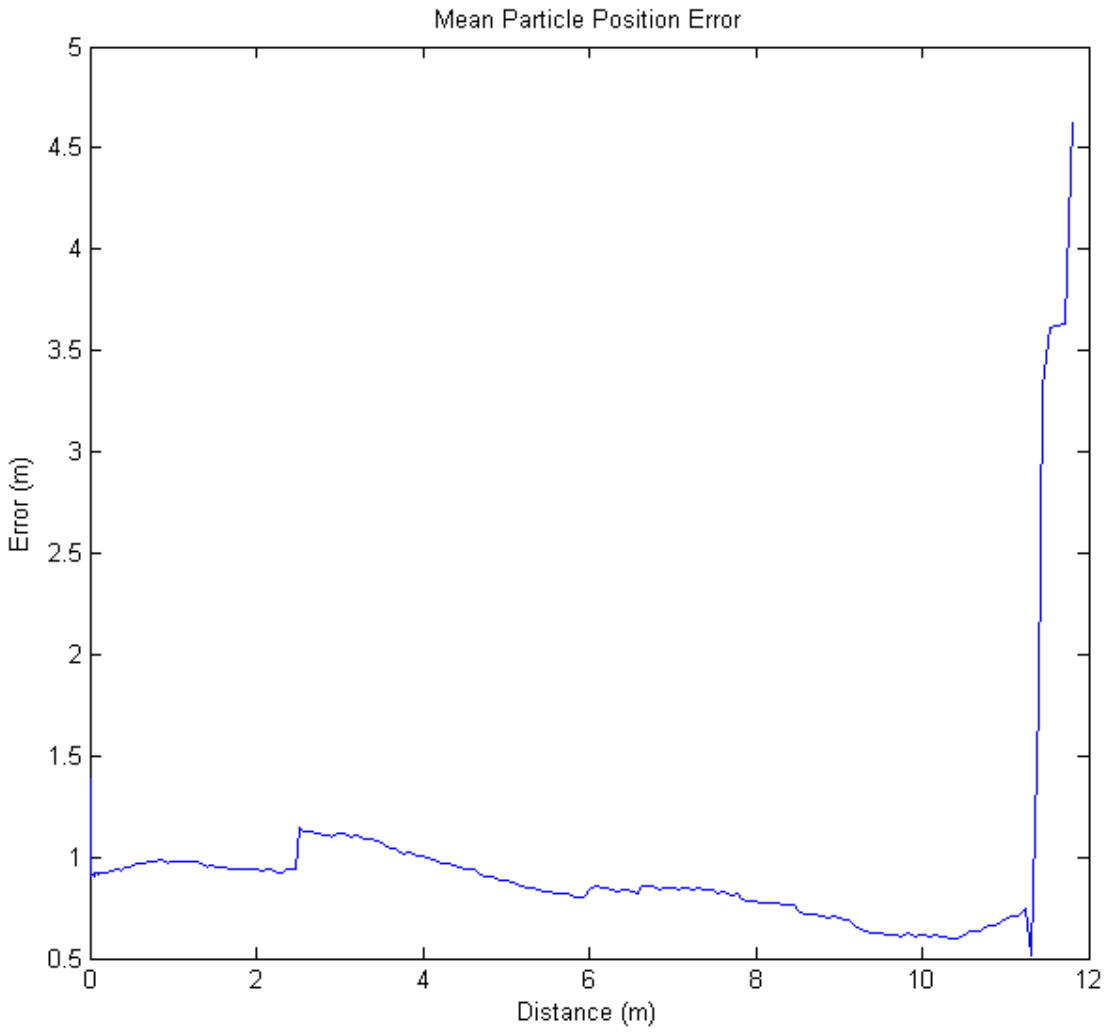

Figure **4.22**: Estimation Error for Average Particle Localization

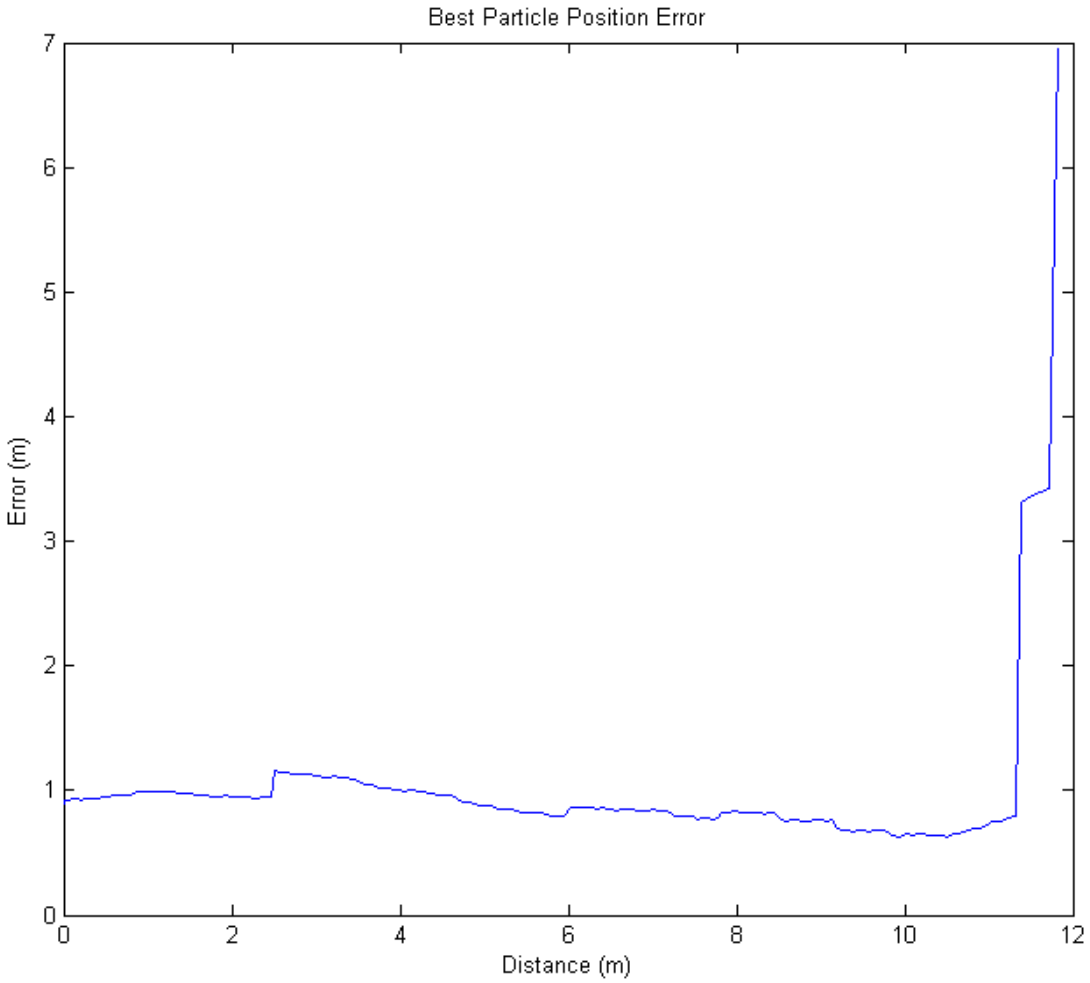

Figure **4.23**: Estimation Error for Best Particle Localization

Figure **4.22** shows that, by using the best particle for localization, the algorithm was able to localize the robot with an error of 0.63 m while in Figure **4.23** the average location of the particles was able to localize the robot with an error of 0.52 m as well since the particles converge to one location. Using the best particle achieved an error of less than 1 m after 10 sensor readings while using the average location of the particles requires twice as many sensor readings to localize the robot with an error of less than

1 m. The constant offset seen in Figure **4.22** is most likely due to ground truth error rather than an algorithm error because particles consistently cluster to the northeast of the robot.

Although the latitudinal and longitudinal position of the robot are significant, the ability to match the bearing of the robot is even more important. According to Schuler's Loop which is shown in Figure **4.24**, position has a linear affect for localization error while bearing has a quadratic effect [10].

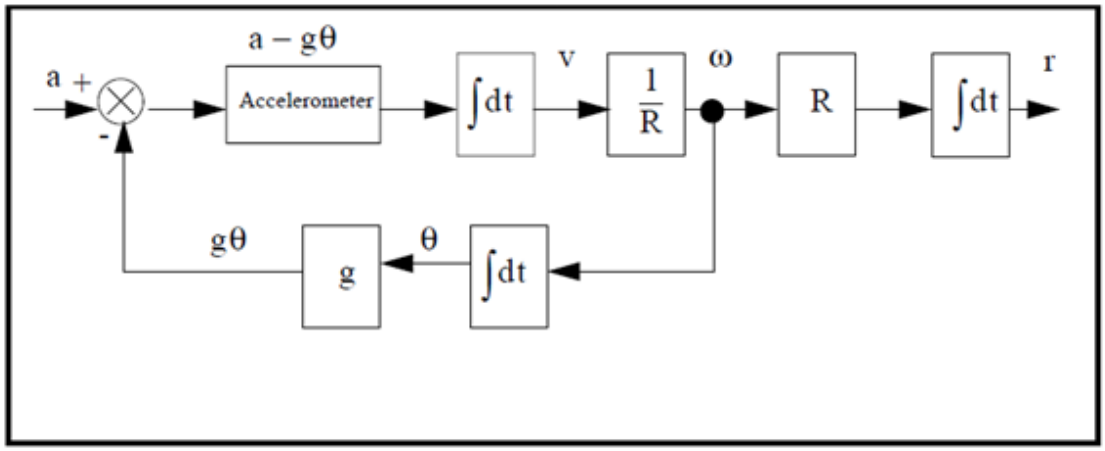

Figure **4.24**: Schuler Loop (Courtesy Of Carnegie Mellon University)

Figure **4.25** shows the plot of best particle's bearing error and Figure **4.26** shows a plot of the average particles' bearing error.

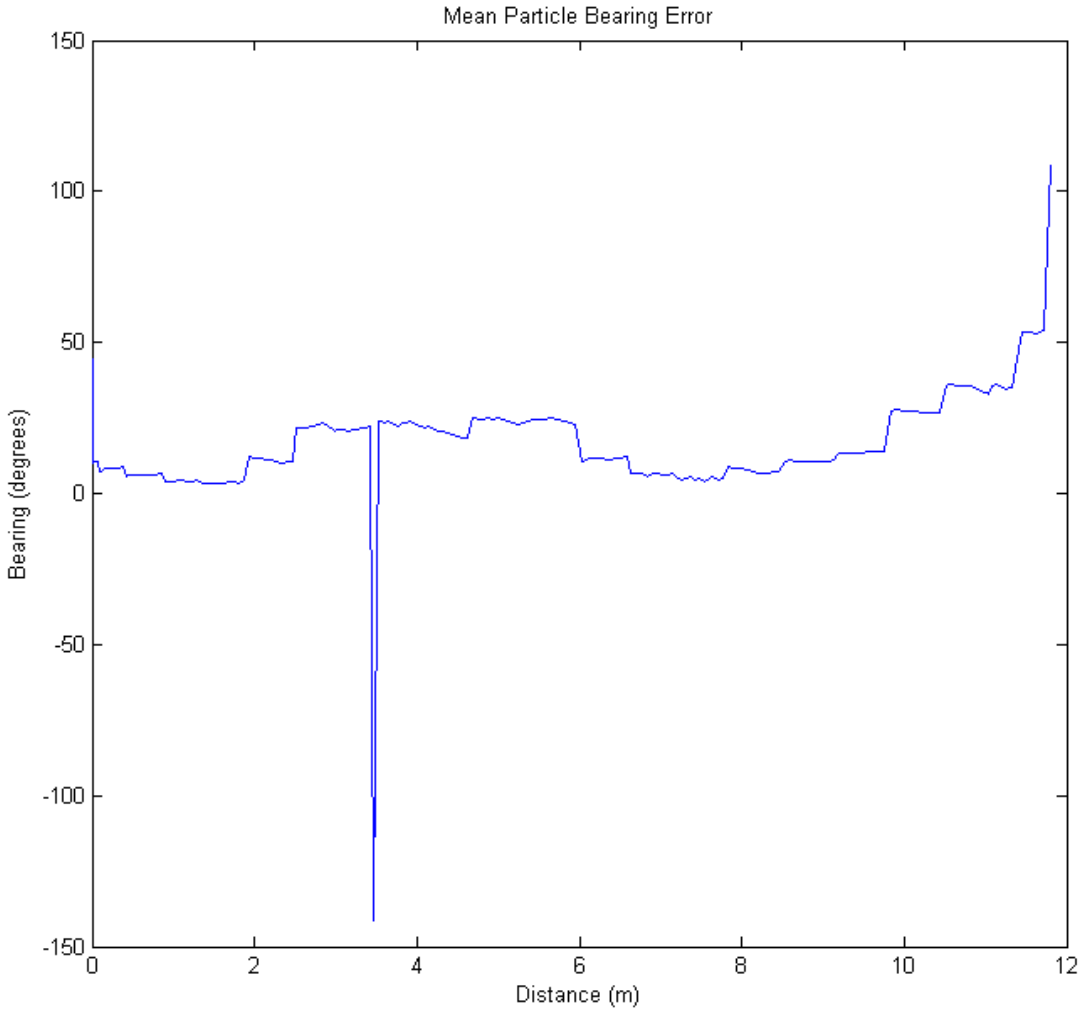

Figure **4.25**: Bearing Error for Average Particle Localization

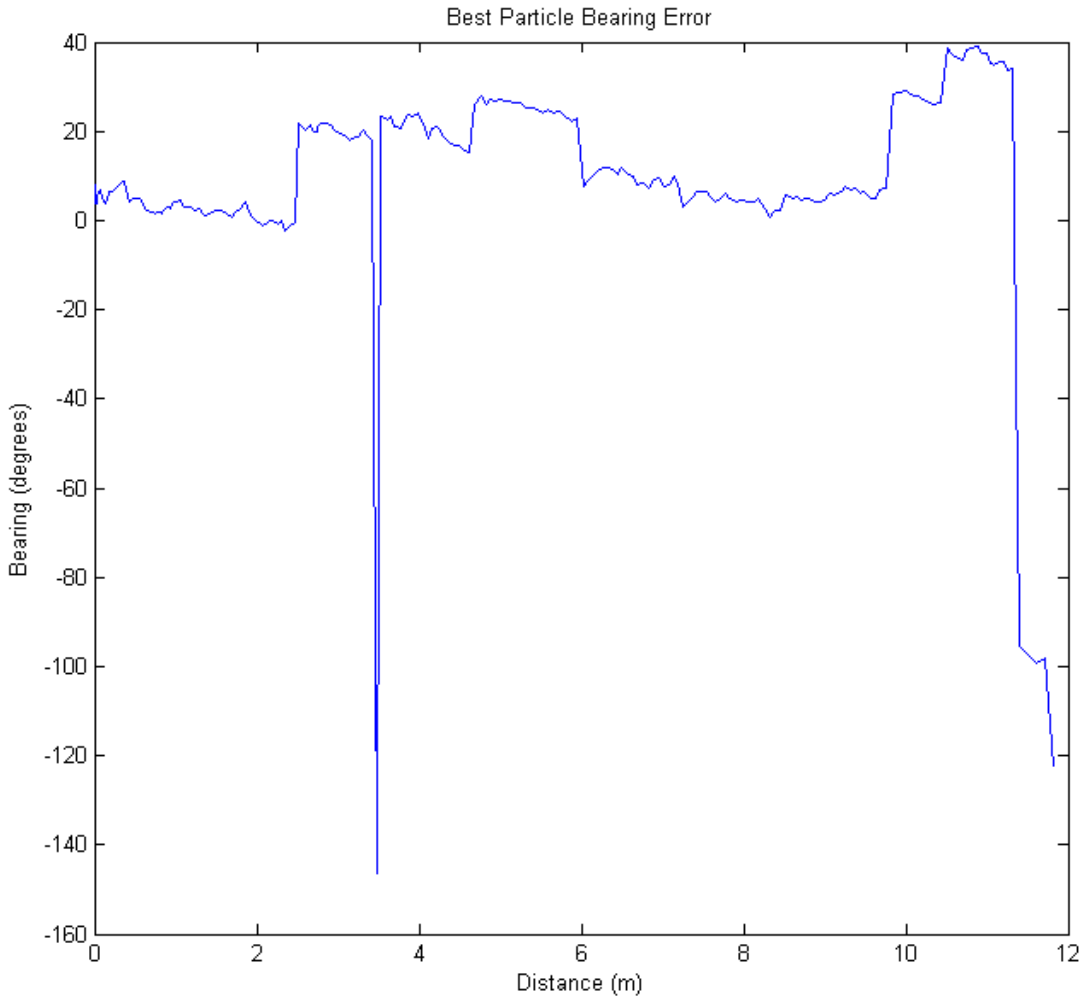

Figure **4.26**: Bearing Error for Best Particle Localization

The best particle and the mean of the particles had similar bearing error ranging from 0° to 40° while typically remaining between 0° and 15°. The 150° increase in bearing error is caused by an instantly bad GPS signal for the robot's location and not an abrupt change in the algorithm's estimated bearing error.

Figures **4.23** through **4.26** show the particles tracking the robot for 12 m. At this point, the robot moves to the right of the obstacles and the LIDAR scans of the robot cannot be compared to the particle's simulated LIDAR scan. As a result, the algorithm loses track of the robot.

Chapter 5

Conclusions

The goal of this thesis was to investigate the use of sensor fusion to accurately localize a robot. A modified Monte Carlo Localization algorithm was proposed as a means to fuse LIDAR and odometry provided by wheel encoders for localization. By modifying the resampling section of the Monte Carlo Localization algorithm, the algorithm consistently converged to the robot's position. The algorithm's effectiveness was verified through two distinct simulated environments.

This thesis detailed the effect of key parameters associated with the Monte Carlo Localization Algorithm, the R_p value for scan comparisons, the number of LIDAR scans to consider, as well as the significance of the environment's features. Finally, this thesis explained the use of Monte Carlo Localization in environments were a GPS signal is not available. A map, odometry and ranging sensors can be fused to provide accurate localization.

5.1 Future Research

Research in the area of robot localization is an important study and many new areas are available for research. First, numerous areas exist to improve the speed of the algorithm proposed in this thesis in order for it to operate in real-time. Generating each particle's simulated LIDAR scan consumes the most time and can be improved. Second, instead of using wheel encoders to provide odometry, LIDAR scan matching can be implemented to provide more accurate odometry and eliminate an extra sensor.

Finally, this algorithm can be fused with a GPS and provide localization when the GPS system fails to provide accurate information.

REFERENCES

[1] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. “Monte Carlo Localization for Mobile Robots.” IEEE International Conference on Robotics and Automation (ICRA99). 1999.

[2] C. Boucher, A. Lahrech, and J. C. Noyer. “Non-linear Filtering for Land Vehicle Navigation with GPS Outage.” Systems, Man and Cybernetics, 2004 IEEE International Conference on. 2004.

[3] W. Burgard, A. Derr, D. Fox, and A. B. Cremers. “Integrating Global Position Estimation and Position Tracking for Mobile Robots: the Dynamic Markov Localization Approach.” In *Proceedings of IEEE/RSJ* InternationalConference on Intelligent Robots and Systems (IROS). 1998.

[4] F. Abrate, B. Bona, M. Indri. “Monte Carlo Localization of Mini-Rovers With Low-Cost IR Sensors.” Advanced Intelligent Mechatronics, IEEE/ASME International conference on. 2007.

[5] A. R. Cassandra, L. P. Kaelbling, J. A. Kurien. “Acting Under Uncertainty: Discrete Bayesian Models for Mobile-Robot Navigation.” In Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems. 1996.

[6] W. Burgard, D. Fox, D. Hennig, T. Schmidt. “Estimating the Absolute Position of a Mobile Robot Using Position Probability Grids.” In Proceedings of the Thirteenth National Conference on Artificial Intelligence. 1996.

[7] S. Arulampalam, S. Maskell, N. Gordon, T. Clapp. “A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking.” IEEE Transactions on Signal Processing. 2002

[8] Player Project. 13 Apr. 2009 <http://playerstage.sourceforge.net/>.

[9] A. Dean "Terrain-Based Road Vehicle Localization Using Attitude Measurements." Diss. Pennsylvania State University, 2008.

[10] A. Kelly, "Modern Inertial and Satellite Navigation Systems," tech. report CMU-RI-TR-94-15, Robotics Institute, Carnegie Mellon University, 1994

APPENDIX

Modified Monte Carlo Localization Code (Simulation)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TwoDParticleFilter.m
% Requires map image to be in the directory
%
% Author: Zachary Correll
%         Using Monte Carlo Methods for Robot Localization
%         Pennsylvania State University
%         04/16/09
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% initialize variables
clear; close all; clc;

map = 'leonhardHallway.jpg';
robotXPos = 16;
robotYPos = 200;
robotTheta = -90;
numOfParticles = 500;
robotSize=10;
robotForwardMaxVel=2;
robotLateralMaxVel=0;
robotThetaMaxVel=4;
R = 5*10^4;
numOfLIDARScans = 361;
goalreached_flag=0;

goalXpos=16;
goalYpos=300;

% Read image and convert RGB image to binary image
globalMap = imread(map);

% convert map to binary grayscale
grayGlobalMap = rgb2gray(globalMap);
% figure()
% imagesc(globalMap); %show map
BINGlobalMap = grayGlobalMap<200;

[numOfRows,numOfCols] = size(grayGlobalMap);

% create list of valid positions for the given map
filledGlobalMap = imfill(BINGlobalMap,'holes');
se = strel('disk',4);
erodedGlobalMap = imerode(filledGlobalMap,se);
[validYValues,validXValues] = find(erodedGlobalMap==1);
numOfValidPixels = length(validYValues);

%% create initial random particles
% create uniformly distributed random positions (x,y)
```

```
particlesIndex =  floor(rand(numOfParticles,1)*numOfValidPixels)+1;
particlePosition(:,1) = validXValues(particlesIndex);
particlePosition(:,2) = validYValues(particlesIndex);

% create uniformly distributed random orientations
particlePosition(:,3) = floor(rand(numOfParticles,1)*360)+1;

find_points=find(particlePosition(:,3)>180);
particlePosition(:,3)=particlePosition(:,3)-180;

INI_position= particlePosition;

particle_valid=zeros(numOfParticles,1);
particle_postwt=zeros(numOfParticles,1);
max_particle_wt_thre=1;
odomStep =1;
iterationCount=0;

%% infinite loop to continually move the particles to the correct
location
while (1)
    % generate the movement of the robot
    validPosition=0;
    while (validPosition==0)

        [THETA,RHO] = cart2pol(goalXpos-robotXPos,-(goalYpos-
robotYPos));
        THETA=THETA*180/pi;
        if (abs(THETA-robotTheta)<10)
            if (RHO > 10)
                rand1=2;
                rand2=2*(THETA-robotTheta)/10;
            else
                rand1=0;
                rand2=0;
                goalreached_flag=1;
            end
        else
            if (RHO > 10)
                rand1=0;
                rand2=2*sign((THETA-robotTheta));
            else
                rand1=0;
                rand2=0;
                goalreached_flag=1;
            end
        end

        holdXPos=robotXPos;
        holdYPos=robotYPos;
        holdTheta=robotTheta;

        step_size=1;
```

```
        for count1=1:step_size

delPos(1,1)=(robotForwardMaxVel*rand1*cosd(holdTheta))/step_size;
            delPos(1,2)=-
(robotForwardMaxVel*rand1*sind(holdTheta))/step_size;
            delPos(1,3)=robotThetaMaxVel*rand2/step_size;
            holdXPos=holdXPos+delPos(1,1);
            holdYPos=holdYPos+delPos(1,2);
            holdTheta=holdTheta+delPos(1,3);
        end

        if(erodedGlobalMap(round(holdYPos),round(holdXPos))==1)
            validPosition=1;
            robotXPos=holdXPos;
            robotYPos=holdYPos;
            robotTheta=holdTheta;
        end
    end

    %% shift the particles according to the odometry
    for counti=1:numOfParticles
        validParticle = 0;
        while (validParticle ==0)
            delhold1
=(robotForwardMaxVel*rand1*cosd(particlePosition(counti,3)))-
(robotLateralMaxVel*rand2*sind(particlePosition(counti,3)));
            delhold2 =-
(robotForwardMaxVel*rand1*sind(particlePosition(counti,3)))-
(robotLateralMaxVel*rand2*cosd(particlePosition(counti,3)));

            particlePositionHold1
=particlePosition(counti,1)+round((0.25)*randn(1,1)+delhold1);
            particlePositionHold2
=particlePosition(counti,2)+round((0.25)*randn(1,1)+delhold2);
            particlePositionHold3
=particlePosition(counti,3)+round((1)*randn(1,1)+delPos(1,3));


            % check is particle is on the map
            if
(erodedGlobalMap(round(particlePositionHold2+1),round(particlePosition
Hold1+1))==1)%%%%%
                particlePosition(counti,1)=particlePositionHold1;
                particlePosition(counti,2)=particlePositionHold2;
                particlePosition(counti,3)=particlePositionHold3;
                if (particlePositionHold3<-180)
                    particlePositionHold3=particlePositionHold3+360;
                elseif (particlePositionHold3>180)
                    particlePositionHold3=particlePositionHold3-360;
                end
                particle_valid(counti)=1;
                validParticle = 1;
            else
                particlePosition(counti,1)= INI_position(counti,1);
```

```
                particlePosition(counti,2)= INI_position(counti,2);
                particlePosition(counti,3)= INI_position(counti,3);
                particle_valid(counti)=1;
                validParticle = 1;
            end

        end
    end

    robotXHold=[robotXPos,robotXPos-
robotSize*cosd(robotTheta+30),robotXPos-robotSize*cosd(robotTheta-
30)];

robotYHold=[robotYPos,robotYPos+robotSize*sind(robotTheta+30),robotYPo
s+robotSize*sind(robotTheta-30)];
    if (iterationCount==0)
        figure();
        h1=imagesc(globalMap);
        hold on
        % plot the current position of the robot (black arrow)
        h2=patch(robotXHold,robotYHold,[0,0,0]);
        % plot particle position (small red dot)

h3_red=plot(particlePosition(find(particle_postwt<max_particle_wt_thre
),1),particlePosition(find(particle_postwt<max_particle_wt_thre),2),'r
.');
        h3_green=plot(0,0,'g.');
        h4_goal=plot(goalXpos,goalYpos,'bo');
    else
        set(h2,'XData',robotXHold,'YData',robotYHold);
    end

    iterationCount=iterationCount+1;

    if (mod(iterationCount,10)==1)

        robotLidarDistance =
showLidarScan(BINGlobalMap,robotXPos,robotYPos,robotTheta,numOfLIDARSc
ans);

        %% calculate the particle's lidar scans
        particleLidarDistance = zeros(numOfParticles,numOfLIDARScans);
        for particleCounter = 1:numOfParticles
            if (particle_valid(particleCounter))
                particleLidarDistance(particleCounter,:) =
showLidarScan(BINGlobalMap,particlePosition(particleCounter,1)...

,particlePosition(particleCounter,2),particlePosition(particleCounter,
3),numOfLIDARScans);
            else
                particleLidarDistance(particleCounter,:)=0;
            end
        end
```

```
        %% compare the particle's lidar scan with the current lidar
scan and save the weight
        robotLidarDistance =
ones(numOfParticles,1)*robotLidarDistance;
        lidarDifference = (particleLidarDistance -
robotLidarDistance);
        sumOfLidarDifferenceSquared = sum(lidarDifference.^2,2);
        particleWeightNum = (exp((-
1/(2*R))*sumOfLidarDifferenceSquared));
        particleWeightNum(find(particle_valid==0))=0;
        particleWeight =
particleWeightNum./(sum(sum(particleWeightNum)));


        %% resample according to the weight of the particle
        particleCummWeight=zeros(numOfParticles,1);
        particleCummWeight(1)=particleWeight(1);
        for counti=2:numOfParticles
            particleCummWeight(counti)=particleCummWeight(counti-
1)+particleWeight(counti);
        end
        particleCummWeight=particleCummWeight/particleCummWeight(end);

        newParticlePosition=zeros(numOfParticles,3);

        for counti=1:numOfParticles
            q=rand(1,1);
            if ~isempty(find(particleCummWeight<q,1,'last'))
                q1=find(particleCummWeight<q,1,'last')+1;
            else
                q1=1;
            end
            newParticlePosition(counti,:)=particlePosition(q1,:);
            particle_postwt(counti)=particleWeight(counti);
        end
        particlePosition=newParticlePosition;

    end

    max_particle_wt_thre=0.99* max(particle_postwt);
    %% plot robot and particle position

set(h3_red,'XData',particlePosition(find(particle_postwt<max_particle_
wt_thre),1),...

'YData',particlePosition(find(particle_postwt<max_particle_wt_thre),2)
);

set(h3_green,'XData',particlePosition(find(particle_postwt>max_particl
e_wt_thre),1),...

'YData',particlePosition(find(particle_postwt>max_particle_wt_thre),2)
);
    drawnow;
```

```
    if (goalreached_flag==1)
        [x1,y1] = ginput(1);
        x1=round(x1);
        y1=round(y1);
        if (erodedGlobalMap(y1,x1)==1)
            goalXpos=x1;
            goalYpos=y1;
            goalreached_flag=0;
            set(h4_goal,'XData',goalXpos,'YData',goalYpos);
        else
            break;
        end
    end
    pause(0.01)
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% showLidarScan.m
% Requires map image to be in the directory
%
% Author: Zachary Correll
%         Using Monte Carlo Methods for Robot Localization
%         Pennsylvania State University
%         04/16/09
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function objectLidarDistance =
showLidarScan(BINGlobalMap,xPos,yPos,theta,numOfLIDARScans)
% map is the paint file xPos, yPos and theta are the input parameters
in
% pixels of the desired location
% returns the lidar scan distances from 0 to 180 degrees in .5
increments

%% initialize the variables
[numOfRows,numOfCols] = size(BINGlobalMap);
% create variables to hold the obstacle coordinates
lidarScanX = zeros(1,numOfLIDARScans);
lidarScanY = zeros(1,numOfLIDARScans);
foundObstacle = zeros(1,numOfLIDARScans); % flag variable in order to
stop the radial search
objectLidarDistance = 930*ones(1,numOfLIDARScans);

%% find the lidar scan given given a (x,y,theta)
for index=0:numOfLIDARScans-1
    distance = 0;
    xPrime = xPos;
    yPrime = yPos;

    % while an object is not found and the distance has not surpassed
the
    % lidar's max distance
    while((foundObstacle(index+1)==0) && (sqrt(((xPos - xPrime).^2) +
(yPos - yPrime).^2)<930)) % for current map 930 is 80m in pixels. 4
pixels = 1 ft

        distance = distance+0.95; %grow the radial search

        % the rounding off must be done only at the end , to keep the
whole
        % thing as accurate as possible
        xPrime = floor(xPos + (distance*cos((theta-
90+(floor(361*(index/numOfLIDARScans))/2))*(pi/180))));
        yPrime = floor(yPos - (distance*sin((theta-
90+(floor(361*(index/numOfLIDARScans))/2))*(pi/180))));
        % check if the lidar search is within the bounds of the matrix
```

```
        if(xPrime>numOfCols || xPrime<=0 || yPrime>numOfRows ||
yPrime<=0)
            break;
        else
            % if object is found, raise the flag and store the object
            % coordinates
            if(BINGlobalMap(yPrime,xPrime)==1)
                foundObstacle(index+1) = 1;
                lidarScanX((numOfLIDARScans - index)) =
distance*cos((floor(361*(index/numOfLIDARScans))/2)*pi/180); % change
                lidarScanY((numOfLIDARScans - index)) =
distance*sin((floor(361*(index/numOfLIDARScans))/2)*pi/180);
                objectLidarDistance(1,index+1) = distance;
            end
        end
    end
end
%plot the lidar scan and current particle
%     figure()
%
plot(objectLidarDistance.*cosd(0:0.5:180),objectLidarDistance.*sind(0:
0.5:180),'.');
%     %plot(-lidarScanX(1,:).*.0762,lidarScanY(1,:).*.0762,'.');
%     axis([-100 100 0 1000])
%     %axis([-20 20 0 40])
%     title('Lidar view from given (x,y,theta) position');
%%
return
```

Modified Monte Carlo Localization Code (Outdoor)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TwoDParticleFilterOutdoor.m
% Requires map image to be in the directory
%
% Author: Zachary Correll
%         Using Monte Carlo Methods for Robot Localization
%         Pennsylvania State University
%         04/16/09
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% initialize variables
clear; close all; clc;
load('inputData.mat')
inputData(:,1:7)=[];
%profile on;
map = 'largeObs.bmp';
numOfParticles = 800;
robotSize=20;
R = 5*10^3;

%% decide the number of lidar scans you want to compare
numOfLidarScans = 100;
% calculate indices of robot's lidar scans to compare
for num = 1:numOfLidarScans
    lidarIndex(num) = floor(361/numOfLidarScans)*num;
end

%% input data the collected data
numOfReadings = numel(inputData);
robotXPos = inputData(379,:)';
robotYPos = inputData(378,:)';
robotTheta = inputData(380,:)';
delPosX = inputData(375,:)';
delPosY = inputData(374,:)';
delPosTheta = inputData(376,:)';
robotLidarScan = inputData(2:362,:)';

%% process image
% Read image and convert RGB image to binary image
globalMap = imread(map);

% convert map to binary grayscale
grayGlobalMap = globalMap;
% grayGlobalMap = rgb2gray(globalMap);
BINGlobalMap = grayGlobalMap>100;
[numOfRows,numOfCols] = size(grayGlobalMap);
% create list of valid positions for the given map

[validYValues,validXValues] = find(BINGlobalMap==1);
numOfValidPixels = length(validYValues);
```

```
%% create initial random particles
% create uniformly distributed random positions (x,y)
particlesIndex =  floor(rand(numOfParticles,1)*numOfValidPixels)+1;
particlePosition(:,1) = validXValues(particlesIndex);
particlePosition(:,2) = validYValues(particlesIndex);
% create uniformly distributed random orientations
particlePosition(:,3) = floor(rand(numOfParticles,1)*360)+1;



findPoints=find(particlePosition(:,3)>180);
particlePosition(:,3)=particlePosition(:,3)-180;

initialPosition= particlePosition; % save initial particle position
for case when particles leave the map

particle_valid=zeros(numOfParticles,1);
particlePostWeight=zeros(numOfParticles,1);
maxParticleWeightThresh=1;
iterationCount=0;

%% loop to continually move the particles to the correct location
for index=1:1:numOfReadings
    %% shift the particles according to the odometry
    for counti=1:numOfParticles
        validParticle = 0;
        while (validParticle ==0)

            delhold1
=(inputData(375,index)*cosd(particlePosition(counti,3)));
            delhold2 =-
(inputData(375,index)*sind(particlePosition(counti,3)));



            particlePositionHoldX
=particlePosition(counti,1)+round(delhold1);
            particlePositionHoldY
=particlePosition(counti,2)+round(delhold2);
            particlePositionHoldTheta =particlePosition(counti,3)+
1*randn(1,1)+round(inputData(376,index));

            % check is particle is on the map
            particleFlag =0;
            yValueCheck = find(validYValues==particlePositionHoldY);
            for counter =1:1:length(yValueCheck)

if(validXValues(yValueCheck(counter,1))==particlePositionHoldX)
                    particleFlag=1;
                    break;
                end
            end
```

```
            if(particleFlag==1)
                particlePosition(counti,1)=particlePositionHoldX;
                particlePosition(counti,2)=particlePositionHoldY;
                particlePosition(counti,3)=particlePositionHoldTheta;
                if (particlePositionHoldTheta<-180)

particlePositionHoldTheta=particlePositionHoldTheta+360;
                elseif (particlePositionHoldTheta>180)

particlePositionHoldTheta=particlePositionHoldTheta-360;
                end
                particle_valid(counti)=1;
                validParticle = 1;
            else %if particle is off of the map replace it to its
initial position
                particlePosition(counti,1)= initialPosition(counti,1);
                particlePosition(counti,2)= initialPosition(counti,2);
                particlePosition(counti,3)= initialPosition(counti,3);
                particle_valid(counti)=1;
                validParticle = 1;
            end
        end
    end

    robotXHold=[round(robotXPos(index,1)),...
        round(robotXPos(index,1))-
robotSize*cosd(round(robotTheta(index,1))+30),...
        round(robotXPos(index,1))-
robotSize*cosd(round(robotTheta(index,1))-30)];
    robotYHold=[round(robotYPos(index,1)),...

round(robotYPos(index,1))+robotSize*sind(round(robotTheta(index,1))+30
),...

round(robotYPos(index,1))+robotSize*sind(round(robotTheta(index,1))-
30)];
    if (iterationCount==0)
        figure();
        h1=imagesc(globalMap);
        colormap(gray);
        hold on
        % plot the current position of the robot (black arrow)
        h2=patch(robotXHold,robotYHold,[0,0,1]);
        % plot particle position (small red dot)

h3_red=plot(particlePosition(find(particlePostWeight<maxParticleWeight
Thresh),1),particlePosition(find(particlePostWeight<maxParticleWeightT
hresh),2),'r.');
        h3_green=plot(0,0,'g.');
    else
        set(h2,'XData',robotXHold,'YData',robotYHold);
    end

    iterationCount=iterationCount+1;
```

```
    %% resample the particles every 10 iterations
    if (mod(iterationCount,10)==9)
        robotLidarDistance = robotLidarScan(index,lidarIndex);
        % calculate the particle's lidar scans
        particleLidarDistance =
81.91*ones(numOfParticles,numOfLidarScans);
        for particleCounter=1:numOfParticles
            if (particle_valid(particleCounter))
                particleLidarDistance(particleCounter,:) =
showLidarScan(BINGlobalMap,particlePosition(particleCounter,1)...

,particlePosition(particleCounter,2),particlePosition(particleCounter,
3),numOfLidarScans);
            else
                particleLidarDistance(particleCounter,:)=0;
            end
        end

        %% compare the particle's lidar scan with the current lidar
scan and save the weight

        robotLidarDistance =
ones(numOfParticles,1)*robotLidarDistance;
        lidarDifference = (particleLidarDistance -
robotLidarDistance);
        sumOfLidarDifferenceSquared = sum(lidarDifference.^2,2);
        particleWeightNum = (exp((-
1/(2*R))*sumOfLidarDifferenceSquared));
        particleWeightNum(find(particle_valid==0))=0;
        particleWeight =
particleWeightNum./(sum(sum(particleWeightNum)));

        %% resample according to the weight of the particle
        particleCummWeight=zeros(numOfParticles,1);
        particleCummWeight(1)=particleWeight(1);
        for counti=2:numOfParticles
            particleCummWeight(counti)=particleCummWeight(counti-
1)+particleWeight(counti);
        end
        particleCummWeight=particleCummWeight/particleCummWeight(end);

        newParticlePosition=zeros(numOfParticles,3);

        for counti=1:numOfParticles
            q=rand(1,1);
            if ~isempty(find(particleCummWeight<q,1,'last'))
                q1=find(particleCummWeight<q,1,'last')+1;
            else
                q1=1;
            end
            newParticlePosition(counti,:)=particlePosition(q1,:);
            particlePostWeight(counti)=particleWeight(counti);
        end
```

```
        particlePosition=newParticlePosition;
    end
    if(iterationCount>9)
        maxParticleWeightThresh=0.99* max(particlePostWeight);
        maxWeightIndex = find(particleWeight ==max(particleWeight));
        bestParticleWeightError(1,iterationCount-9) =
sqrt((particlePosition(maxWeightIndex(1),1)-robotXPos(index,1)).^2 ...
            +(particlePosition(maxWeightIndex(1),2)-
robotYPos(index,1)).^2)/40.7;
        meanXEstimate = mean(particlePosition(:,1));
        meanYEstimate = mean(particlePosition(:,2));
        meanParticleWeightError(1,iterationCount-9) =
sqrt((meanXEstimate-robotXPos(index,1)).^2 ...
            +(meanYEstimate-robotYPos(index,1)).^2 )/40.7;
%
set(h4_blue,'XData',particlePosition(maxWeightIndex,1),'YData',particl
ePosition(maxWeightIndex,2));
%         set(h5_yellow,'XData',meanXEstimate,'YData',meanYEstimate);
        drawnow;

    end
    %% plot particle position

set(h3_red,'XData',particlePosition(find(particlePostWeight<maxParticl
eWeightThresh),1),...

'YData',particlePosition(find(particlePostWeight<maxParticleWeightThre
sh),2));

set(h3_green,'XData',particlePosition(find(particlePostWeight>maxParti
cleWeightThresh),1),...

'YData',particlePosition(find(particlePostWeight>maxParticleWeightThre
sh),2));
    drawnow;
    pause(0.01)
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% showLidarScan.m
% Requires map image to be in the directory
%
% Author: Zachary Correll
%         Using Monte Carlo Methods for Robot Localization
%         Pennsylvania State University
%         04/16/09
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function objectLidarDistance =
showLidarScan(BINGlobalMap,xPos,yPos,theta,numOfLIDARScans)
% map is the paint file xPos, yPos and theta are the input parameters
in
% pixels of the desired location
% returns the lidar scan distances from 0 to 180 degrees in .5
increments

%% initialize the variables
[numOfRows,numOfCols] = size(BINGlobalMap);
% create variables to hold the obstacle coordinates
lidarScanX = zeros(1,numOfLIDARScans);
lidarScanY = zeros(1,numOfLIDARScans);
foundObstacle = zeros(1,numOfLIDARScans); % flag variable in order to
stop the radial search
objectLidarDistance = 81.91*ones(1,numOfLIDARScans);

%% find the lidar scan given given a (x,y,theta)
for index=0:numOfLIDARScans-1
    distance = 0;
    xPrime = xPos;
    yPrime = yPos;

    % while an object is not found and the distance has not surpassed
the
    % lidar's max distance
    while((foundObstacle(index+1)==0) && (sqrt(((xPos - xPrime).^2) +
(yPos - yPrime).^2)<1000)) % the max distance from one corner of the
map to the other
        distance = distance+0.95; %grow the radial search

        xPrime = floor(xPos + (distance*cos((theta-
90+(floor(361*(index/numOfLIDARScans))/2))*(pi/180))));
        yPrime = floor(yPos - (distance*sin((theta-
90+(floor(361*(index/numOfLIDARScans))/2))*(pi/180))));
        % check if the lidar search is within the bounds of the matrix
        if(xPrime>numOfCols || xPrime<=0 || yPrime>numOfRows ||
yPrime<=0)
            %%if (filledGlobalMap(yPrime,xPrime)==0)
            break;
        else
            % if object is found, raise the flag and store the object
            % coordinates
```

```
            if(BINGlobalMap(yPrime,xPrime)==0)
                foundObstacle(index+1) = 1;
                lidarScanX((numOfLIDARScans - index)) =
distance*cos((floor(361*(index/numOfLIDARScans))/2)*pi/180);
                lidarScanY((numOfLIDARScans - index)) =
distance*sin((floor(361*(index/numOfLIDARScans))/2)*pi/180);
                objectLidarDistance(1,index+1) = distance./40.7; %
convert back to meters for comparison
            end
        end
    end
end
%plot the lidar scan and current particle
%     figure()
%
plot(objectLidarDistance.*cosd(0:0.5:180),objectLidarDistance.*sind(0:
0.5:180),'.');
%     %plot(-lidarScanX(1,:).*.0762,lidarScanY(1,:).*.0762,'.');
%     axis([-100 100 0 1000])
%     %axis([-20 20 0 40])
%     title('Lidar view from given (x,y,theta) position');
%%
return
```

Academic Vita

Name: Zachary Correll

Education: The Pennsylvania State University
The Schreyer Honors College
Bachelor of Science in Electrical Engineering, May 2009
Interdisciplinary Honors in Electrical Engineering and Mechanical Engineering

Thesis Title: *Using Monte Carlo Methods for Robot Localization*

Thesis Supervisors: Dr. Sean Brennan, Assistant Professor of Mechanical and Nuclear Engineering

Dr. Karl Reichard, Assistant Professor of Acoustics
Research Associate, Applied Research Laboratory

Awards: Boeing Scholarship (2007 - 2009)
Robert W Gocher Memorial Scholarship (2006 - 2009)
Kruest Electrical Engineering Scholarship (2007, 2008)
Lee Hai-Sup Electrical Engineering Scholarship (2008, 2009)
Lockheed Martin Lockheed Corporate Scholarship (2007, 2008)
William and Ethel Madden Honors Scholarship (2007-2009)
Shuman and Elizabeth Moore Scholarship (2007 - 2009)
President's Freshman Award