Localization and Mapping Using NI Robotics Kit

Anson Dorsey (ajd53), Jeremy Fein (jdf226), Eric Gunther (ecg35)

Abstract

Keywords: SLAM, localization, mapping

Our project attempts to perform simultaneous localization and mapping (SLAM) using an autonomous mobile robot with a single ultrasonic ranging sensor. We developed a complex system to exchange data and instructions between the NI Robotics Starter Kit hardware and an external data-processing computer. Because of an unreliable ultrasonic sensor on the NI robot, rather than traditional SLAM we implement a novel way to build a real-time, dynamically changing probabilistic map, yielding decent performance in simple environments.

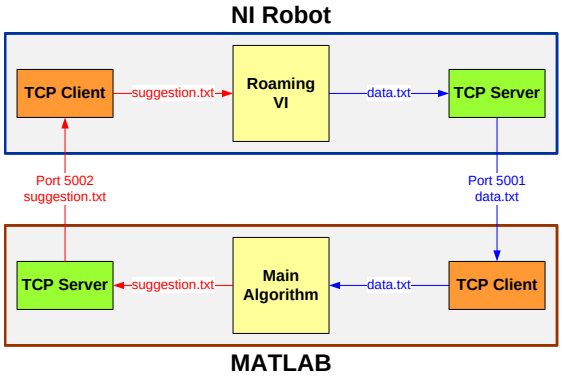
1 Introduction

Simultaneous localization and mapping using autonomous mobile robots is a fairly common problem. One of the primary challenges facing robots is a reliable way to localize and define its environment to enable it to perform assigned tasks. There are several approaches one can take in designing a robot, and we sampled two of them in the course of this project. Our initial attempt was traditional; we wanted to use a Kalman filter on odometry and ultrasonic sensor data to landmark and localize the robot in a room while using the sensor to define objects and approximate the dimensions of the room. We ran into problems when our algorithm indicated that the accuracy of the ultrasonic sensor was on par with the odometry. The sensor was too unreliable to effectively identify objects; it is also unidirectional, so our robot could only track landmarks in front of it. Our robot typically saw zero or one possible landmark, which isn't nearly enough for a reliable localization (this is probably why most other robots use a LIDAR to run SLAM). Our solution was to take a probability-based approach; we used odometry and sensor data to assess the likelihood of the robot's and objects' locations in a discrete grid of the environment. This improved the results, and we were able to obtain a very rough representation of an environment.

2 System Overview

We use MATLAB for all of our data processing and analysis on the external computer. The LabVIEW layer controls the actual robot and the TCP layer connects LabVIEW with our MATLAB base station. We have one computer which runs three instances of MATLAB: one for receiving data from the robot, one for sending data to the robot, and one to perform data analysis and navigation. The following flochart describes the dataflow of our entire system.

Dataflow of Complete System



All data analysis and TCP transfers takes place in real time as the robot roams. LabVIEW runs its main roaming VI and saves sensory data to data.txt while loading trajectory suggestions from suggestion.txt (see Section 2.1). TCP transfers updates to these files in real-time. MATLAB runs a constant loop, looking for new sensory information received from the robot in data.txt. The data is processed in real time, and plots of the robot moving in real-time along with trajectory suggestions are generated. The new robot heading is sent back to the robot to be used in its main loop, and the cycle continues unless stopped remotely. The entire engineered system effectively and efficiently performs real-time data acquisition, analysis, localization, mapping, and control.

2.1 NI Robotics Kit and LabVIEW

For this project, we were given a National Instruments Starter Robotics Kit to be programmed in the LabVIEW Robotics platform. The platform comes equipped with an FPGA processor and a higher-level processor that relay between each other. The robot is designed with differential drive and a Parallax Ultrasonic Sensor mounted to a servo, all of which are wired directly to the FPGA. Everything, including the FPGA, is programmed in LabVIEW. Unlike typical imperative programming languages, LabVIEW follows a data-flow style paradigm, where there are functional "blocks" literally wired together by inputs and outputs. There is no code involved whatsoever, only building bigger blocks out of basic, fundamental blocks. This unconventional data-flow style of programming is very inefficient and computationally weak to the experienced programmer. Therefore, rather than performing the advanced algorithms necessary for our project on the robot itself, we opted to outsource data analysis and computations to an external computer.

A major principle of LabVIEW is to use VI blocks (LabVIEW's equivalent to a function) already written and to simply modify them or use them in a bigger system rather than doing it by scratch. Thus, we used the already written Starter Kit Roaming project provided with the robot as our starting point. This project contains an FPGA VI to control the robots wheel encoders, ultrasonic sensor, and the servo motor. The main VI initializes the FPGA connections and uses standard robotics navigation VIs along with ones made for this specific platform to roam a random environment aimlessly and avoid obstacles using ultrasonic sensor data and simple obstacle avoidance algorithms. Here, the ultrasonic sensor is merely used to indicate "object ahead" or "no object ahead", and the simple algorithm, unlike complex SLAM, does not require precise nor accurate data. This algorithm runs one iteration every 10ms and continues unless stopped remotely.

Our modifications to the provided VI are focused on integrating this existing, obstacle-avoiding algorithm with our entire control system with an external computer. This involved sending ultrasonic and odometry data from robot to computer and receiving movement instructions from computer to robot. Every iteration, the counter clockwise left and right wheel velocities (in rad/sec), the servo sensor angle, and the ultrasonic sensor reading are encoded and written to a data file (see section 2.2). The VI also updates the steering frame each 10ms iteration by applying an array of three numbers, an x-direction velocity (always 0 for differential drive), a y-direction velocity, and an angular velocity, to an encoding algorithm that converts the trajectory to left/right counterclockwise wheel velocities to send to the FPGA. The trajectory chosen is either to turn and move backward if the obstacle avoidance algorithm returns an obstacle ahead or to load a trajectory from a local file that is computed by the external computer and sent real-time through TCP to LabVIEW (see section 2.2). In addition to these modifications, a TCP server and TCP client were written to cater to the LabVIEW algorithm and our system as a whole to allow real-time data transfer of sensory information and trajectory information between the robot and an external processing laptop.

2.2 TCP

From the conception stages of this project is was obvious that LabVIEW would be insufficient to perform the data processing and algorithm implementation required to successfully localize and map an environment. We decided instead to use a TCP connection to transmit data from the robot to an external computer running MATLAB. TCP is more complex than UDP yet more reliable, making TCP a better choice since we can't afford to lose any information during transfer. Outsourcing computation is disadvantageous in that the robot must still physically connect to a network, and it is impractical to run an ethernet cable to a roaming robot. Thus, we decided to mount a wireless router on the robot that any computer with a wireless card could connect to.

By necessity we threaded the TCP connections with our data processing. For simplicity, we opted to have two separate TCP connections operating in two separate threads and to let our computer's OS handle the threading by simply running multiple instances of MATLAB. One instance retrieves the latest data from the robot and another sends exploration suggestions to the robot. To form the actual TCP connections in MATLAB, we used code for an example client and server with some small modifications (iheartMATLAB.blogspot.com/).

For sending exploration suggestions, MATLAB listens as a TCP server on port 5002 for a LabVIEW TCP client. MATLAB sends 24 bytes, or 3 sets of numbers each 8 hex characters long; the first must be zero, the second contains forward velocity, and the third con-

tains angular velocity. The values MATLAB sends are determined by our controller function and act more as trajectory suggestions rather than real-time navigation (see Section 3.5).

For data acquisition, LabVIEW maintains a TCP server that waits for a MATLAB client request on port 5001. After some hacking, we figured out that LabVIEW limits TCP send size to 8192 bytes per transmission. Hence, we are forced to query data as quickly as possible. In practice a data package is acquired from the robot roughly every 1.8 seconds. Our first attempts at real-time transmission failed; our data acquisition rate was far greater than our transmission rate. Since we cannot speed up the transmission rate, data compaction is the only alternative. LabVIEW was initially transmitting data as floating point doubles, but maximum value of any single piece of transmitted data is 16, meaning we waste the most significant bits in a double representation. Instead, we convert the doubles to strings of 6 hex characters, which doubled the significance of each character from 4 bits to a full byte yet forced rounding of decimals. To keep decimal data, we multiplied each data point by 10,000 before transmission, rounded, and divided by 10,000 on the other side. After encoding the data, we can send, on average, more than 300 lines of information (3 seconds of data) in 1.8 seconds; this throughput is more than sufficient for real-time data transmission. If we had not threaded the TCP data acquisition and the data processing, the external computer would quickly fall behind the robot's data transmission, since it would take more than 3 seconds to obtain and process 3 seconds of data, reinforcing our decision to run separate instances of MATLAB.

To communicate between instances of MATLAB, the TCP client writes and received packet to a text file, and the TCP server loads data to be sent from a text file. Thus, the main processing thread has to only fetch new data from the data text file and save the current trajectory suggestion to a suggestion text file.

On the LabVIEW side, we also had to do a great deal of hacking to ensure lossless data transfers. Our main concern was running out of memory and losing new data after running the robot's real-time data acquisition for a long time. Originally we designed a system with two text files on the robot where one is transmitted while the other is written to and their roles switch upon a successful transfer, but this dropped some data and was quite unpredictable. However, compresing the saved doubles as 6 hex characters yielded sufficient storage; given that a line of data is collected every 10ms, each line has 4 values, and each value is expressed using 6 hex characters (1 byte each character), data is acquired at a rate of 24 bytes in 10ms or 2.4 kilobytes/second. The robot has 61 Megabytes of storage, and we assumed we could safely use 50MB; this allows for 347 minutes of data acquisition. Six hours is beyond the normal use of this robot, so we decided that only using one text file in LabVIEW was sufficient.

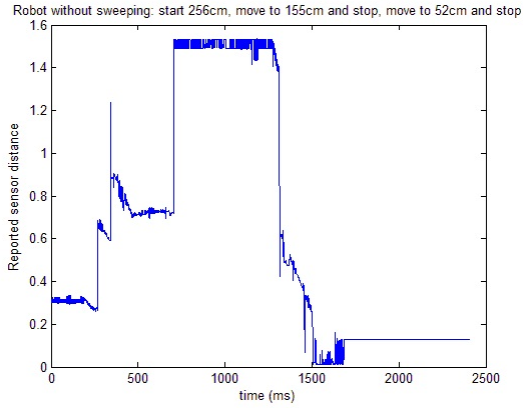
2.3 Odometry and Ultrasonic Sensor Data Reliability

The bulk of the issues we encountered in the project were direct results of poor sensor performance. Initially, we weren't aware of the issues, as the sensor gives decent performance when sweeping while the robot is moving. However, we realized the necessity for our robot controller to, on occasion, stop and sweep to extract landmarks and walls; at this point we noticed that the sensor performance was unusable for non-point objects. Once we started seeing poor data we examined the sensor results in further detail.

The largest contributing error is likely a physical restraint of the sensor. As defined in the spec sheet, the sensor projects and receives signals in a 20 degree arc, returning the smallest measured value (the closest object) in the cone, which isn't necessarily directly in front; it cannot measure with confidence that an object is directly ahead in the direction it is pointing, only that an object is +/- 10 degrees of the aimed direction. Our algorithm takes this into account and tracks the likelihood of objects in the swept cone (see Section 3.4), but the result is an inherent imprecision of localizing objects.

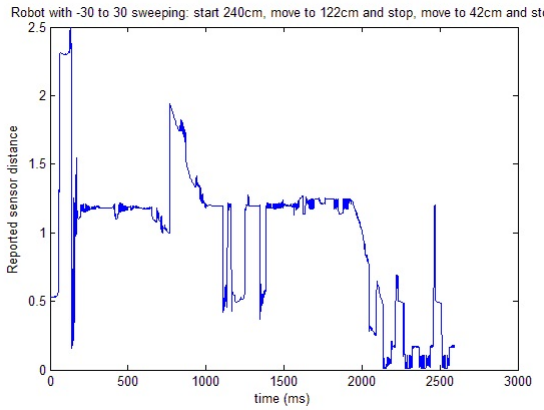
The following graphs exhibit the performance of the sensor in both sweeping and non-sweeping cases. Both tests are the same; the robot starts 2.5m away from a wall, moves to about 1.25m, pauses and then moves toward the wall and stops around .25m. In the case of no sweeping, the resulting data is unusable; we cannot identify conclusively the wall distance from the robot except when it is stationary at 155cm. At any other time, the sensor data is either incorrect or returning values below it's operating range.

Sensor Sweep



We hoped that sweeping the sensor would improve the results though this was not the case. Again, the sensor did relatively well while stationary at 122cm, but the majority of the rest of the data is unusable.

Sensor No Sweep



The sensor performs very poorly in most situations. The spec sheet states its range is 2cm to 3m, but we received accurate results in only a tiny range of the manufacturer specified operating range. More hacking and advice from Professor Saxena revealed that the best data is returned when the robot is moving and sweeping; though this is the best case scenario the resulting data is still quite bad when the robot is sensing broad objects, such as a wall, as shown by our test plots.

On the other hand, our odometry data is comparitively good; our average error from the differential drive calculation was off by 1.4cm every meter, and an average angular error of 2.2 degrees on every 360 degree rotation.

Because the sensor is so poor, we had to make drastic changes to our planned algorithm. The Kalman filter we developed gave a significant amount of weight to the odometry data; this signifies that the ultrasonic sensor results and odometry data are of comparable trustworthiness. The odometry data is also inaccurate, as we expected, but without a trustworthy sensor we had to reformulate our plan of attack on SLAM. We were unable to landmark with any accuracy since the sensor is too poor to allow for confident identification of objects.

3 Modified SLAM Algorithm

Our initial goal was to perform SLAM using traditional methodology: use a Kalman filter to improve odometry data from extracted landmarks in ultrasonic data, assuming this data is reliable enough to extract landmarks, keep track of their positions, and be inherently better than odometry data. As discussed in 2.4, this does not apply to our hardware. The ultrasonic data is much worse and less reliable than the odometry data, so there is no way to perform a traditional SLAM with a Kalman filter. Thus, we have created our own dynamic, probabilistic mapping algorithm.

3.1 Virtual Data Acquisition and Initial Testing

Early in our project, before we integrated our MATLAB algorithm with the NI robot's interface, we developed a means of creating synthetic robot data to test implementation of our algorithm. This system allows a complete simulation of the robot and the ultrasonic sensor running in a user defined environment, simulating the NI robot's role of data acquisition and data transfer in MATLAB as it relates to our entire system (see Section 2.4). To define an environment, we created a MATLAB GUI to draw walls to map out a room. The robot is simulated to roam this environment based on suggestions from a navigation algorithm (see Section 3.5) and acquire the virtual forward distance to the closest wall, thus simulating the ultrasonic sensor. Random Gaussian noise is added to the virtual odometry and ultrasonic sensor readings to mimic what we deemed to be the behavior of the robot. The virtual data is sent to our main MATLAB algorithm to be processed the same way as real data and returns navigation instructions back to the simulated robot.

This overall robot simulation allowed us to begin en-

gineering our MATLAB system while the TCP data transfer and LabVIEW VI were still being developed. We started by designing the algorithm's overall structure (see Section 3.2) with the traditional SLAM algorithm (see Section 3 header), but later realized this methodology only applied to our simulated data with Gaussian noise, not the real, unreliable, inaccurate data acquired by the NI robot. Therefore, we were forced to develop a new approach.

3.2 MATLAB Algorithm Overview

Our main MATLAB program runs on an independent thread to perform all external data analysis. The algorithm runs a loop that performs updates to the perceived map and robot's location while generating navigation suggestions (to suggestion.txt) based on any newly acquired robot data (in data.txt) since the last iteration. The loop is structured as follows:

while {robot acquiring new data}

Fetch new data (see Section 2.2)

for {each datapoint in fetched dataset}

Update robot's location from odometry data (see Section 3.3)

Update object locations from ultrasonic data (see Section 3.4)

}

Generate trajectory suggestion for robot (see Section 3.5)

Plot the updated map and location (see Section 3.6)

}

Extrapolate objects and walls from final mapping (see Section 3.6)

In MATLAB, the robot's environment is represented as a matrix with each entry corresponding to a quantized 5cm x 5cm block in space. The robot's position is represented as a coordinate in the discrete matrix. The value stored at a point (a,b) corresponds to the perceived likelihood of an obstacle being present at that block in space, based solely on ultrasonic data (see Section 3.4). Thus, our algorithm builds a dynamically updated, probabilistic mapping of the robot's environment while keeping track of a rough location of the robot, essentially achieving SLAM.

3.3 Odometry and Localization

The robot's current state is represented by an x-position, a y-position, and an absolute angle. Updates are performed to the state using standard forward kinematic equations of differential drive, modified for a robot with 4 wheels by dividing the updates by two. This method relies on the robot's wheel radius, distance between wheels, the current state, the left/right angular wheel velocity, and a time step (10ms for our robot). A state update is calculated and applied for each piece of data in the fetched data set being processed each loop iteration.

Since the ultrasonic sensor data is too unreliable, a Kalman filter cannot be applied to correct update errors from odometry errors. Instead, we initially took a probabilistic approach to localization by updating the probability of the robot occupying each block in the environment's matrix representation as it moved; over time the probabilities became too diluted to extract anything reasonable, and we opted to localize a single point. Therefore, robot's current location is based solely on odometry data.

3.4 Sensor Readings and Object Likelihood Grid

To build a map of the environment from acquired ultrasonic data, our algorithm dynamically updates an object likelihood grid, where the value at a point (a,b) correspond to a count of the number of times an object was perceived at that quantized block in the environment. As described in Section 2.3, the ultrasonic sensor returns the smallest distance perceived in a 20 degree cone, which is not guaranteed to be the direct forward distance. To account for this, we place objects along the arc of the sensor's cone at a radius equal to the sensor's measurement. With the sensor sweep angle, the robot's current location, and the sensor's measured distance, we generate a list of possible object locations for that one sensor measurement.

Using this list of possible object locations allows us to perform an update to the object likelihood grid. If an object in the list is at point (a,b), then the value at (a,b) in the likelihood grid is incremented by 1, indicating we have perceived an object there one more time. Therefore, the more we sense an object at that point, the higher the count becomes and the more confident we are an object exists at that location. Additionally, since the ultrasonic sensor returns the minimum distance in the cone, we can assume there is only empty space in the cone between the robot

and the distance returned, and any point between the robot's position and a perceived object's position cannot contain an object (if that measurement is reliable, that is). To reflect this, a ray is traced between the robot and each object in the list of possible object locations, and any point along the ray is decremented by 1 in the object likelihood grid. This corrects for misperceived objects, makes the mapping truly dynamic, and increases the reliability of the object likelihood grid as a whole. Finally, the object count at the point where the robot is located is decremented by 1; the robot cannot be where an object is, and we trust our odometry data more than our ultrasonic data.

Like the localization updates, this process of extracting a list of possible objects and updating the object likelihood grid is repeated for every acquired sensor measurement in the most recently fetched data set, and over time effectively maps the environment. The definite objects will have very large counts as they are sensed more, and areas definitely without objects will have very large negative counts as they are sensed less.

3.5 Control and Navigation

For the robot to thoroughly explore an area, it must react based on the current object likelihood grid and explore the areas that aren't well defined. There are two parts to this; one function to take the area and computes a destination and another to take a destination and current location and compute a heading suggestion in the form of forward and angular velocities.

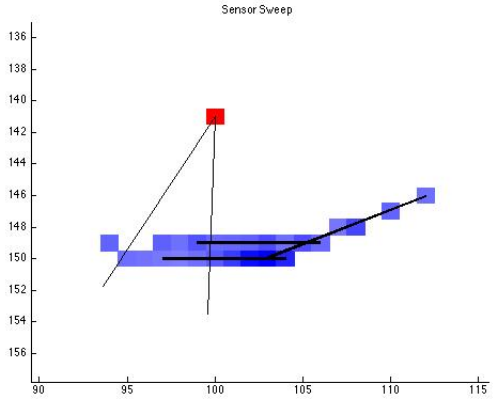
The goal of the navigation algorithm is to calculate the best vector to explore and turn it into a desired point. The algorithm looks at the object likelihood grid in a range of area surrounding the robot and calculates a weight of objects in all directions; if a direction is dense with objects (either in quantity or confidence) it is assigned a negative weight since the robot has already explored there, and if a direction is sparse of objects it is assigned a positive weight to indicate an unexplored area. It then averages the weighted vectors to calculate a desired exploration vector, which, as a result, is the opposite direction of the area with the most perceived dense objects and towards the direction of the area with the least perceived dense objects. The navigator calculates a desired point to explore to from this averaged vector and returns that point to the controller.

The controller takes the current location of the robot and a desired location from the navigation function and translates it into forward velocity and angular velocity. The controller returns one of three possible movement modes. If the desired location is within +/- 22.5 degrees of the robot's heading and greater than 2 meters away, it will move forward quickly. If the destination is within +/- 22.5 degrees and closer than 2 meters, it will move forward slowly. In any other case it turns slowly until it is oriented correctly. After calculating the trajectory, the forward speed and angular velocity are written to a text file that can be loaded and translated by the MATLAB TCP Server thread (see section 2.2).

3.6 Extrapolating Walls from Data

Once the object likelihood grid was constructed we needed a way to extract lines from the data in order to properly assess possible walls. We accomplished this using a modified RANSAC (RANdom SAmple Consensus) method that depends on a variety of parameters. The algorithm makes N attempts at extracting lines from the data points in the object grid. Each time a random point is selected and every D degree cone surrounding the point is investigated. If S points lie within M distance from the point (and within the cone) then these points are appropriate for creating a wall. To make the best possible wall we use a linear least squares data fit and then remove those points from the object grid. Once this is finished the process is repeated until N attempts have been made or there are no points left in the grid. This parametrized approach allows for tuning and catering the method to different situations.

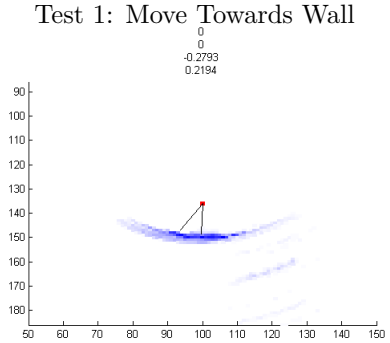
Sensor Sweep Wall Extraction



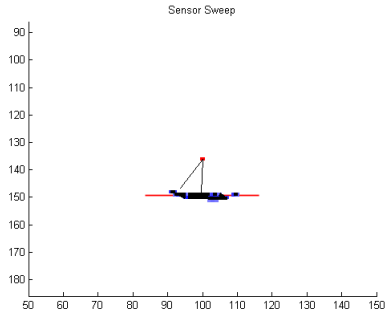
4 Results

Our first test involved moving the robot straight to a wall under manual control. As seen in the plots below, the robot sees an area of high object likelihood where it perceives the wall. The darker the blue, the more likely an object occupies that point. Because objects are extracted from a cone, the straight wall appears curved, with the outer points having less likelihood than the inside points. The final wall extrapolated, as seen below, is perceived by the robot to be .95m long, when in reality it was 1m long, an error of 5%. Subsequent tests reinforced this error. Thus, our system is capable of basic wall extrapolation when moving towards a wall.

Test 1: Move Towards Wall

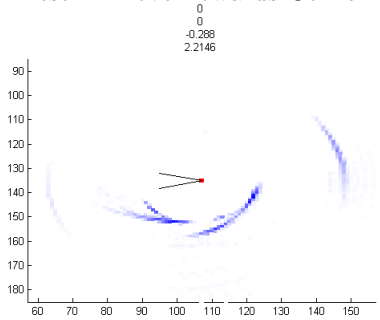


Test 1: Move Towards Wall w/ Extraction

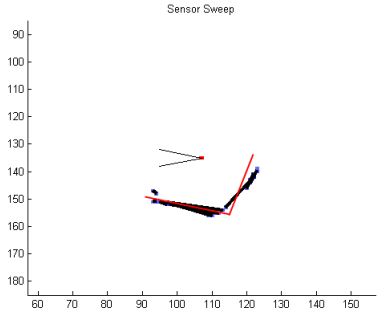


Mapping an environment consists of extrapolating straight walls, as seen in the previous test, and corners. The next test involved the robot moving towards a 90 degree corner under manual control. We guided the robot to move straight towards the corner, stop, and rotate to see the entire corner. As seen in the following figures, the robot sensed two distinct walls that were extrapolated to a corner. Although this is not ideal, we feel that this is the best possible extraction based on the data received from the robot.

Test 2: Move Towards Corner

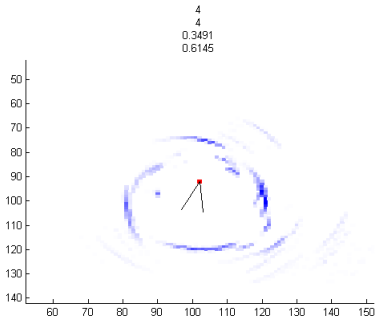


Test 2: Move Towards Corner w/ Extraction

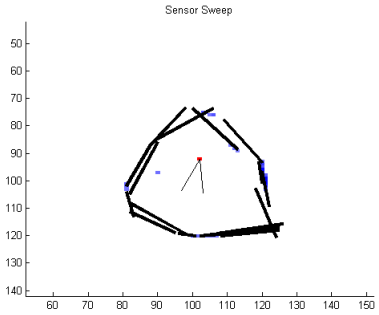


Knowing that our system is somewhat able to extrapolate corners and walls effectively from the data, the next test is to map out a simple environment. This test, along with testing our SLAM algorithm, will test our navigation's algorithm to move the robot to object-sparse areas. The plots below show the resulting map of a 2.6m x 2.3m room. The extrapolated room was perceived to be 2.54m x 2.28m, with an error of 1.6%. Because of the ultrasonic cone, the room appears to be circular rather than square an extension of the curved corner problem from the previous test. Subsequent attempts to map a simple room yielded similar results. When mapping a room with an opening (eg, a door), the robot would navigate through the opening in an attempt to map a wall in that area, proving the effectiveness of our navigation algorithm.

Test 3: SLAM Simple Room



Test 3: SLAM Simple Room w/ Extraction



5 Conclusions and Future Work

Although our results aren't as expected, we feel that we accomplished as much of our original goal as possible given our hardware constraints. Our robot can give a decent approximation of a room and is able to thoroughly explore an area; our navigation algorithm will find doorways and lead the robot through them to explore additional rooms. We feel that we were able to squeeze results out of data that had serious inaccuracies, and that our results are strong given the weaknesses of the robot's hardware. Our SLAM algorithm is designed to work around inconsistent data and, instead of trying to create a perfect map, comes up with a maximum likelihood representation of an area. With a better sensor, our algorithm would likely produce highly accurate results, even without extracting landmarks.

The practical applications of our solution are quite promising. Though the map it creates is sub-par to a robot using a very expensive LIDAR, our algorithm gives decent performance using a sensor that costs $30. For low-cost robots that only need a rough representation of surroundings, our algorithm and associated hardware provide an ideal solution.

Since all computation is outsourced and the processing / sensing on the robot can be achieved for cheap, the MATLAB Robot system we developed can ideally be used to have many small, cheap robots roam an environment to create an even more detailed, dynamic probabilistic graph. This would only require more TCP connections in the TCP MATLAB threads and essentially no change to our main processing algorithm. With more time, resources, and a better robot platform, a swarm of robots could ideally achieve our goal faster and more accurately.

6 Note to graders

Please see the wmv file attached for a video demonstration of the project. Please contact us if you wish to view the code repository as it is rather large.

References

[1] R. Thomson "TCP/IP Socket Communications in MATLAB" *Gaffer Tape and Matlab* 9/7/2009 [Online] Available: http://iheartmatlab.blogspot.com/2009/09/tcpip-socket-communications-in-matlab.html

[2] E. Papadopoulos, M. Misailidis. "On Differential Drive Robot Odometry with Application to Path Planning," Proceedings of the European Control Conference 2007 pp. 5492-5499

[3] S. Riisgaard, M. R. Blas "SLAM for Dummies"