PARTICLE FILTER BASED SLAM TO MAP RANDOM ENVIRONMENTS USING "IROBOT ROOMBA"

by

Akash Patki

Thesis

Submitted to the Faculty of the

Graduate School, Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

December, 2011

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Alan Peters

ACKNOWLEDGMENTS

I would like to put forth an honest and sincere "Thank You" to my advisor Dr. Gabor Karsai for his continuous support and patience. Our discussions helped me realize one of the overlooked assumptions. I would also like to thank Dr. Alan Peters for his input on mapping and maintaining the environment using "Grid Mapping Technique/s". I am grateful to Dr. Sandeep Neema for lending the robot - "iRobot Roomba" and ISIS for providing rest of the hardware and sensors. A special mention of Nihaar Mahatme, Indranil Chatterjee and Sayan Sen for their help on the thesis. Last and definitely not the least I thank my parents for being there and for their silent but constant support throughout my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

Baud Rate: It is the number of distinct symbol changes (signaling events) made to the transmission medium per second in a digitally modulated signal or a line code

Gaussian distribution: It is an approximation to describe real-valued random variables that tend to cluster around a single mean value.

Gaussian white noise: It is a time series (rt) that is normally distributed with mean 0 and standard deviation σ.

Markov Chain: It is a mathematical system in which a system's next state is dependent only on its current state.

Mean: Expected value of a random variable.

MIDI: (Musical Instrument Digital Interface) is an industry-standard protocol that enables electronic musical instruments (synthesizers, drum machines), computers and other electronic equipment (MIDI controllers, sound cards and samplers) to communicate and synchronize with each other

Particle Filters: These are estimation techniques based on Monte Carlo methods.

Probability Density Function (pdf): It is a function that describes the relative likelihood of a random variable to occur at a given point.

Random variable: It is a variable whose value results from a measurement on some type of random process.

SLAM: Simultaneous Localization And Mapping

Standard Deviation (Std.Dev): It is the measurement of variability or diversity of a variable.

State variable: This is one of the set of variables that describes the "state" of a dynamical system

Variance: It is the measure of how far a set of numbers are spread out from each other.

CHAPTER 1

INTRODUCTION

For any mobile robot application it is important that the robot should be able to know its location in an operating environment. The operating environment map may not be available every time. So, we need to build a map as the robot explores its surroundings. As a result, robot simultaneously Localizes and Maps the operating environment. This is (the) SLAM - Simultaneous Localization And Mapping problem [1]. The mapping can be achieved by using the robot's internal sensors, externally placed sensors and/or by referring to the map of the operating environment. Examples of internal sensors could be accelerometer, proximity sensors, etc. Ultrasonic Distance sensor, Compass module, Color sensors, etc. are the examples for external sensors. The "map" mentioned here maybe a database of obstacles the robot knows beforehand, while "referring to" could be a lookup of the database based on robot's current state. Robot's internal sensor readings are also known as *odometry readings*. But, the odometry readings (e.g. measuring distance travelled or angular rotation by the robot) are error prone. Navigating a map solely on the estimate of the known position and current set of readings is called "Dead Reckoning". As the new position is calculated based on previous values the errors/uncertainties (that are cumulative in nature) grow with time. So, we need external sensors as measuring devices. While some external sensors (like direction sensor) provide better estimate for odometry readings, others (like Ultrasonic range sensor) help determine a measurement, robot is unable to make by itself. There are a few variations of the problem where, the robot has entire map of the environment, no map at all or just a few landmarks [24]. In any of these cases, robot needs to localize itself based on the sensor measurements. Robot localization is the process through which robot tries to determine on its position in a given map, known or unknown. SLAM aims to iteratively, either build a map or improve a known existing map and at the same time keep track of robot's current position. SLAM is applicable for both 2D and 3D motion.

The other aspects of this problem are the sensors and the filters used to process the readings from sensors. We cannot rely just on the odometry readings from robot's internal sensors because they tend to accumulate errors over time. We use external sensors in conjunction with filters to overcome this problem. Particle Filters [26] generally work for any kind of system. This is because

sufficiently large numbers of particles/samples tend to take into account most of the possible states a system can be in at a particular point of time.

Particle Filter is a Monte Carlo based simulation technique which uses particles or samples to represent different possible states a system can possibly be in at a given point of time. These samples represent Probability Distribution Function (PDF) of the state. It works for any kind of system (linear or non-linear, Gaussian, or non-Gaussian, etc.). We get a better estimation of system state if we use higher number of samples to represent the PDF. The only drawback of maintaining higher number of samples is that, the whole process becomes computationally expensive.

The work presents, a working prototype of "iRobot's Roomba" and simulations for mapping a random maze using MATLAB and Blender [2]. Blender is a 3D animation tool that provides a rich feature set for realistic robotic simulations.

Section [II] presents the background of SLAM problem and Particle filter. Section [III] presents the "Mathematical Model" and describes (in detail) Monte Carlo Simulation Method –"Particle Filter" and the stages that make up the entire Slam Process. Pseudo code is presented where ever necessary for a complete understanding of the process. Section [IV] delves into "Implementation Details" – the robot (hardware) setup and coding details. Section [V] presents the "Results" of simulation (both MATLAB and Blender) and Roomba. Section [VI] and [VII] are "Conclusion" and "Related Work" respectively.

CHAPTER 2

BACKGROUND

Simultaneous localization and mapping (SLAM) is a technique used by robots and autonomous vehicles to build up a map within an unknown environment (without a priori knowledge), or to update a map within a known environment (with a priori knowledge from a given map), while at the same time keeping track of their current location. This section sets up the problem of SLAM and provides a brief introduction to Particle Filter.

2.1 SLAM Problem

Maps help us navigate and locate objects in a given environment. Mapping of an environment involves locating and mapping objects under conditions of error/noise. SLAM binds the processes of locating and mapping together in an iterative feedback loop to improve the results. FIG 2.2.1 is a flow chart that depicts logic flow in the SLAM process.

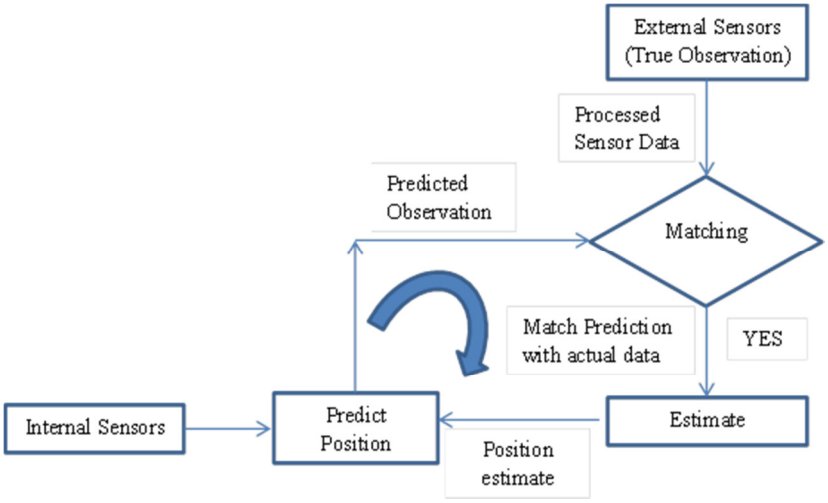

FIG 2.2.1 SLAM Flowchart

SLAM tries to answer two fundamental questions for a robot “What does the world around me look like?” and “Where am I?” In order to answer these questions the robot needs to interpret sensor measurements, estimate its current pose and take into account the inherent uncertainties in

sensor measurements and robot movement. Over time, this results in a cumulative buildup of errors and an inaccurate map. This, however, can be compensated using a number of statistical techniques. Statistical techniques help model the noise and their effects on measurements. Kalman and Particle Filters are the most commonly used techniques in SLAM. SLAM finds its applications in various real life situations where automated vehicles need to map the environment during disaster relief, underwater navigation, airborne systems, minimal invasive surgery, visual tracking, etc.

A mobile robot, while operating in a 2 dimensional environment is represented by a 3-tuple.

$$\chi = [x, y, \theta] \qquad 2.1.1$$

where x, y are the Cartesian co-ordinates and θ is the robot orientation – the angle robot makes with the positive X-axis. Initially the 3-tuple is set to [0,0,0].

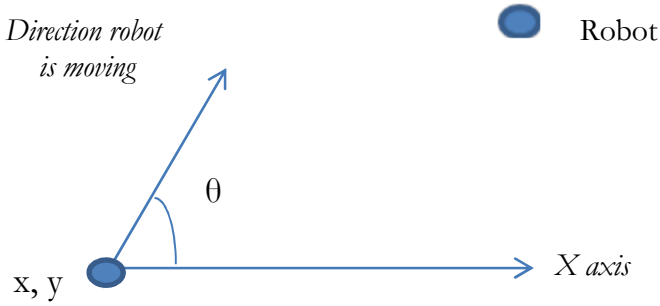

FIG 2.1.2 System Definition

With every control, consisting of a velocity (V), turn rate(R) and time step (k)[1] robot moves according to the motion model in equation 2.1.2[51]

$$\begin{bmatrix} x(t+1) \\ y(t+1) \\ \theta(t+1) \end{bmatrix}^T = \begin{bmatrix} x(t) + V * k * \cos\big(\theta(t) + R(t)\big) + e_x \\ y(t) + V * k * \sin\big(\theta(t) + R(t)\big) + e_y \\ \theta(t) + k * R(t) + e_\theta \end{bmatrix}^T \qquad 2.1.2$$

Here, e_x , e_y and e_θ are the noise terms that take into account wheel slippage, control response, etc. While moving around the environment robot makes measurements to the landmarks [51].

[1] k = The time duration for which the robot moves

$$\emptyset = arctan\left(\frac{x_o - x_{t+1}}{y_o - y_{t+1}}\right) - \theta(t) + N(0, \sigma_\theta) \qquad 2.1.3$$

$[x_o, y_o]$ is co-ordinate of the obstacle. $N(0, \sigma_\theta)$ is the white Gaussian noise[52].

$\emptyset$ is the angle robot makes with the obstacle

The SLAM problem boils down to provide an estimate for the landmarks and at the same time positioning itself in the environment. This is achieved though the three stages of "Prediction", "Update" and "Resampling". Detailed explanation is provided in SECTION III.

2.2 Filters

Kalman and Particle Filters are the most commonly used filters in SLAM to compensate for errors in measuring and movement.

Kalman is a recursive filter that provides an optimal estimate of the desired state of a linear system from a series of noisy measurements. The results are optimal for a linear system with white Gaussian noise. This poses a certain constraints in developing solutions for SLAM. Recursive filters, memory-wise, are expensive. Most of the real life systems are non-linear. As a result Kalman filter cannot be applied to these systems.

On the other hand, Particle Filters (PF) are a class of Monte Carlo Simulation methods. They are used in cases where we see conditional independence between random variables connected in a Markov chain i.e. the process is memoryless or the next state depends only on the current state and not the events that preceded it. Tracking framework provided by PF is robust enough to be applied for non-linear systems or/and non-Gaussian noise or/and multi nodal distributions.

The main aim of "Particle Filters" is the estimation of state variable (χ) based on observed data (data calculated using odometry readings) and sensed data (data calculated using external sensors). A Probability Distribution Function (PDF) is used to represent the state information. PF uses a large number of samples (x_i) to represent possible system states i.e. PF takes into account multiple state hypotheses simultaneously.

The PDF represented by these particles evolves based on motion model of the system. At time zero, system PDF is represented as $P(\chi_0)$. As the system evolves with time, PDF gets updated and the PDF at time $(t+1)$ is $P(\chi_{t+1}|\chi_t)$. Since probability of measuring an observation (Z) is $P(Z_{t+1}|\chi_{t+1})$. Using Bayes formula we have [51],

$$P(\chi_{t+1}|Z_{t+1}) \propto P(Z_{t+1}|\chi_{t+1}) * P(\chi_{t+1}|\chi_t) * P(\chi_t|Z_t) \qquad 2.2.1$$

Equation 2.2.1 is used to calculate the weights of samples representing the system. Implementation of PF includes passing the samples through motion model, calculating the weights and (if needed) resampling of the particles.

Theoretical development of the model can be found in [45]. An outline of particle filter is given below

Initialize particles x_i
Propagate the particles through motion model at the end of each time step $x_i \rightarrow x_i'$
Calculate weight for each sample $P(x_i'|Z_t)$
Resample particles according to their weights. $x_i \rightarrow x_j$

A more detailed explanation of Particle Filter and the different stages involved is given in SECTION [III].

2.3 Related Work

With robots being used widely, robotic mapping is one of the most important applied areas. [6] presents the work done over a period of time. The current work is related with SLAM, where in the robot, while mapping the unknown environment tries to localize its position [7]. Recent works focus on variations of SLAM like FAST SLAM [12] and Visual SLAM [13, 14]. Particle filters are frequently used in vision applications [27, 28, 29, 30]. Particle filters are also used to detect and track multiple objects using Boosted Particle Filters, Hierarchical Particle Filters [31, 32]. Different flavors of Particle Filter, e.g. sampling importance resampling (SIR) filter [42], auxiliary sampling importance resampling (ASIR) filter [43], regularized particle filter (RPF) [44], etc. are used based on different assumptions. SLAM in conjunction with "Particle filter" is used to map large 3D outdoor environments [16, 17]. A number of techniques are used while mapping either 2D or 3D environments: Planar 2D mapping [33], Planar 3D mapping [36, 34, 35], Slice-wise 6D mapping [37], Full 6D SLAM [38. 39, 40], Globally consistent image alignment [41]. The environment to map here is a maze and while many maze traversal algorithms exist [8] this work simply uses a random path.

CHAPTER 3

PARTICLE FILTER

The main objective of a particle filter is to track the State variable (χ) as it moves through and tries to map the environment. Based on some actions, the State variable keeps changing. The objective is to provide a better estimate for this variable. Multiple copies of the same variable are associated with a weight function. These copies are called *samples (particles)*. The weight function decides the importance of each particle. The updated value of State variable is weighted sum of all particles. Greater the number of particles better the accuracy. But this increases the processing time.

Particle filter is an iterative process that has two main stages: prediction and update. At the end of each iteration, particles are modified based on a model. This is the prediction stage. Once this stage is complete, weight associated with every particle is recalculated based on the latest sensory readings. This is the update stage. Sometimes, after a few iterations, the weights of certain particles become so small that they do not contribute at all. As a result the particle set needs to be re-sampled.

In our case State variable (χ) that we track represents the position robot. The sensory readings are the obstacles in the maze (i.e. the walls). χ is called pose of the robot and is presented as a 3-tuple.

$$\chi = [x, y, \theta]$$

Also, χ is represented by a set of M samples or the "*particle set*".

$$S_i = [\chi_i, w_i] : i = 1 \dots M$$

Here, i is the index for every particle while w_i is weight of the particle that it contributes during update stage.

Since pdf at time *t* is known, using the motion model we predict pdf at time *t+1*. Using the distribution at time (*t+1*) we update the state variableχ. We use weighted mean[2] method to update the variableχ. As a result, particles with most weight contribute in the update process while lower weight particles eventually get left out.

[2] A mean that is computed with a weight given to every particle in a sample set.

The algorithm for particle filter is given below

```
while (stillexploring)
  obstacles = sense(χ)
  particleSet = motionModel(particleSet)
  particleSet = calcWeights(particleSet, obstacles)
  χ = update(particleSet)
  if (EffectiveSampleSize(particleSet) < threshold)
    particleSet=resample(particleSet)
```

Algorithm: Particle Filter pseudo code

This section focuses on the mathematical models that go into implementing a "Particle Filter". It also presents pseudo code for a resampling algorithm. But, initially it describes and explains the basic assumptions on which the models are built upon.

3.1 Assumptions

The work focuses on implementation of SLAM process using Particle filter to localize an unmapped random maze based on the following assumptions.

- *Almost no Drift:* This means that the robot, when it moves from "Point A" to "Point B" will have an almost zero angular displacement. i.e. both wheels move relatively at the same speed. Experimental measurements with the robot have confirmed this assumption.

- *Gaussian distribution for motion model:* The errors in measurement of sensors, odometry readings of the motion model along with the noise have a Gaussian distribution. As a result we need to calculate "mean" and "Standard Deviation" values also referred to as system constants of sensors.

- *Number of Particles:* The work assumes that 1000 particles are sufficient enough to accurately estimate value of the state variable.

- *In place Angular Displacement:* This means that every time robot turns, only change is in its orientation and not the co – ordinates and even if there is, it's negligible.

3.2 Prediction Model

We use Gaussian model to predict pose for the robot and for probability distribution of the samples. There are a few other models [4-5] that can be used instead of the Gaussian model.

FIG 3.2.1 is used as a reference to derive motion model for the robot

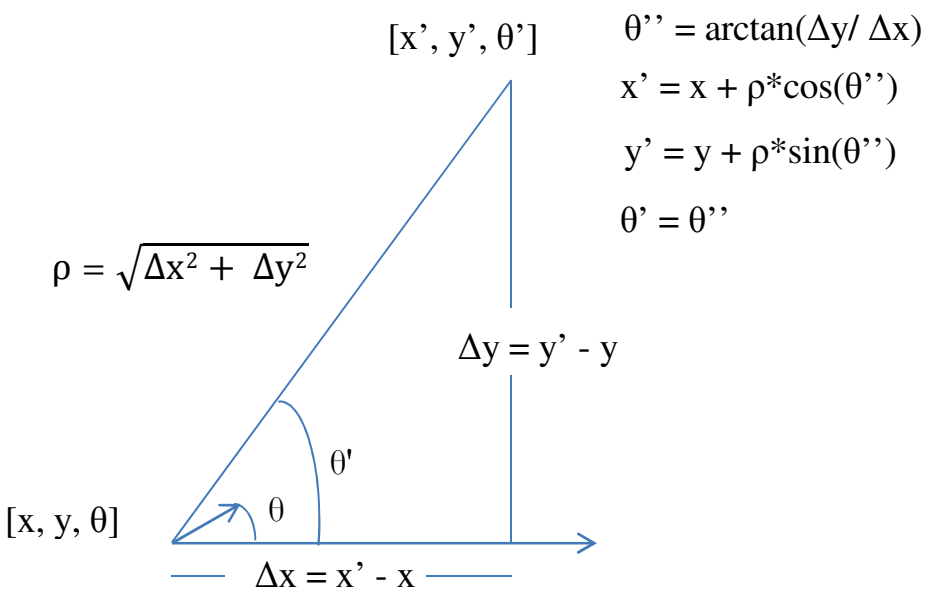

FIG 3.2.1 Motion Model

For a robot to move $[\Delta x, \Delta y]$ from an initial pose of $\chi = [x, y, \theta]$ to $\chi' = [x', y', \theta']$ it first rotates by $\Delta\theta = \theta'' - \theta'$ and then moves a distance $\rho = \sqrt{\Delta x^2 + \Delta y^2}$.

$\Delta x = x' - x, \Delta y = y' - y$ and $\theta'' = \arctan(\Delta y/\Delta x)$.

The final pose of the robot is given by

$$[x', y', \theta'] = [x + \rho * \cos(\theta''), y + \rho * \sin(\theta''), \theta''] \qquad \text{Eq. 3.1}$$

While performing translational motion, we have errors with respect to orientation and the distance traveled. The orientation error is because of the drift during linear motion. As a result the robot's final orientation is

$$\theta' = \theta + \Delta\theta + N(\Delta\theta, \sigma_{rot}) \qquad \text{Eq.3.2}$$

Since the robot moves in discrete steps and not in just one jump we need to add Gaussian noise to these steps. The movement in discrete steps is an approximation because it is analytically difficult to model continuous movement. The discrete number of steps κ is chosen to be large enough to approximate the continuous motion but low enough not to be computationally intensive. Since the step size we have chosen is so small we do not perform this computation in the code at all.

Pseudo code for Motion model for "Particle Set" is

```
ρ = Distance to move
for(i = 1 to M)
N_trs = normDist(ρ, σ_p)
      θ[i] = θ[i] + N(θ_m, σ_θ)
      x[i] = x[i] + N_trs*cos(θ[i])
      y[i] = y[i] + N_trs*sin(θ[i])
```

Pseudo Code for (Translation) Motion Model

Validity of the motion model and its results are interpreted in Chapter V. But, Figs. 3.2.2 and 3.2.3 show results for a variation of angular and/or linear displacements.

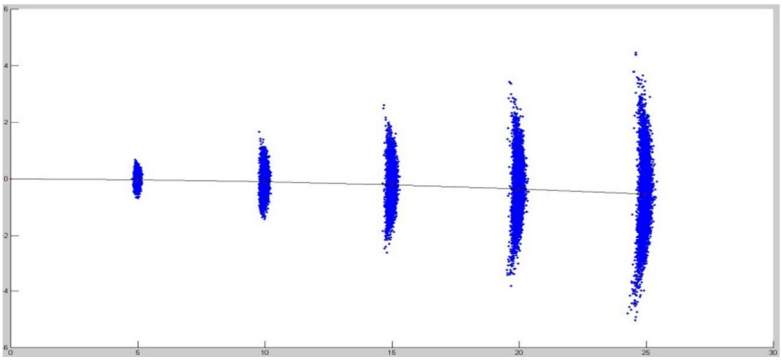

FIG 3.2.2 More angular displacement, less linear displacement

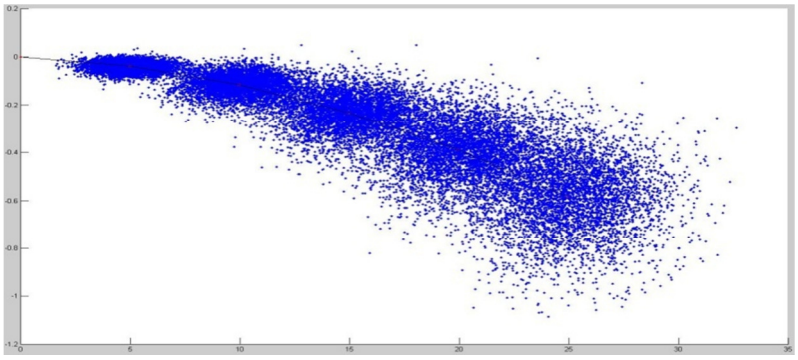

FIG 3.2.3 More angular & linear displacement

3.3 Update Model

Once the robot completes its action of moving to a new position after moving a distance (ρ), its sensors try to detect landmarks around it. At this point, the robot has two readings: Odometry (the distance it has moved) and the sensor (distance to obstacles and the robot orientation) readings. Based on sensor readings the robot updates it position. With every reading, the sensors detect robot orientation (θ) and distances (ρ_l, ρ_r, ρ_c) to the obstacle in front of left, right and center sensors respectively.

Let,

pose of the robot be $\chi_t = [x_r, y_r, \theta_r]$

co-ordinates of the obstacle $O_i = [x_i, y_i] \quad i = left, right, center$

sensor measurement $Z = [\theta, \rho_l, \rho_r, \rho_c]$

Based on sensor measurements, co-ordinates of the obstacle/s as calculated.

$$\left.\begin{aligned} O_{left} &= \begin{bmatrix} x_r + \rho_l * cos(\theta_r + \pi) + e_\rho \\ y_r + \rho_l * sin(\theta_r + \pi) + e_\rho \end{bmatrix} \\ O_{right} &= \begin{bmatrix} x_r + \rho_r * cos(\theta_r - \pi) + e_\rho \\ y_r + \rho_r * sin(\theta_r - \pi) + e_\rho \end{bmatrix} \\ O_{center} &= \begin{bmatrix} x_r + \rho_c * cos(\theta_r) & + e_\rho \\ y_r + \rho_c * sin(\theta_r) & + e_\rho \end{bmatrix} \end{aligned}\right\} \quad (Eq. 3.3)$$

e_ρ is the noise or the error in measuring the distance to an obstacle.

We need to calculate the probability of pose for the next state given the current state and the sensor readings, i.e.$P(\chi_{t+1}|\chi_t, Z)$, closer the particle is to the actual robot position greater the weight.

$$P(\chi_{t+1}|\chi_t, Z) = \frac{1}{\sqrt{2\pi(\sigma_\rho)^2}} e^{-\frac{(\rho_i-\rho)^2}{2(\sigma_\rho)^2}} \qquad \text{(Eq. 3.4)}$$

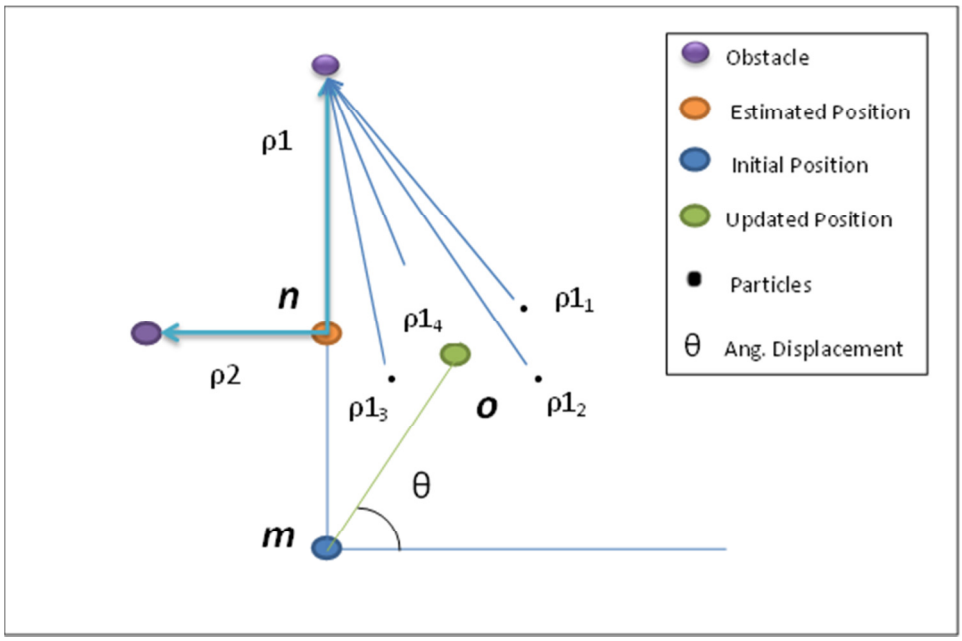

FIG 3.3.1 Update Model

Update model can be explained using FIG 3.3.1

Robots initial position "**m**"

After moving a step it thinks it is in position "**n**"

But, the robot is actually at position "**o**".

From position "n", robot senses obstacles at distances .ρ1& ρ2 .

As a result we have (an estimate of) obstacle(s) co-ordinates

We pass the particle set through the motion model.

(i) We measure obstacle - particle distance for every particle corresponding to robot obstacle distance (e.g. $\rho 1_1, \rho 1_2, \ldots$ correspond to $\rho 1$).

(ii) We calculate weight of every particle using Eq. 3.4.

(iii) Using weighted average we calculate updated co-ordinates.

Steps (i) – (iii) are repeated for the next obstacle.

3.4 Resample

With Particle filters, after a few iterations the weights of certain particles become so low, that that they do not contribute in updating χ. As a result the particle set needs to be resampled. This means that some particles with higher weights get duplicated. A variety of sampling algorithms are available. In this case "Select with Replacement" is used. It is one of the simplest and fastest algorithms. The algorithm selects a particle with probability equal to its weight. It does so by calculating cumulative sum of weights of all the particles and sorting N random numbers uniformly distributed with [0, 1]. The number of times a random number appears in the cumulative sum represents the number of times the particles that are propagated to the next stage or how many times a particle is represented in the final particle set. It works based on the fact that a particle with small weight would have small cumulative sum interval and therefore would have a less chance of being selected. Similarly for a particle with high weight the cumulative sum range would be high. So, particles with high weights have a higher probability of being selected.

```
#W = Weights of Particles
#N = Number of Particles

C = cumulativeSum(W)
r = random(N+1)
R = sort(r)
R(N+1) = 1; I =1; j = 1;
while(i<=N)
   if(R[i]<C[j])
      I[i] = j
      i = i + 1
   else
      j = j + 1
return I
```

Pseudo code for resampling algorithm

CHAPTER 4

SETUP

4.1 Hardware

Following components were used to setup the robot.

- iRobot Roomba [20, 49] is used as an autonomous vehicle to move around and map the environment. FIG 4.1.1. Roomba is a robot manufactured by iRobot. It was initially introduced as an automated robot to clean floors indoors. It comes preloaded with a set of algorithms to move around in a predetermined path. Roomba has quite a few built in sensory and actuation capabilities. Table 4.1.1 lists the number of sensory/input and actuation/outputs of the iRobot Roomba.

Sensors/Input		Actuation/Outputs
Wheel encoders(2)	Buttons(3-4)	Drive wheels(L & R[3])
Bump Sensors (2)	Dirt-Detection(1-2)	
IR Wall Sensor (1)	IR Receiver(-255)	Cleaning Motors (3)
Cliff sensors(7)	Electrical(5)	

Table 4.1.1 Sensory and Actuation capabilities of Roomba

With growing popularity of these robots iRobot released the serial API[4] for Roomba called Roomba Serial Command Interface or Roomba SCI [11]. Roomba SCI is a serial protocol that allows users to control a Roomba through its external serial port. These protocols have been integrated into quite a few software drivers written in C++ [46], Python [47], JAVA [48] and MATLAB [19]. With time Roomba has found many applications more significantly in robotics research and education community.

[3] Left and Right

[4] Application Programming Interface

iRobot-based software support along with compatible third party hardware interfaces have helped Roomba modify, both programmatically and physically, to perform a variety of tasks, ranging from controlling Roomba with a cellphone, using Roomba as a MIDI synthesizer, using Bluetooth with Roomba for wireless control, etc. [50]

- 3, PING Ultrasonic Sensors(PiUS)[21] placed on the robot's edges to measure distances left, right and center. FIG 4.1.2
- Hm55b compass module (HCM) [22] to measure angular displacement. FIG 4.1.3
- Arduino UNO Microcontroller [23] to program and measure the readings off the sensors. FIG 4.1.4
- Standard USB printer cable is used to connect Arduino Uno microcontroller with a computer.
- Serial cable to communicate with Roomba from MATLAB. FIG 4.1.5
- And, "KeyspanUSA-19HS" Serial to USB Converter. FIG 4.1.5

Final robot setup with all the sensors and complete connection is shown in FIG 4.1.6 and FIG 4.1.7

FIG 4.1.1 iRobot Roomba

FIG 4.1.2 Ping Ultrasonic

FIG 4.1.3 HM55B Compass Module

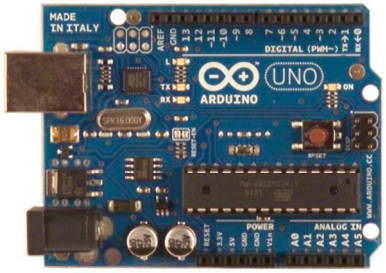

FIG 4.1.4 Arduino Uno Microcontroller

FIG 4.1.5 Roomba Serial and USB cables

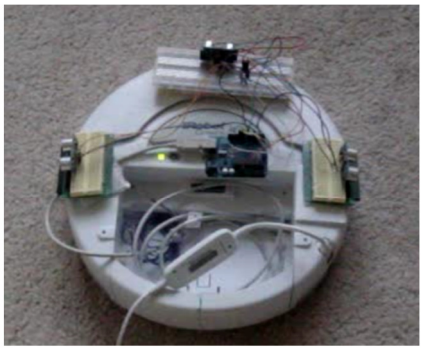

FIG 4.1.6 Final Roomba Setup

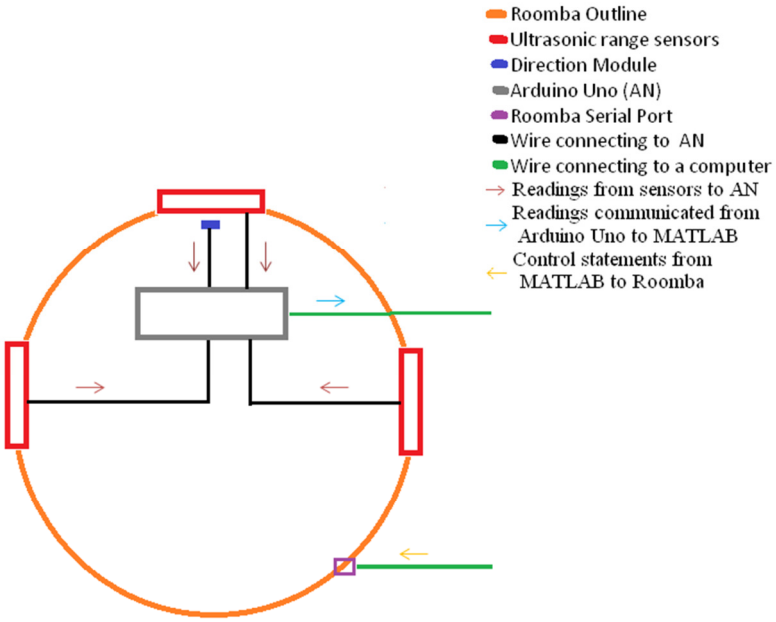

FIG 4.1.7 Outline of Final Roomba Setup

4.2 Evaluating System Constants

A series of measurements were carried out to determine system constants. Mean and Standard Deviation values were calculated over a sufficient sample size[5] for external and internal sensors for linear and angular displacement:

Robot was issued a command to move a fixed step size (5cm) and at the end of every command sensors were queried for the linear displacement.

- Robot's (internal) sensor for linear displacement: Robot was issued a command move a fixed step of 5cm. and at the end of every command execution, the internal sensor was queried (over a sample size of 40)

- External Sensor (PiUS) for linear displacement: There is no external sensor for measuring linear displacement. So, at the end of every command distances were measured manually. (over a sample size of 30)

Robot was issued a command to turn 90^o and at the end of every command sensors were queried for the angular displacement.

- Robot's (internal) sensor for angular displacement: Robot was issued a command move 90deg and at the end of every command its internal sensor was queried for the angular displacement (over a sample size of 40).

- External Sensor (HCM) for angular displacement: Robot was issued a command move 90deg and at the end of every command HCM was queried through Arduino Microcontroller for the angular displacement (over a sample size of 30).

Tables 4.2.1 and 4.2.2 show the values of "Mean" and "Standard Deviation" (S.Dev) for external and internal sensors respectively. The mean is for a 5cm linear and 90^o angular displacement. The robot always moves in steps of 5cm and turns 90^o (since the walls of the maze are at right angles).

[5] Number of times the experiment was carried out or a measurement was made

	Linear Disp (sample size 30)		Ang. Rotation (sample size 30)	
	Mean	**S.Dev**	**Mean**	**S.Dev**
External Sensor	5.48181	0.24318	90.52517	3.72526

Table 4.2.1 System Constants for external sensors

	Linear Disp (sample size 40)		Ang. Rotation (sample size 40)	
	Mean	**S.Dev**	**Mean**	**S.Dev**
Internal Sensors	5.58333	0.07914	91.02514	0.1581

Table 4.2.2 System Constants for internal sensors

Based on values "Mean" and "Standard Deviation", following observations can be made

- There is a bias in case of "Linear displacement" measurements. We see a mean of approximately 5.5 and a bias of 10% for both external and internal sensor measurements.
- "Standard Deviation" values for "Linear displacement" in both cases is low. This is considered a good sign since robot moves a consistent fixed distance in spite of the bias.
- "Mean" values for "Angular Rotation" are fairly consistent with the issued command to rotate by 90 deg., in both cases.
- "Standard Deviation" value for "Angular Rotation" in case of external sensor is quite high when compared to internal sensor.

4.3 Implementation

The work uses MATLAB, C and C# to communicate between the different components of the robot. Main components (programming/implementation wise) of the Roomba robot are

- Arduino Microcontroller
- Serial Port reader:

- Roomba communicator:

- Core module

Implementation of each of the components is explained next

Arduino Microcontroller

This component is an interface between hardware and software. The Arduino Microcontroller provides a programmatic door to control, modify and interpret the behavior for compatible hardware (in our case "sensors"). The hardware in our case being Ultrasonic Range and Direction Sensors, while the software is the code that interprets the raw signals received from these sensors. The code is implemented mainly in C. Once the readings are interpreted, they are output on a COM port to be read by a different component. The code that is uploaded into the microcontroller is added in Appendix.

Serial Port reader

The Serial Port Reader (SPR) is an executable that reads values off the COM port that Arduino Microcontroller writes. The SPR reports values of the direction and obstacles. Direction being the angle robot makes to geographic North and obstacles being the distances "Ultrasonic Range Sensors" placed on left, right and forward edges detect, to their nearest and respective obstacles. These values are reported back to the main module. This code is implemented in C#.

The executable takes "COM Port name", "Baud rate" and "iterations" as command line parameters and returns an array - direction and 3 distance values to the calling component.

The C# code Serial Port Reader is added in Appendix.

Roomba communicator

This module communicates with the Roomba. It is used to issue commands to move or turn. It is also used to query the Roomba for linear and angular displacements. This module is developed in MATLAB by Dr. Joel Esposito, United States Naval Academy [19].

e.g. The commands used from this API are

```
RoombaInit(comPortR)
travelDist(serPort,spd,dist)
DistanceSensorRoomba(serPort)
turnAngle(serPort,spd,angl)
AngleSensorRoomba(serPort)
```

`RoombaInit` sets up the connection between MATLAB and Roomba through a COM Port `comPortR`.

`travelDist` tells the robot to move a distance of `dist` meters at a speed of `spd` meters per second.

`DistanceSensorRoomba` queries the robot for the distance travelled since last invocation of this command.

`turnAngle` tells the robot to turn an angle of "`angl` deg". at a speed of "`spd` meters per second".

`AngleSensorRoomba` queries the robot for the change in angular orientation of Roomba since last invocation of this command.

Core module

This is the module where SLAM is implemented. This module issues control statements and communicates between the above mentioned components. All the decisions of Roomba either moving in a straight direction or turning by an angle of 90 degrees are made here including the issuing of commands to SPR, sensing obstacles, updating estimated position and if required, resampling the particle set.

```
(01)   get sensor readings
(02)   while(robot explores the maze)
(03)    if robot can move ahead
(04)      move
(05)     calculate obstacles based on sensor readings
```

```
(06)    update Particle Set
(07)   calculate particle weights
(08)  update robot position
(09)  else
(10)    turn the robot by 90 degrees.
(11)      get sensor readings
(12)      if particle set below threshold
(13)         resample
```

Pseudo Code for core module

`readSerialPort(execFile, comPortA, baudRate, iter)` is the function call that implements line (1) and (11) in the pseudo code. `execFile` is the name of SPR.

`travelDist(serPort, spd, dist)` is the function call to implement line (04) in pseudo code. This function makes the robot travel a certain distance (`dist`) at some speed (`spd`).

`calcObst(sensorReading, offset, currPose)` returns obstacles(for line [05] of pseudo code) based on the current sensor readings and current pose (`currPose`) of the robot. Offset is the distance from center of the robot to the sensors (placed on edges of the robot).

`transM(partSet, rho, stdLinDisp, stdAngDisp, steps)` for line (06) is for passing the particle set through motion model to get PDF. This function returns updated particle set as return value. Only the co-ordinates and orientation of the particles are updated in this function and not the weights.

Calculation of weights and updating of robots position based on obstacles(`obst`) and current pose(`currPose`) is done through `[partSetupdPose] = calcWeights(currPose, obst, partSet)` for lines (07) & (08) in pseudo code. This function returns particle set with updated weights and updated pose (`updPose`)

`turnAngle(serPort, spd, tAngle)` makes the robot turn by an angle(`tAngle`) at some speed(`spd`). This is for line (10) of pseudo code.

If value/s of weights become so low that they do not contribute to updation process, the particle set is resampled. `partSet = resample(partSet)` implements this function for line (13) of pseudo code.

CHAPTER 5

RESULTS AND INTERPRETATION

The figures used to discuss the results of different models and simulations have been generated using MATLAB.

5.1 Prediction Model

The motion model helps us understand the effect of linear and angular "Standard Deviation" about the mean on samples in a particle set.

Mean and Standard Deviation of the external sensors for linear displacement are respectively, $\rho_{\mu_{sensor}} = 5.48181$ $\rho_{\sigma_{sensor}} = 0.24318$.

Mean and Standard Deviation of the external sensors for angular displacement are respectively, $\theta_{\mu_{sensor}} = 90.52517$ $\theta_{\sigma_{sensor}} = 3.72526$.

These are the same values we see in Tables 4.2.1.

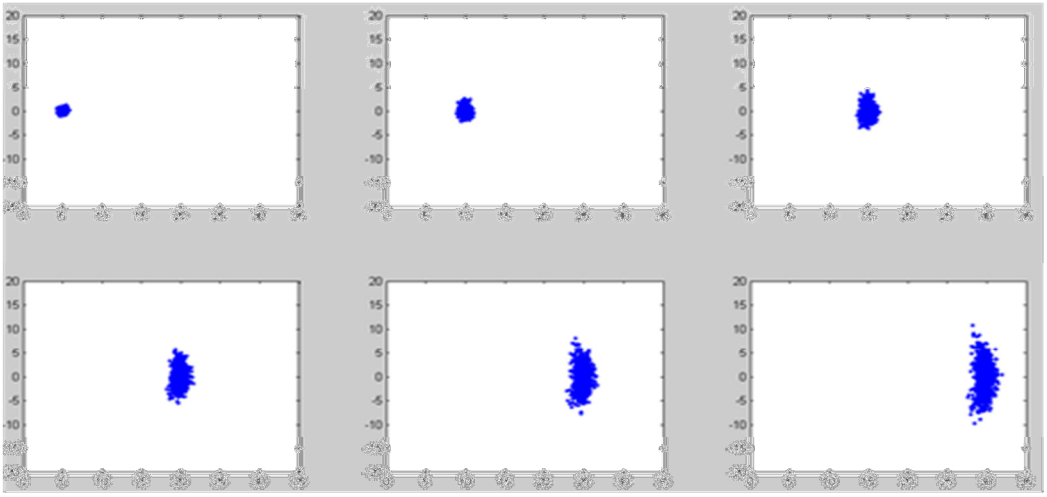

FIG 5.1.1 Motion model based on external sensor values

We see that with more linear movement, particles move further apart. This is the expected behavior. Higher the value of $\theta_{\sigma_{\text{sensor}}}$ higher the dispersion of particles.

Although the system values of "Mean" and "Standard Deviation" for the robot will never be used in a motion model. The plot is more for validation.

Mean and Standard Deviation of internal sensors for linear displacement are respectively, $\rho_{\mu_{\text{robot}}}$ = 5.5833 $\rho_{\sigma_{\text{robot}}}$ = 0.07914.

Mean and Standard Deviation of internal sensors for angular displacement are respectively, $\theta_{\mu_{\text{robot}}}$ = 91.02514 $\theta_{\sigma_{\text{robot}}}$ = 0.1581.

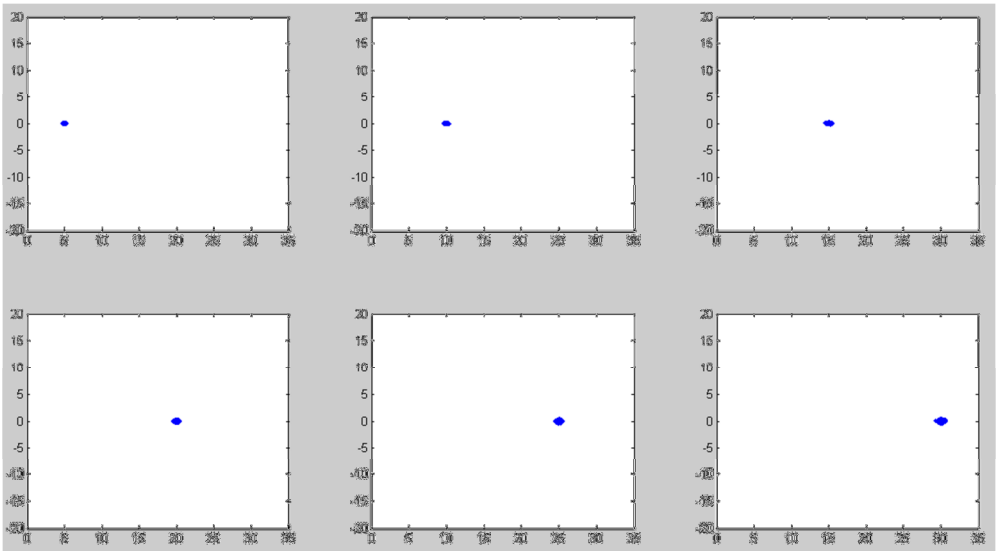

FIG 5.1.2 Motion model based on robot's sensor values

Since values for "Standard Deviation" are much smaller we do not see much of particle dispersion in FIG 5.1.2.

So, the motion model works correctly.

5.2 Update Model

The update model comes into play whenever sensors detect an obstacle. Based on position of an obstacle and weights of the samples/particles, we need to update current pose of the robot. This is seen from the output of the MATLAB code, once it is started.

Following is a part of the output

```
OrigPos   [2.00   2.00]

EstdPos   [3.03   2.15]

UpdtPos   [2.90   1.99]
```

FIG 5.2.1 Update of position

"EstdPos" is the estimated position by the robot based on robot sensors. "UpdtPos" is the updated position based on external sensor readings and the weights of all particles.
"OrigPos" is the previous position of the robot.

5.3 Simulations

This section will, in detail explain and interpret the process of mapping.

"createGrid" function is used to generate a random grid. The initial position of the robot is set to [2, 2] and it is assumed that robot makes an angle of 90 degrees with the X-axis. These assumptions just help initiate the process. Once the simulation starts robot iterates and alternates between sensing and moving/turning. At the end of each iteration, robot updates its position. Simulation plots the estimated robot path (the path robot thinks is right) and updated path (a more accurate path based on the SLAM process).

The values used for the simulation were $\rho_{\sigma_{\text{robot}}} = 0.4$ and $\theta_{\sigma_{\text{robot}}} = 0.5$.

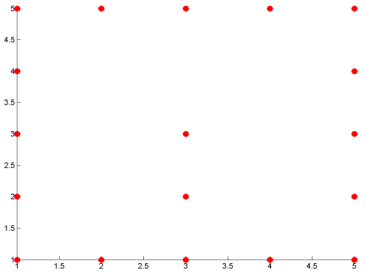

FIG 5.3.1 Random Grid (generated)

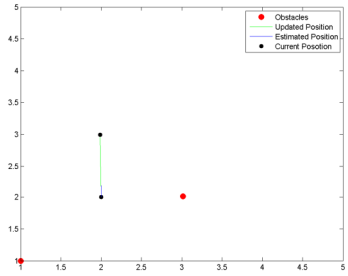

FIG 5.3.2 Step 1

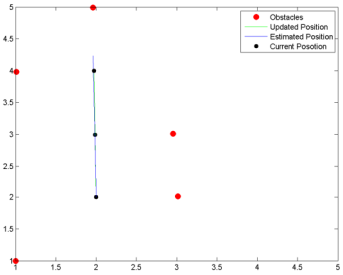

FIG 5.3.3 Step 3

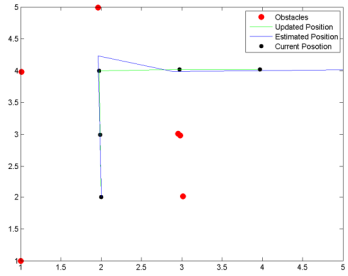

FIG 5.3.4 Step 5

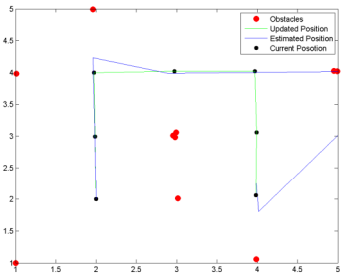

FIG 5.3.5 Step 8

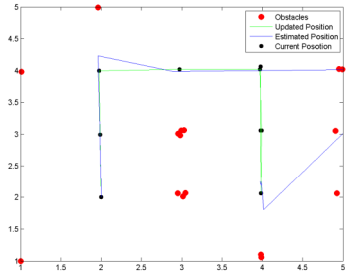

FIG 5.3.6 Step 11

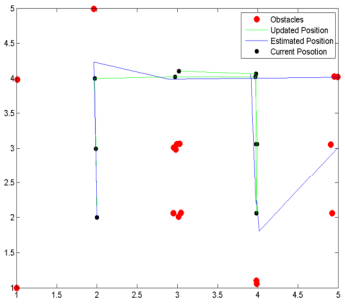

FIG 5.3.7 Step 13

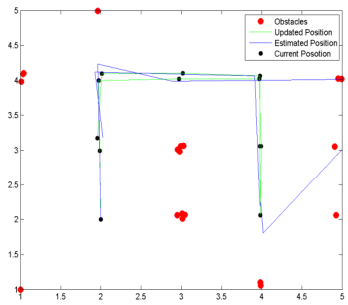

FIG 5.3.8 Step 16

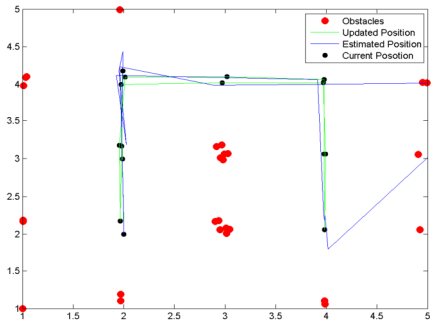

FIG 5.3.9 Step 20

With the path traversed in FIG 5.3.9 and the actual grid in FIG 5.3.1, the simulation (within its limits) works correctly. In "Steps 1-5" (FIG 5.3.2-4) the robot behaves correctly. We see that in Step 8 & 11 (FIG 5.3.5 & FIG 5.3.6 respectively) a bad reading from robot sensor can make the robot behave erratically. But, because of low variance values the paths always converge. The system in this sense is resilient to one time inaccurate reading. In subsequent steps 13-20 (FIG 5.3.7-9) the robot moves around the maze and maps it successfully.

As a sanity check, simulation used values for a perfect system by setting variance to 0. Estimated and Updated paths were exactly the same.

Now, we know that the simulator behaves as expected we input the system constants from Tables 4.2.1 and 4.2.2 Here are the results.

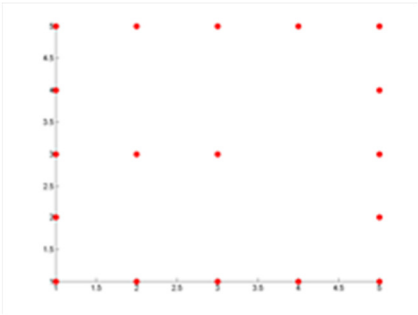

FIG 5.3.10 Random Grid (generated)

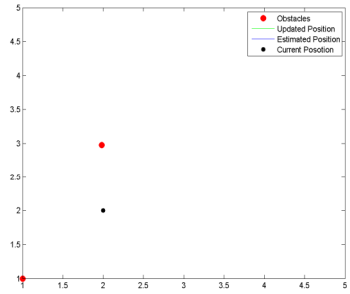

FIG 5.3.11 Step 1

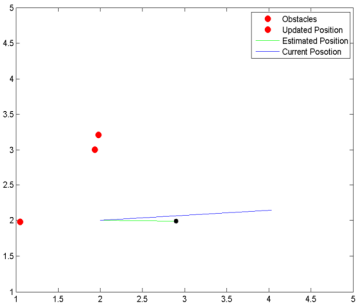

FIG 5.3.12 Step 2

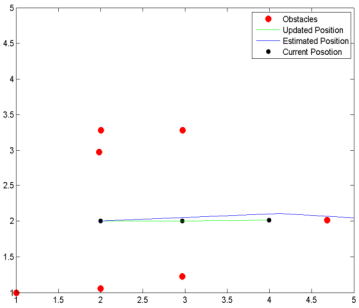

FIG 5.3.13 Step 4

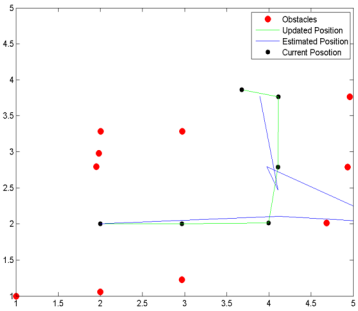

FIG 5.3.14 Step 8

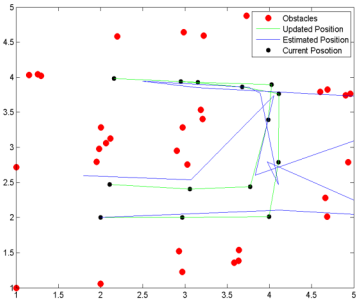

FIG 5.3.15 Step 20

FIG 5.3.10 was a random 2x2 grid. Subsequent figures are snapshots of robot during maze traversal at different stages. The robot begins by positioning itself at [2, 2] and placing the obstacles correctly at [1, 2] and [2, 3] as seen in FIG 5.3.11 and FIG 5.3.12. It's from this point onwards that the robot

goes hay wire. In step 4 FIG 5.3.13 and onwards we see that the robot's "estimated path" (blue line) always tries to catch up to the "updated path" (green line). This is seen in FIG 5.3.14 and FIG5.3.15. Eventually both positions are similar (co-ordinate wise).
If we look up at the paths and the grid we can make out that the robot has mapped the grid, but, just barely.
If we try to interpret this erratic behavior, we could argue that it is because of the high ratio between Standard deviations of external sensors and that of robot's own sensors. This ratio is as high as 25. This means that one of the sensors is highly inaccurate.

If we try to analyze the constants for the robot i.e. $\rho_{\mu_{\text{robot}}}$ = 5.5833, $\rho_{\sigma_{\text{robot}}}$=0.07914, $\theta_{\mu_{\text{robot}}}$= 91.02514, $\theta_{\sigma_{\text{robot}}}$= 0.1581, we see that measurements did not vary much about their respective "means". If this were considered to be true, one can conclude that every time in executing either command of linear displacement or angular rotation, the robot's motor functioned exactly the same each time and there were no unforeseen circumstances like wheel slippage, motor not behaving correctly, etc. Or even if there was wheel slippage, it happened almost every time and had almost the same effect on almost all measurements. This is highly unlikely. So, either internal robot sensors are not being able to measure correctly or external sensors are inaccurate.

The other case where external sensors are inaccurate would mean that the manual measurements done earlier are faulty. This case is highly unlikely since measurements made by hand were repeated many times to make sure they were correct. This leaves us with one conclusion that the robot's internal sensors inaccurately report the readings that cause the robot to behave erratically.

One of the assumptions was that the robot's angular displacement would be "almost" in place. But experiments have revealed that there is quite a bit of linear displacement even when the robot is commanded to "just" turn. FIG 5.3.16 is an overlap of initial and final positions, after robot was commanded to turn.

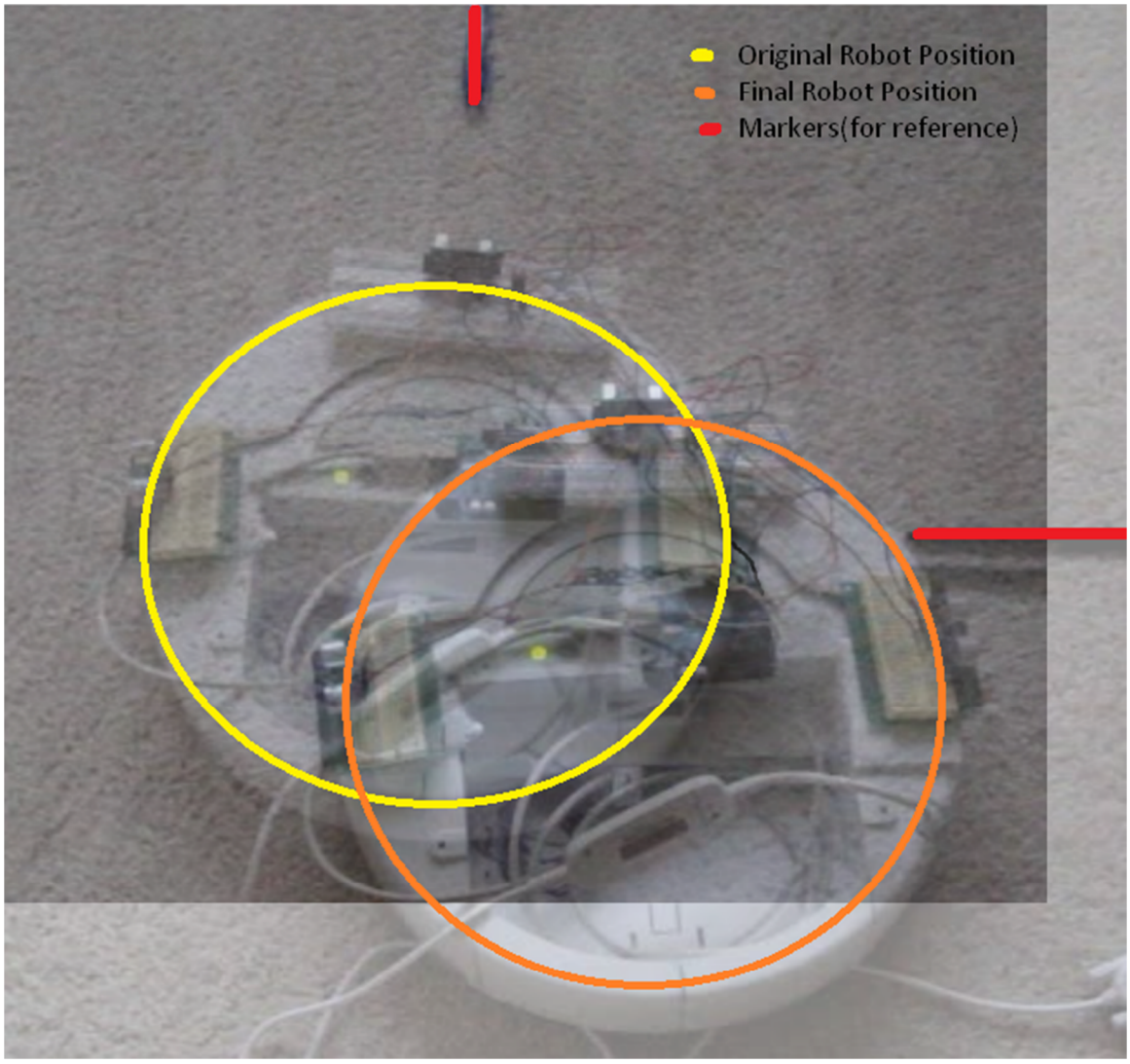

FIG.5.3.16 Linear Displacement along with Angular

Since this linear displacement is random in both direction and magnitude no model can be built around it. As a result this behavior cannot be assimilated in an implementation.

The simulation actually assumed that rotation was in place. But just the internal sensors were inaccurate enough to make the robot behave erratically. Even if we were to introduce the linear shift somehow, “iRobot Roomba’s” problems would be compounded. With this being the ground reality, any simulation and/or a robot in an actual environment would definitely do much worse. This could however work if, there were minimal robot rotation.

5.4 Blender Simulations

Blender is an open source 3d animation and game development suite. The simulation creates of a random NxN maze.

FIG 5.4.1 shows Blender interface while FIG 5.4.2 shows the Blender interface once (random) maze is created. The randomly generated maze is similar to the maze in physical setup.

Red cubes are walls of the maze.

White square is floor of the maze.

Green sphere is the robot.

Blue square means that robot has visited that grid.

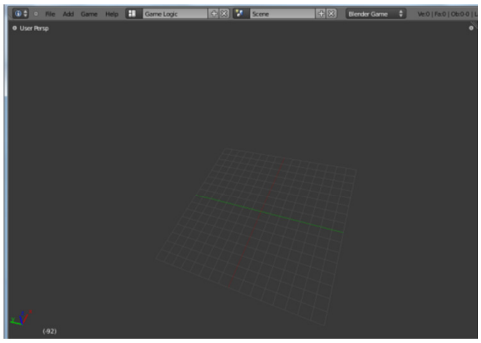

FIG 5.4.1 Blender initial state

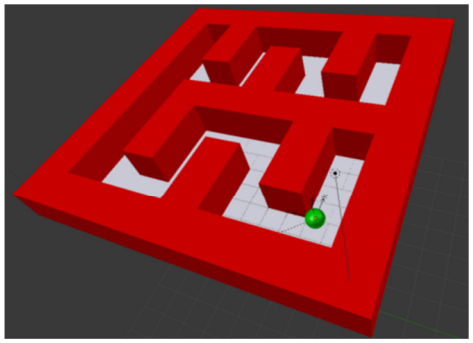

FIG 5.4.2 Maze initial state

FIG 5.4.3 shows the setup required, in terms of adding sensors, data structures and making necessary connections for proper working of the "virtual" robot.

FIG 5.4.4 shows connections, sensors and parenting the virtual robot with the actual robot(sphere).

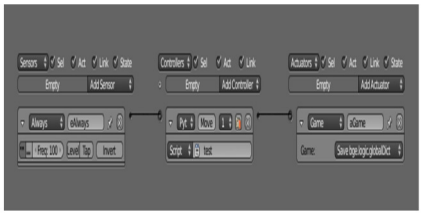

FIG 5.4.3 Setup

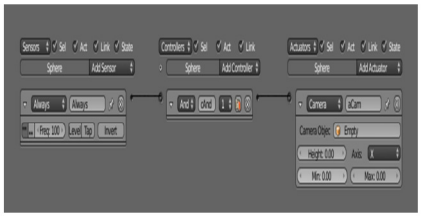

FIG 5.4.4 Connections for robot

FIG 5.4.5 shows the robots position and state after its 1st step. We see that the white grid has turned blue, meaning that the grid is visited. Similarly in FIG 5.4.6 we see that the robot has returned to its original position and the entire grid is blue, meaning that the grid is mapped.

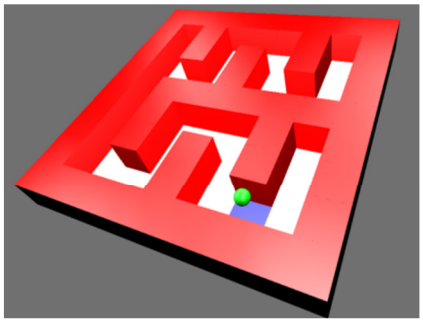

FIG 5.4.5 Robot's 1st move

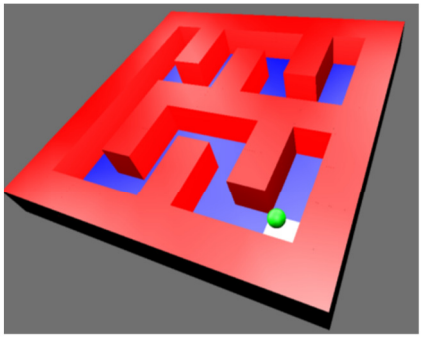

FIG 5.4.6 Maze used with robot

5.5 Roomba results

The robot was placed in an actual maze to see the performance of Roomba. The sample maze is shown in FIG 5.5.1. Maze traversal path (black dots) and obstacle detection (red dots) by Roomba are shown in FIG 5.5.2-6. FIG 5.5.2 is the Step 1 for maze traversal where we see robot detecting 3 obstacles. FIG 5.5.3 is Step 13 where Roomba has moved in an almost straight path and is about to turn left. FIG 5.3.4 is Step 14 where Roomba has turned left and the newly detected obstacles are in "green". We see that these obstacles align with the obstacles detected in previous step. This confirms that turning of robot works correctly. The robot continues to explore the maze FIG.5.5.5 and FIG 5.5.6 where Roomba makes a left turn. Again, obstacles (green dots) align with previously detected obstacles. FIG 5.5.8 is the final mapped (and rotated) result. We see that the mapped maze FIG 5.5.8 is almost comparable to the maze in FIG 5.5.7 or FIG 5.5.1.

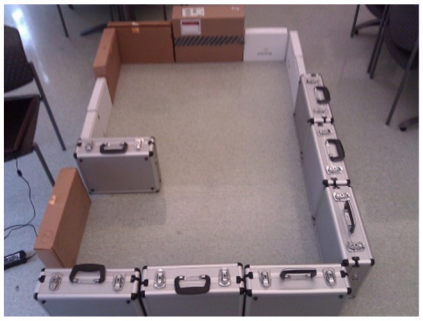

FIG 5.5.1 Maze used with robot

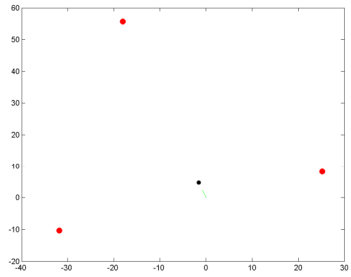

FIG 5.5.2 Step 1

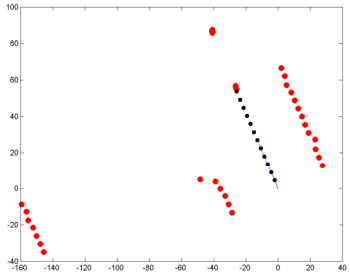

FIG 5.5.3 Step 13

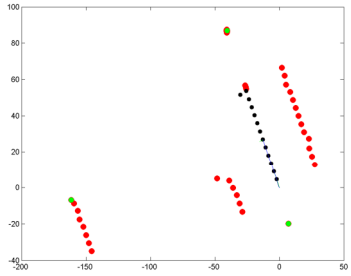

FIG 5.5.4 Step 14

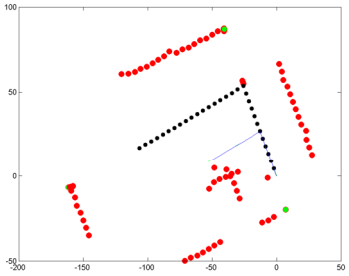

FIG 5.5.5 Step 31

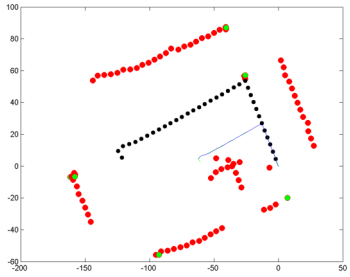

FIG 5.5.6 Step 37

FIG 5.5.7 Maze

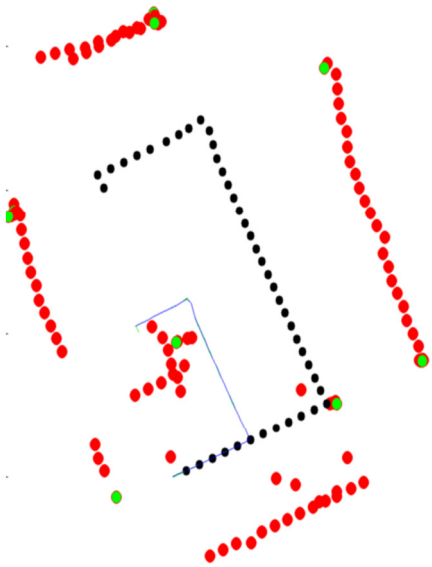

FIG 5.5.8 Final Mapped Maze

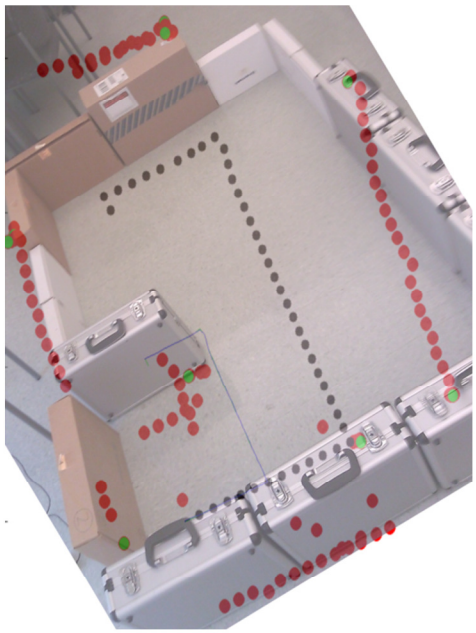

FIG 5.5.9 Merged Mapped Mazes

To see the mapping more clearly, actual maze FIG 5.5.7, and mapped maze FIG 5.5.8 have been merged, scaled and rotated in FIG 5.5.9.

We see that mapping is almost accurate. The stray “Red Dots” are mainly because of limitations of the robot itself, like “non in-place” angular displacement and less accurate robot’s internal sensors. Apart from these, the other contributing factor was the cables hanging from the sides. When robot turned left (the second time), left sensor detected some of cables lying around. This sensing resulted in plotting them as obstacles.

5.6 Comparison of Roomba and simulator

For a more accurate comparison, the Particle Filter simulator was run on a maze similar (in dimensions w.r.t to number of rows and columns) to the maze used with "iRobot Roomba" (the experimental setup – Section 5.5.5). In simulation, the robot was placed in a position similar to the one in the actual maze. The maze used with Roomba and the simulated maze are shown in FIG 5.6.1 & 5.6.2 respectively. FIG 5.6.3 demonstrates that FIG 5.6.1 and 5.6.2 are similar in dimension and setup

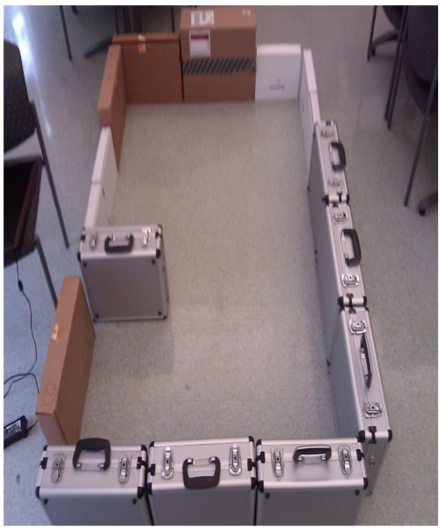

FIG 5.6.1 Maze used with Roomba

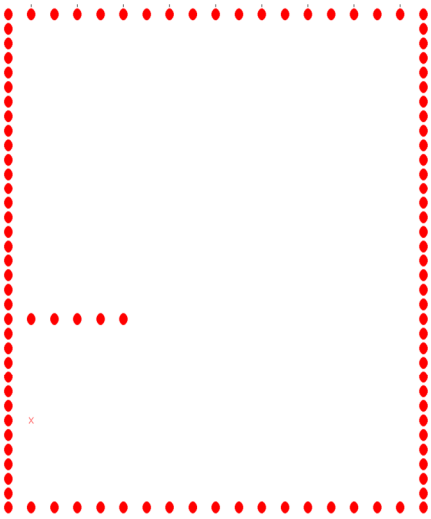

FIG 5.6.2 Maze used simulator

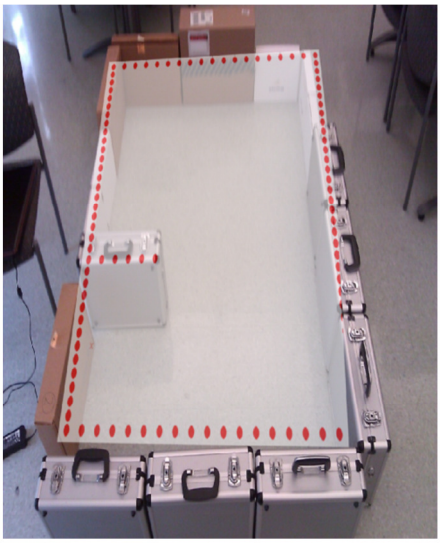

FIG 5.6.3 Overlapped mazes
(actual and simulated)

The main aim of this comparison is to check if the simulator and the robot behave similarly given (approximately) the same environment (i.e. the maze and system constants) and the same initial condition (approximately identical point of origin to map the maze). FIG 5.6.4 is the same mapped maze as in FIG 5.5.8. FIG 5.6.5 is the maze (in FIG 5.6.2) mapped by the simulator.

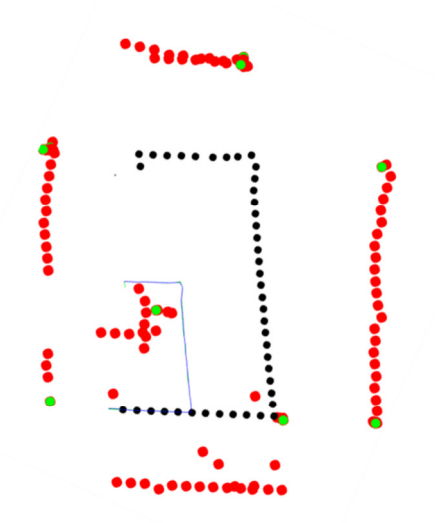

FIG 5.6.4 Actual Maze mapped by iRobot Roomba

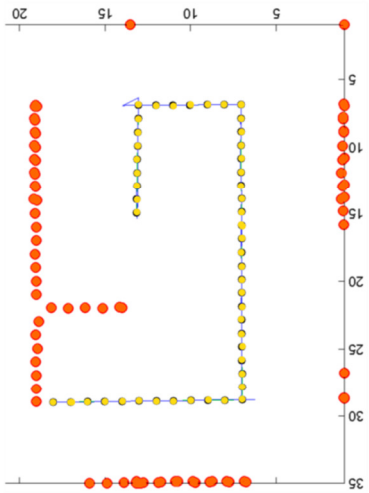

FIG 5.6.5 Simulated maze mapped by the simulator

The two mapped mazes (FIG 5.6.4 and FIG 5.6.5) are overlapped in FIG 5.6.6 to verify the similarity in mapping.

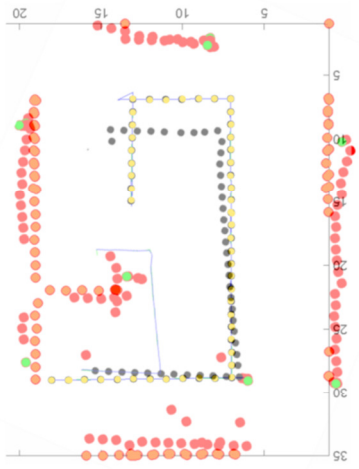

FIG 5.6.6 Overlapped mapped mazes by simulator and iRobot Roomba

We see that walls of the maze (red and orange dots) and path taken by the robot (back and yellow dots) approximately coincide. There are a few differences in the maps.

To quantify the differences, the obstacles mapped using the robot and the simulator were compared, i.e. sufficient number of samples (44) were collected for the distance between an obstacle mapped by the simulator and the robot. The mean and standard deviation for the collected samples is,

$$\mu_{distance} = 5.208\ cm \qquad \sigma_{distance} = 1.029\ cm$$

This means that the results of the simulation and the robot are **identical within *5.208* cm accuracy**. This difference in mapping of obstacles and path taken by the robot in either of the cases could be a result of differences in the dimensions of the maze and/or initial positioning of the robot in the simulator and actual maze and/or the fact that the actual distance to the wall at which the robot (Roomba) makes a turn cannot be formulated to units for the simulator (with a certain accuracy).

Even with these constraints simulator did map the maze with reasonable accuracy. This means that both, simulator and the robot behave in a similar fashion.

CHAPTER 6

CONCLUSION

As far as mapping of the maze is concerned, with the given constraints, Roomba did it with reasonable accuracy. If not for the constraints of robot's internal sensors and "non in-place"[6] angular displacement Roomba would have mapped the maze with better accuracy.

In case of the robot simulation, mapping did complete but the results were just passable (notwithstanding the "linear shift" issue). This is attributed to the inaccurate internal sensors. With better sensors robot would not have problems in mapping the environment. This can be seen in the results of first simulation.

In case of Blender simulation, the main aim was to show that it is simple enough to implement the SLAM concepts even in the suite that was meant for 3D simulations or Game development. Blender is really helpful since it provides real life constructs of gravity, friction, viscosity in case of fluids, mass, torque, linear/angular velocity/ displacement, etc. With these constructs it can simulate realistic conditions for robotic behavior. It is helpful in cases where the environment is not known or the robot needs to be tested in extreme conditions. In cases where robot design needs to be tested this suite can prove helpful. Most importantly it is helpful in cases where a robot is not present at all, but we need to test the theory and mathematical models.

SLAM in conjunction with "Particle filter", can map an unknown environment with accuracy, provided we use high number of particles. High numbers of particles help in cases where obstacles/landmarks are rare. Particle filters are inherently suited for mapping unknown environments. Although observed, but not quantifiable, random movement as opposed to prioritized movement map the environment much quicker. This may be because the random movement tends to sense/observe the unseen obstacles more often than in the case of predetermined prioritized movement.

[6] The angular displacement commands resulted in erratic linear displacement

REFERENCES

[1] D. Hahnel, D. Fox, W. Burgard, and S. Thrun, “An efficient FastSLAM ¨algorithm for generating maps of large-scale cyclic environments from raw laser range measurements,” in Proceedings of the IEEE/RSJ Conference on Intelligent Robots and Systems (IROS), vol. 1, Las Vegas Nevada USA, 27-31 October 2003, pp. 206 – 211.

[2] http://www.blender.org/

[3] SLAM for Dummies: A Tutorial Approach to Simultaneous Localization and Mapping http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf

[4] Borenstein, J., Everett, B., and Feng, L., 1996, "Navigating Mobile Robots: Systems and Techniques." A. K. Peters, Ltd., Wellesley, MA, ISBN 1-56881-058-X, Publication Date: February 199

[5] Borenstein, J., Everett, H.R. ,Feng, L., and Wehe, D., 1996, "Mobile Robot Positioning: Sensors and Techniques." Invited paper for the Journal of Robotic Systems, Special Issue on Mobile Robots. Vol. 14, No. 4, April 1997, pp. 231-249.

[6] S. Thrun, “Robotic mapping: A survey,” in Exploring Artificial Intelligence in the New Millenium, G. Lakemeyer and B. Nebel, Eds. Morgan Kaufmann, 2002, to appear.

[7] H. Durrant-Whyte and T. Bailey, “Simultaneous localisation and mapping (slam): Part i the essential algorithms,” Robotics and Automation Magazine, June 2006.

[8] "Random walks, universal traversal sequences, and the complexity of maze problems," focs, pp.218-223, 20th Annual Symposium on Foundations of Computer Science (FOCS 1979), 1979.

[9] G. Welch and G. Bishop. An introduction to the Kalman filter. Technical Report TR 95-041, University of North Carolina, Department of Computer Science, 1995

[10] A Particle filter Tutorial for Mobile Robot Localization by Ioannis Rekleitis
http://www.cim.mcgill.ca/~yiannis/particletutorial.pdf

[11] Roomba Serial Command Interface (SCI) Specification, iRobot, 2006. [Online]. Available:
http://www.irobot.com/images/consumer/hacker/Roomba_SCI_Spec_Manual.pdf

[12] Michael Montemerlo, FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem

[13] L. Goncavles, E. di Bernardo, D. Benson, M. Svedman, J. Ostrovski, N. Karlsson, and P. Pirjanian, "A visual front-end for simultaneous localization and mapping," in Proc. of ICRA, apr 2005, pp. 44–49

[14] N. Ouerhani, A. Bur, and H. H¨ ugli, Visual attention-based robot self-localization, in Proc. of ECMR, 2005, pp. 8–13

[15] Arulampalam, Maskell, Gordon, Clapp: A Tutorial on Particle Filters for on-line Nonlinear / Non-Gaussian Bayesian Tracking, IEEE Transactions on Signal Processing, Vol. 50, 2002

[16] T. Bailey. Mobile Robot Localisation and Mapping in Extensive Outdoor Environments. PhD thesis, Univ. of Sydney, 2002.

[17] A. N¨ uchter, K. Lingemann, J. Hertzberg, H. Surmann, 6D SLAM for 3D mapping outdoor environments, Journal of Field Robotics 24 (8–9)(2007)

[18] iRobotCreate: Open Interface Specifications,
http://www.irobot.com/filelibrary/pdfs/hrd/create/Create%20Open%20Interface_v2.pdf

[19] Joel M. Esposito and Owen Barton, "MatlabCreate:Matlab Toolbox for the iRobot Create", 2008, www.usna.edu/Users/weapsys/esposito/roomba.matlab/

[20] iRobot Create Owner's Manual
http://www.irobot.com/filelibrary/create/Create%20Manual_Final.pdf

[21] PING)))™ Ultrasonic Distance Sensor (#28015), http://www.parallax.com/dl/docs/prod/acc/28015-PING-v1.3.pdf

[22] Hitachi HM55B Compass Module (#29123), http://www.crustcrawler.com/products/Nomad/docs/HM55BModDocs.pdf

[23] ArduinoBoardUno, http://arduino.cc/en/Main/ArduinoBoardUnoand http://arduino.cc/en/uploads/Main/arduino-uno-schematic.pdf

[24] Crowley J. L., Position Estimation for a Mobile Robot Using Vision and Odometry. Proc. 1992 IEEE Intl. Conf. on Robotics and Automation, 2588-2593, Nice, May 1992

[25] Dissanayake G., Newman P., Clark S., Durrant-Whyte H. F. and Csorba M., A Solution to the Simultaneous Localisation and Map Building (SLAM) problem. IEEE Trans. on Robotics and Automation, 17(3):229-241, June 2001

[26] Gordon N. J., Salmond D. J. andSmith A. F. M., Novel Approach to Nonlinear/Non Gaussian Bayesian State Estimation. IEE Proceedings F,Radar and Signal Processing, 140(2):107-113, April 1993.

[27] Dieter Koller, Joseph Weber, Jitendra Malik. Robust Multiple Car Tracking with Occlusion Reasoning. In Proceedings of ECCV (1)'1994. pp.189~196

[28] Michael Isard, Andrew Blake. Contour Tracking by Stochastic Propagation of Conditional Density. In Proceedings of ECCV (1)'1996. pp.343~356

[29] Michael Isard, Andrew Blake. CONDENSATION - Conditional Density Propagation for Visual Tracking. International Journal of Computer Vision, 1998: 5~28

[30] Dorin Comaniciu, Visvanathan Ramesh, Peter Meer. Real-Time Tracking of Non-Rigid Objects Using Mean Shift. In Proceedings of CVPR'2000. pp.2142~2142

[31] Changjiang Yang, Ramani Duraiswami, Larry S. Davis. Fast Multiple Object Tracking via a Hierarchical Particle Filter. In Proceedings of ICCV'2005. pp.212~219

[32]Kenji Okuma, Ali Taleghani, Nando de Freitas, James J. Little, David G. Lowe. A Boosted Particle Filter: Multitarget Detection and Tracking. In Proceedings of ECCV (1)'2004. pp.28~39

[33] S. Thrun, Robotic mapping: A survey, CMU-CS-02-111, February 2002, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

[34] C. Fruh, A. Zakhor, 3D model generation for cities using aerial photographs and ground level laser scans, in: Proceedings of the Computer Vision and Pattern Recognition Conference, CVPR '01, Kauai, Hawaii, USA, December 2001

[35] H. Zhao, R. Shibasaki, Reconstructing textured CAD model of urban environment using vehicle–borne laser range scanners and line cameras, in: Second International Workshop on Computer Vision System, ICVS '01, Vancouver, Canada, 2001, pp. 284–295.

[36] S. Thrun, D. Fox, W. Burgard, A real-time algorithm for mobile robot mapping with application to multi robot and 3D mapping, in: Proceedings of the IEEE International Conference on Robotics and Automation, ICRA 00, San Francisco, CA, USA, April 2000

[37] S. Thrun, M.Montemerlo, A. Aron, Probabilistic terrain analysis for high-speed desert driving, in: Proceedings of Robotics: Science and Systems,Cambridge, USA, June 2006

[38] V. Sequeira, K. Ng, E. Wolfart, J. Goncalves, D. Hogg, Automated 3D reconstruction of interiors with multiple scan-views, in: Proceedings of SPIE, Electronic Imaging '99, The Society for Imaging Science and Technology/SPIE's 11th Annual Symposium, San Jose, CA, USA, January 1999

[39]A. Georgiev, P.K. Allen, Localization methods for a mobile robot in urban environments, IEEE Transaction on Robotics and Automation (TRO) 20 (5) (2004) 851–864.

[40] P. Allen, I. Stamos, A. Gueorguiev, E. Gold, P. Blaer, AVENUE: Automated site modelling in urban environments, in: Proceedings of the Third International Conference on 3D Digital Imaging and Modeling, 3DIM '01, Quebec City, Canada, May 2001.

[41] Y. Chen, G. Medioni, Object modelling by registration of multiple range images, Image and Vision Computing 10 (3) (1992) 145–155

[42]N. Gordon, D. Salmond, and A. F. M. Smith, "Novel approach to non-linear and non-Gaussian Bayesian state estimation," Proc. Inst. Elect.Eng., F, vol. 140, pp. 107–113, 1993.

[43] M. Pitt and N. Shephard, "Filtering via simulation: Auxiliary particle filters," J. Amer. Statist. Assoc., vol. 94, no. 446, pp. 590–599, 1999.

[44] C. Musso, N. Oudjane, and F. LeGland, "Improving regularised particle filters," in Sequential Monte Carlo Methods in Practice, A. Doucet, J. F. G. de Freitas, and N. J. Gordon, Eds. New York: Springer-Verlag, 2001.

[45] Gordon, Salmond & Smith, Novel approach to nonlinear non-Gaussian Bayesian state estimation, IEE, 1993

[46] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in In Proceedings of the 11th International Conference on Advanced Robotics (ICAR), 2003, pp. 317–323

[47] Z. Dodds and B. Tribelhorn, "Erdos: Cost effective peripheral robotics for AI education," in Proceedings, AAAI, 2006, pp. 1966–1967.

[48] T. Kurt, Hacking Roomba: ExtremeTech. Wiley, 2006.

[49] Ben Tribelhorn, Zachary Dodds. Evaluating the Roomba: A low-cost, ubiquitous platform for robotics research and education. In Proceedings of ICRA'2007. pp.1393~1399

[50] http://hackingroomba.com/projects/

[51] N. M. Kwok and G. Dissanayake, "Bearing-only. SLAM in indoor environments using a modified particle filter

[52] http://www.analog.com/library/analogDialogue/Anniversary/7.html

APPENDIX

Arduino Uno Microcontroller

This is the implementation code for Arduino Uno Microcontroller component to write the sensed data to a COM Port

```
#include <math.h> // (no semicolon)
//// VARS
byte CLK_pin = 8;
byte EN_pin = 9;
byte DIO_pin = 10;

intX_Data = 0;
intY_Data = 0;
int angle;

constint pingPin1 = 2;
constint pingPin2 = 4;
constint pingPin3 = 7;


//// FUNCTIONS

void ShiftOut(int Value, intBitsCount) {
  for(int i = BitsCount; i >= 0; i--) {
digitalWrite(CLK_pin, LOW);
    if ((Value & 1 << i) == ( 1 << i)) {
digitalWrite(DIO_pin, HIGH);
      //Serial.print("1");
    }
    else {
digitalWrite(DIO_pin, LOW);
      //Serial.print("0");
    }
digitalWrite(CLK_pin, HIGH);
delayMicroseconds(1);
  }
//Serial.print(" ");
}

intShiftIn(intBitsCount) {
intShiftIn_result;
ShiftIn_result = 0;
pinMode(DIO_pin, INPUT);
    for(int i = BitsCount; i >= 0; i--) {
digitalWrite(CLK_pin, HIGH);
delayMicroseconds(1);
      if (digitalRead(DIO_pin) == HIGH) {
ShiftIn_result = (ShiftIn_result<< 1) + 1;
        //Serial.print("x");
      }
      else {
ShiftIn_result = (ShiftIn_result<< 1) + 0;
        //Serial.print("_");
      }
digitalWrite(CLK_pin, LOW);
```

```
delayMicroseconds(1);
    }

  if ((ShiftIn_result& 1 << 11) == 1 << 11) {
ShiftIn_result = (B11111000 << 8) | ShiftIn_result;
  }


  return ShiftIn_result;
}

void HM55B_Reset() {
pinMode(DIO_pin, OUTPUT);
digitalWrite(EN_pin, LOW);
ShiftOut(B0000, 3);
digitalWrite(EN_pin, HIGH);
}

void HM55B_StartMeasurementCommand() {
pinMode(DIO_pin, OUTPUT);
digitalWrite(EN_pin, LOW);
ShiftOut(B1000, 3);
digitalWrite(EN_pin, HIGH);
}

int HM55B_ReadCommand() {
int result = 0;
pinMode(DIO_pin, OUTPUT);
digitalWrite(EN_pin, LOW);
ShiftOut(B1100, 3);
  result = ShiftIn(3);
  return result;
}


long microsecondsToCentimeters(long microseconds){
  // The speed of sound is 340 m/s or 29 microseconds per centimeter.
  // The ping travels out and back, so to find the distance of the
  // object we take half of the distance travelled.
  return microseconds / 29 / 2;
}


long calcDuration(intpingPin){
  // The PING))) is triggered by a HIGH pulse of 2 or more microseconds.
  // Give a short LOW pulse beforehand to ensure a clean HIGH pulse:
pinMode(pingPin, OUTPUT);
digitalWrite(pingPin, LOW);
delayMicroseconds(2);
digitalWrite(pingPin, HIGH);
delayMicroseconds(5);
digitalWrite(pingPin, LOW);

  // The same pin is used to read the signal from the PING))): a HIGH
  // pulse whose duration is the time (in microseconds) from the sending
  // of the ping to the reception of its echo off of an object.
pinMode(pingPin, INPUT);
  return pulseIn(pingPin, HIGH);
}
```

```
void setup() {
Serial.begin(115200);
pinMode(EN_pin, OUTPUT);
pinMode(CLK_pin, OUTPUT);
pinMode(DIO_pin, INPUT);

  HM55B_Reset();
}


void loop(){
  long duration1, duration2, duration3, cm1, cm2, cm3;

  HM55B_StartMeasurementCommand(); // necessary!!
  delay(40); // the data is 40ms later ready
  HM55B_ReadCommand();

  //Serial.print(); // read data and print Status
  //Serial.print(" ");

  X_Data = ShiftIn(11); // Field strength in X
  Y_Data = ShiftIn(11); // and Y direction
  digitalWrite(EN_pin, HIGH); // ok deselect chip

  duration1 = calcDuration(pingPin1);
  duration2 = calcDuration(pingPin2);
  duration3 = calcDuration(pingPin3);

  // convert the time into a distance
  //inches = microsecondsToInches(duration);
  cm1 = microsecondsToCentimeters(duration1);
  cm2 = microsecondsToCentimeters(duration2);
  cm3 = microsecondsToCentimeters(duration3);


  angle = 180 * (atan2(-1 * Y_Data , X_Data) / M_PI); // angle is atan( -y/x) !!!


Serial.print(angle); // print angle
Serial.print(" ");
Serial.print(cm2);
Serial.print(" ");
Serial.print(cm1);
Serial.print(" ");
Serial.print(cm3);
Serial.println();
}
```

Serial Port Reader

This component reads the data from the COM port and reports it to the MATLAB component

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO.Ports;

namespace Serial
{
    class Program
    {
        public static void Main(string[] args)
        {
            //System.Console.WriteLine("Entered");
            Int32 baudRate, iterations;
            string comPort = args[0];

            double angle, lDist, rDist, cDist;
int v0, v1, v2, v3;
int i = 1, numer=0, denom =0, sv1=0, sv2=0, sv3=0;

            Dictionary<int, int>dictAng;

            //System.Console.WriteLine("Initialized");
            while (true) {
                try{
baudRate = Convert.ToInt32(args[1]);
                    break;
                }
                catch (FormatException) {
                    continue;
                }
            }
            while (true){
                try{
                    iterations = Convert.ToInt32(args[2]);
                    break;
                }
                catch (FormatException){
                    continue;
                }
            }

dictAng = new Dictionary<int, int>();

            using (SerialPort port = new SerialPort(comPort, baudRate)) {
port.Open();
string[] strArray = new string[4];
                while (i < iterations){
                    try{
strArray = port.ReadLine().Split(new char[] { ' ' });
                    }
                    catch (TimeoutException) {
continue;
                    }
```

```
                    try{
                        v0 = Convert.ToInt32(strArray[0]);
                        v1 = Convert.ToInt32(strArray[1]);
                        v2 = Convert.ToInt32(strArray[2]);
                        v3 = Convert.ToInt32(strArray[3]);
                    }
                    catch (FormatException){
continue;
                    }
                    catch (IndexOutOfRangeException) {
continue;
                    }
if(dictAng.ContainsKey(v0))
dictAng[v0]++;
                    else
dictAng[v0]=1;


                    sv1 += v1;
                    sv2 += v2;
                    sv3 += v3;

                    ++i;
                }
            }
foreach(KeyValuePair<int, int>kv in dictAng){
numer += kv.Value * kv.Key;
denom += kv.Value;
            }
            angle = numer / denom;
            if(angle<0.0)
                angle = 180.0 + (180.0 - Math.Abs(numer/denom));

lDist = sv1 / iterations;
rDist = sv2 / iterations;
cDist = sv3 / iterations;

System.Console.Write("{0} {1} {2} {3}", angle, lDist, rDist, cDist);
        }
    }
}
```