Indoor Localisation and Mapping using Laser Range Finder

by

Ahmed Hesham Lotfy

14704

Dissertation submitted in partial fulfilment of

the requirement for the

Bachelor of Engineering (Hons)

(Electrical and Electronics)

JANUARY 2015

Universiti Teknologi PETRONAS

Bandar Seri Iskandar

32610 Tronoh

Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

Indoor Localisation and Mapping using Laser Range Finder

by

Ahmed Hesham Lotfy

14704

A project dissertation submitted to the

Electrical and Electronics Engineering Programme

Universiti Teknologi PETRONAS

In partial fulfilment of the requirement for the

BACHELOR OF ENGINEERING (Hons)

(ELECTRICAL AND ELECTRONICS)

Approved by,

____

Dr. Abu Bakar Sayuti Hj Mohd Saman

UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

JANUARY 2015

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own expect as specified in the references and acknowledges, and that the original work contained herein have not been undertaken or done unspecified sources or persons.

……………………………………..

Name: Ahmed Hesham Lotfy

Date: 18 April 2015

Abstract

Simultaneous localization and mapping (SLAM) is one of the uprising and evolving branches in the field of robotics. SLAM gives a robot the capability of drawing a consistent map of the surrounding area while simultaneously localizing itself within the map without the need of having prior data about the surrounding like a pre-encoded map or localization aids like the Global Positioning System (GPS). Solving the SLAM problem have been recognised as the holy grail in the eyes of the mobile robotics community as it will provide the robots with the capability to be independently autonomous. Several algorithms are used in utilizing the SLAM problem and the most highlighted algorithms is the Extended Kalman filter (EKF).

This project is concerned with applying the EKF on a mobile robot and test its efficiency in solving the SLAM problem. The methodology is comprised of two steps which are software implementation where a python code is written to emulate the mathematical model of the EKF and hardware implementation where a test bench including a robot platform and arena are used to test the efficiency of the EKF python code of the in real life SLAM applications.

Acknowledgments

I would like to express my gratitude to all who supported me in this project and special thanks to my supervisor Dr Abu Bakar Sayuti Hj Mohd Saman for giving me the chance to work on this project and guiding me all through the process till the end.

Special thanks to Claus Brenner for his tutorials that have guided me to accomplish this project and finally million thanks to my beloved family and friends for their prayers and help during the whole period of this project.

Contents

List of Figures ... 1
CHAPTER 1 ... 2
1.1 Background ... 2
1.2 Problem Statement ... 3
1.3 Objectives ... 4
1.4 Scope of study ... 4
CHAPTER 2 ... 5
2.1 Definition of the SLAM problem ... 5
2.2 Extended Kalman Filter SLAM ... 11
2.2.1 Motion Model ... 11
2.2.2 Sensor Model ... 14
2.3 EKF SLAM Steps ... 16
2.3.1 *Predi*ction Step ... 16
2.3.2 Correction Step ... 17
CHAPTER 3 ... 19
3.1 Overview ... 19
3.1.1 Software Implementation ... 19
3.1.2 Hardware Implementation ... 23
3.2 Hardware Integration ... 28
CHAPTER 4 ... 29
4.1 Software implementation results ... 30
4.2 Hardware Implementation Results ... 31
CHAPTER 5 ... 33
References ... 34

List of Figures

Figure 1 : Robot observers the environment in initial position [4] 5
Figure 2 : The robot estimates its position using odometry after movement [4] 6
Figure 3 : The robot measures the location of landmarks [4] 6
Figure 4 : The robot identify its true position based on landmark locations [4] 7
Figure 5 : Relation between real robot position, odometry estimate, and updated position [4] 7
Figure 6: SLAM problem example [3].. 8
Figure 7 : Velocity model representation 12
Figure 8 : Odometry Motion Model 13
Figure 9: Spiral Network 15
Figure 10 : Python Simulator GUI 22
Figure 11: Arena presentation 23
Figure 12 : Real Arena 24
Figure 13 : Snapshot from the overhead camera 24
Figure 14 : Snapshot from the overhead camera 24
Figure 15: URG-04LX-UG01 Laser range finder... 26
Figure 16 : Raspberry PI Model B specifications... 27
Figure 17 : Hercules Dual 15A 6-20V Motor Controller 27
Figure 18 : Hardware Integration.. 28
Figure 19: Python Logfile_viewer simulator output 29
Figure 20 : Software implementation Before Correction 30
Figure 21 : Software implementation After Correction.. 31
Figure 22 : Hardware implementation Before Correction... 32
Figure 23 : Hardware implementation After Correction 32

CHAPTER 1
INTRODUCTION

1.1 Background

The idea of simultaneous localization and mapping was first introduced in 1986 in the IEEE Robotics and Automation Conference which was held in San Francisco, California by some researchers including Peter Cheeseman, Hugh Durrant-Whyte and Jim Crowely. Various efforts have been done to solve the SLAM problem but nothing converged until 1995 where the convergence result of the SLAM problem was first introduced in the International Symposium on Robotics Research [3].

For a robot to be autonomous and be able to roam an environment freely while avoiding the obstacles it needs two important key aspects which are mapping and localization [1].

The mapping is a problem of utilizing the sensor gathered data into useful information about the surroundings so the robot can basically understand how the environment look like. Using the robot odometery and the raw data gathered by the sensor the robot can be capable of building a reasonable map as it moves through the environment. After acquiring the map the robot needs to localize its position in this map with the least error in the estimation process and this is considered the localization process [1].

Solving the simultaneous localization and mapping problem provided a great opportunity for autonomous robots where having an integrated system inside the mobile robot that can roam in an environment with the capability to localize its own location and map the surrounding using on board sensors with no pre information about its location or the map of the environment is considered a breakthrough in the field of autonomous robotics [1].

Hence, the aim of this project is to apply the Extended Kalman filter on a mobile robot that should be able to simultaneously localize itself and map the surrounding environment providing a solution to the SLAM problem.

1.2 Problem Statement

Indoor simultaneous localization and mapping is quite a tough problem to fix as the robot performing the SLAM process don't have any pre-loaded map of the environment, basically the robot is totally ignorant about the environment which make the SLAM a complex problem to fix. Therefore, in order to fix the SLAM problem we need to address each of localization and mapping we need to address two issues which are:

- To have accurate localization then an accurate map is needed. (Localization)
- To have an accurate map then an accurate localization is needed. (Mapping)

Various probabilistic mathematical models have been introduced as solutions for the SLAM problem but only few hardware applications were implemented to prove the validity of these solutions and their limits. Therefore, this project is mainly concerned with applying the Extended Kalman filter , which is one of the SLAM solutions, on a mobile robot and prove its efficiency in real life applications.

1.3 Objectives

The main objectives of this project are:

- Apply Simultaneous localization and mapping on a robot equipped with Laser range finder and wheel encoders.
- Prove the validity of the applied algorithm based on the robot performance.

1.4 Scope of study

This study is concerned with three main points which are:

A. Computational Model

Converting the Extended Kalman filter mathematical model to a python code that can be used in robotic applications.

B. Navigation and Data gathering

A robot platform equipped with laser range finder and wheel encoders that is navigate manually through an arena collecting laser scanner and wheel encoders' readings at certain time intervals.

C. Validation and Data Representation

Using a software simulation program for results validation and comparison

CHAPTER 2
LITERATURE REVIEW

2.1 Definition of the SLAM problem

SLAM question whether a robot can navigate through an environment which it has no prior information about and build a map of its surroundings while simultaneously localizing itself within this map. There have been various implementations of the SLAM in different domains including indoor, outdoor, airborne and underwater robotic systems on both theoretical and conceptual levels. [2] [3]

The SLAM is divided into two major parts, first the robot trajectory through the environment and second the environment landmarks which are estimated online without prior knowledge about their locations. To have an overview of the concept of SLAM consider a robot moving through an environment capturing observations using a sensor on its top.

As the robot starts in the environment it starts observing the landmarks as shown in figure 1 where the robot is represented as the triangle and the landmarks are represented as the stars. The robot will define its initial position to be the starting position and measure the distance to the landmarks using the laser range finder sensor.

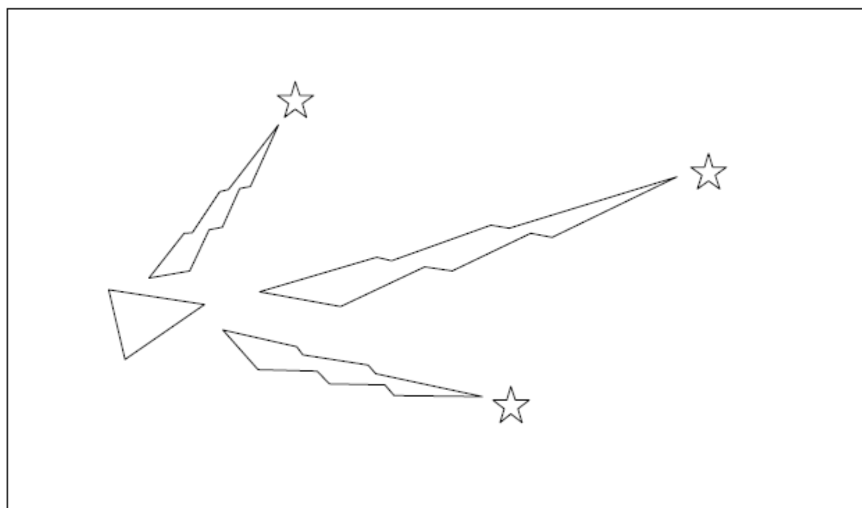

Figure 1 : Robot observers the environment in initial position [4]

Afterwards, the robots starts moving then using the robot odometry it identifies its position in the real world which turns out to be wrongly acquired due to the odometry error as shown in figure 2.

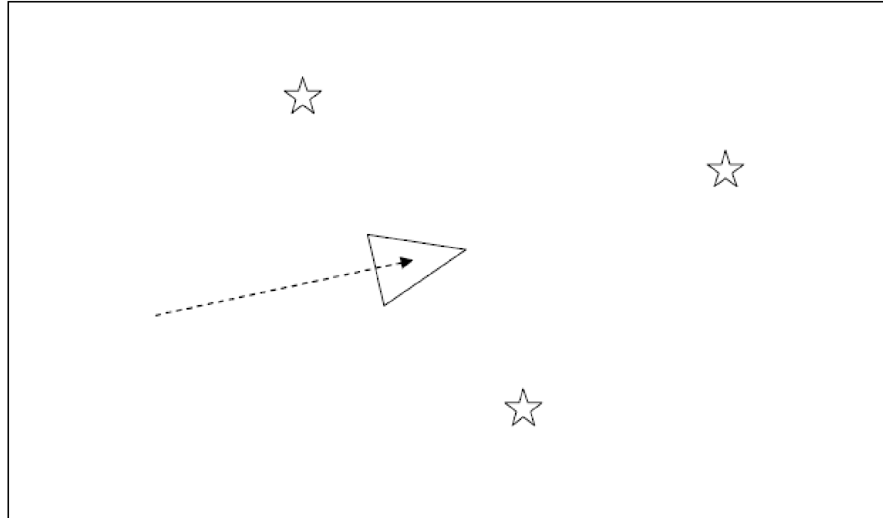

Figure 2 : The robot estimates its position using odometry after movement [4]

Then, as shown in figure 4, the robot measure again the distance to the landmarks using the laser range finder and compare the measurement to the estimated location where this landmarks should be after moving using the odometry data. Through this process it can be found how far is the robot thinks it's located based on odometry from its real location in the environment.

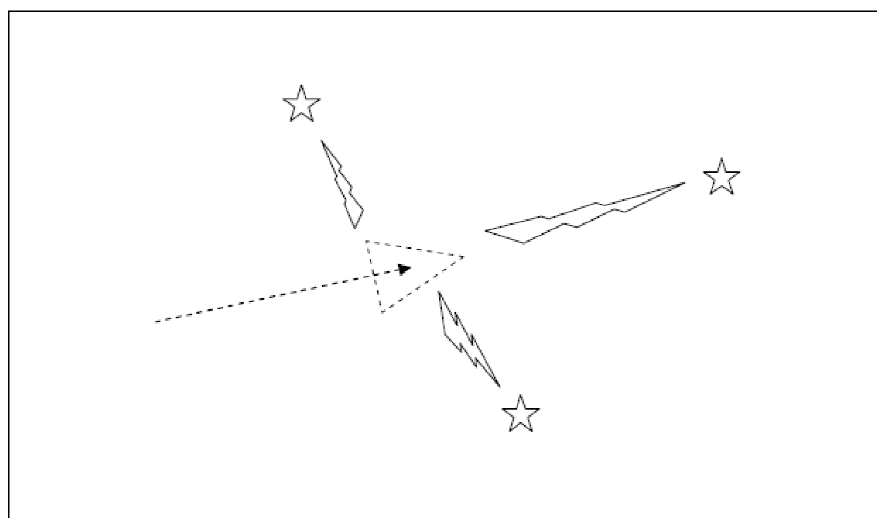

Figure 3 : The robot measures the location of landmarks [4]

Having acquired the odometry calculated position and the laser range finder's readings, the robot will believe more in the laser range finder data as its error is minimal if compared to the odometry error and based on the landmarks reading it will re localize its position estimate. As shown in figure 4, the dashed triangle shows where the robot thought it was and the triangle shows the update of the robot location depending on the landmarks readings.

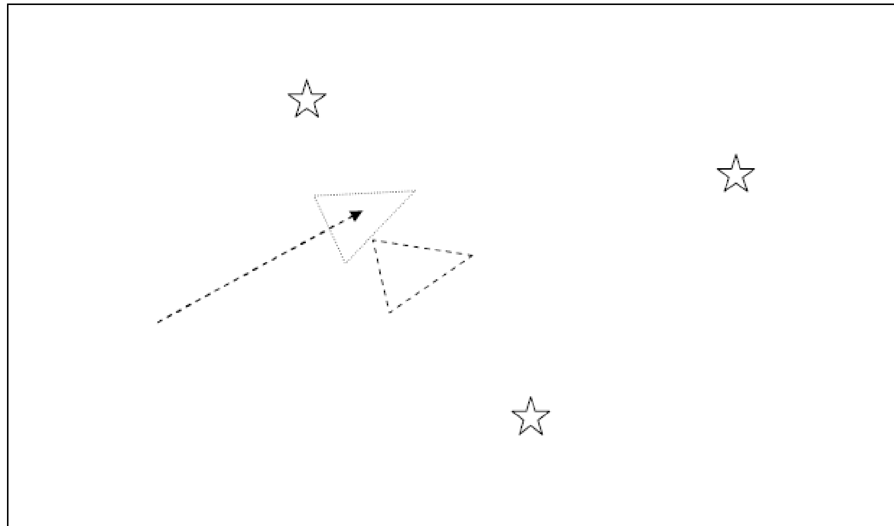

Figure 4 : The robot identify its true position based on landmark locations [4]

Finally, figure 6 shows the difference between depending on the odometry alone or using the landmarks to get a better estimate of the robot position compared to where the robot actually exist in the real world. The dashed triangle shows the odometry estimate of the robot position, the dotted triangle shows the estimate of the robot position after taking in account the landmarks feedback and the triangle shows the real position of the robot in the environment.

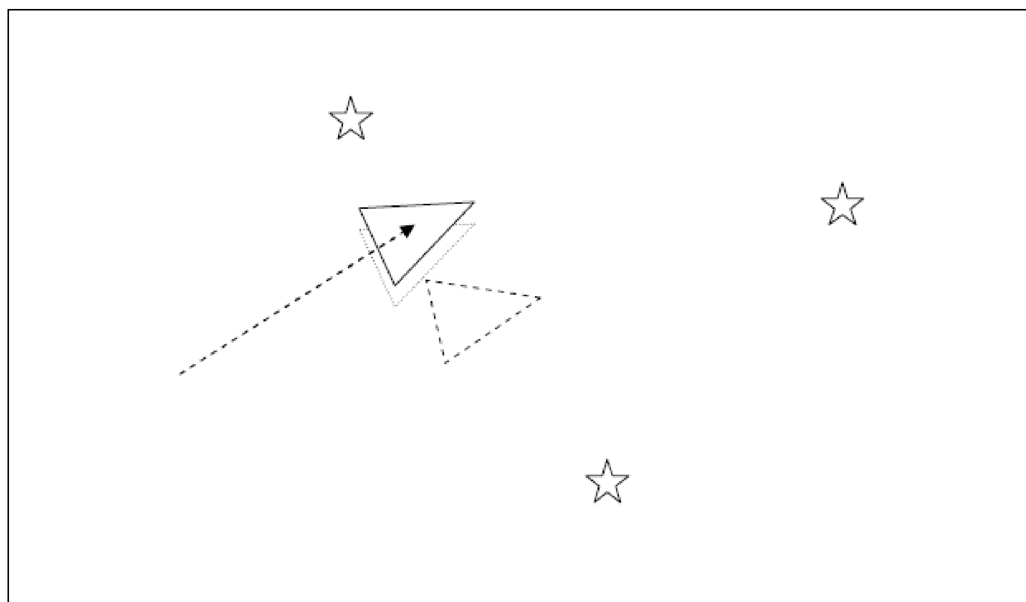

Figure 5 : Relation between real robot position, odometry estimate, and updated position [4]

The full process is as shown in Figure 6, the robot goes through the environment equipped with wheel encoders and laser sensor capturing data from both sensors at certain time intervals. At each time interval k, various variables are calculated which are:

- x_k: the state vector including robot's x , y coordinates and heading.
- u_k: the control vector, includes the motion commands given at time k - 1 to drive the robot to a new state x_k at time k
- m_i: the landmark location coordinates of the i_{th} landmark.
- z_{ik}: the observation reading of the robot of the i_{th} landmark's location at time k.

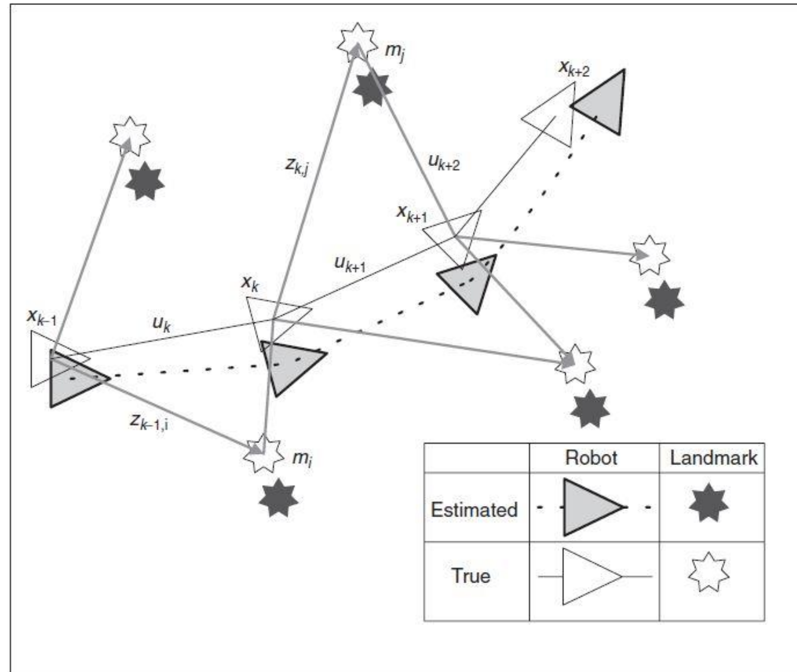

Figure 6: SLAM problem example [3]

The probabilistic nature of the SLAM problem requires that the probability distribution

$$P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) \quad (1)$$

should be solved at all-time intervals k. This probability distribution emphasize the dependency of obtaining the robot current pose x_k and the environment map m on the posterior sensor readings of the environment for all time intervals $Z_{0:k}$, the posterior motion commands for all time intervals $U_{0:k}$ and the robot previous pose [3].

In order to solve the probability distribution (1), a recursive solution is derived which is divided into two steps which are the prediction (2) and correction (3) steps.

$$P(x_k, m|Z_{0:k-1}, U_{0:k}, x_0) = \int P(x_k |x_{k-1}, u_k) \times P(x_{k-1}, m|Z_{0:k-1}, U_{0:k-1}, x_0)\, dx_{k-1} \quad ..(2)$$

$$P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k |x_k, u_k)\, P(x_k, m \mid Z_{0:k-1}, U_{0:k}, x_0)}{P(z_k | Z_{0:k-1}, U_{0:k})} \quad ...(3)$$

Equation (2) is the prediction step which predicts the current state of the robot in the environment $P(x_k, m|Z_{0:k-1}, U_{0:k}, x_0)$ based on the integrating the multiplication of the robot motion model $P(x_k |x_{k-1}, u_k)$, which provides the probable location x_k of the robot based on its previous location x_{k-1} and the motion commands u_k given to the robot to reach x_k state, and the previous belief of the robot location $P(x_{k-1}, m|Z_{0:k-1}, U_{0:k-1}, x_0)$ at time $k-1$.

Equation (3) is the correction step which produce the corrected state of the robot $P(x_k, m \mid Z_{0:k}, U_{0:k}, x_0)$ based on the predicted state $(x_k, m \mid Z_{0:k-1}, U_{0:k}, x_0)$, the sensor observation of the environment $P(z_k |x_k, u_k)$ which provides the robot observation of the environment z_k given the robot current state x_k and the motion commands u_k given to the robot to reach x_k state.

A simpler representation for the prediction and correction models represented in equations (3) and (4) are derived in equations (5) and (6) respectively as follows.

$$\overline{bel(x_t)} = \int P(x_k \,|x_{k-1}, u_k) \times bel(x_{t-1}) \;\; dx_{t-1} \text{ ...... (5)}$$

$$bel(x_t) = \eta \times P(z_k \,|x_k, u_k) \times \overline{bel(x_t)} \text{ .... (6)}$$

where:

- $\overline{bel(x_t)} = P(x_k, m|Z_{0:k-1}, U_{0:k}, x_0)$ *"predicted belief of the robot's position"*
- $bel(x_t) = P(x_k, m \,|\, Z_{0:k}, U_{0:k}, x_0)$ *"corrected belief of the robot's position"*
- $\eta = \frac{1}{P(z_k|\, Z_{0:k-1}, U_{0:k})}$

At this point, a defenition of the SLAM problem probability distribution was introduced and then a recurcisve model for the derieved in (2) "Predection" and (3)"Correction". The next section will be concerned by providing a solution for the SLAM problem using Extended Kalman filter.

2.2 Extended Kalman Filter SLAM

The Extended Kalman Filter is an interpretation from the Kalman Filter, but since the SLAM problem is nonlinear in nature therefore changed form of the normal Kalman Filter, which only uses linear models, had to be derived which is the Extended Kalman Filter [1].

As shown earlier, Equations (3) to (6) provide the outline for the SLAM problem highlighted in equation (1), therefore we will first derive a representation for both the motion model and the observation model then we will implement the EKF to solve the SLAM problem.

2.2.1 Motion Model

In this section, the motion model will be further discussed with the aim to generate the model's structure. The motion model will be represented in two forms which are:

- The Velocity Model
- The Odometry Model

each of the models output is the provide a solution for the motion model in shown below.

$$P(x_k \mid x_{k-1}, u_k) \text{ …… (7)}$$

A. The Velocity Model

The velocity model basis is that the motion model can be calculated given the rotational velocity *w* and transitional velocity *v* of the motion [2]. Based on this, the control commands are represented as follows:

$$u_t = \begin{pmatrix} v_t \\ w_t \end{pmatrix} \quad \ldots\ldots (8)$$

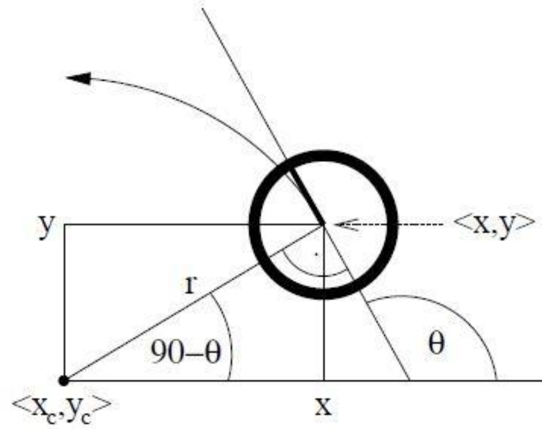

Figure 7 : Velocity model representation

Given the previous state of the robot x_{k-1} and the control command u_t , the current robot's state x_k can be calculated as follows [2]:

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v}{w}\sin\theta + \frac{v}{w}\sin(\theta + w\Delta k) \\ -\frac{v}{w}\cos\theta + \frac{v}{w}\cos(\theta + w\Delta k) \\ w\Delta k \end{pmatrix} \quad \ldots\ldots (9)$$

where:

- $\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix}$ is the current robot state
- $\begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$ is the previous robot state
- v is the transitional velocity
- w is the angular velocity
- $\Delta k = k - k_{-1}$ the delta of the time intervals k and k_{-1}

B. The Odometry Model

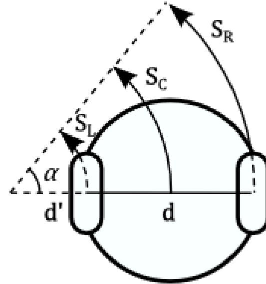

Figure 8 : Odometry Motion Model

As discussed earlier, the velocity model depend on the robot velocity to calculate the current state. Alternatively the odometry model obtain the robot pose by integrating the wheel encoder information as follows [2]:

If $S_R = S_L$

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} S_L \cos(\theta) \\ S_L \sin(\theta) \\ 0 \end{pmatrix} \ldots\ldots. (10)$$

If $S_R \neq S_L$

$$\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} \left(d' + \frac{d}{2}\right)[\sin(\theta + \alpha) - \sin\theta] \\ \left(d' + \frac{d}{2}\right)[-\cos(\theta + \alpha) - \cos\theta] \\ \alpha \end{pmatrix} \ldots\ldots. (11)$$

where:

- $\begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix}$ is the current robot state
- $\begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$ is the previous robot state
- $\alpha = \frac{S_R - S_L}{d}$
- $d' = \frac{S_L}{\alpha}$

2.2.2 Sensor Model

After solving the motion model problem we are left with the sensor model. The sensor model describes how the sensor's measurements formulated in the real world.

$$P(z_k \mid x_k, u_k) \;\ldots\ldots (12)$$

As the robot moves through the environment the sensor will capture various numerical measurements like range finders generating scans of ranges or cameras generating arrays of colours. If we consider the range finders, given the measured scans of ranges we can generate a model for perceiving landmarks as follows [2]:

$$\begin{pmatrix} r_t' \\ \theta_t' \end{pmatrix} = \begin{pmatrix} \sqrt{(m_{jx} - x)^2 + (m_{jy} - y)^2} \\ \text{atan}(\frac{m_{jy}-y}{m_{jx}-x}) - \theta \end{pmatrix} \ldots\ldots (13)$$

where:

- $\begin{pmatrix} r_t' \\ \theta_t' \end{pmatrix}$ are the range and bearing of the extracted landmark
- $\begin{pmatrix} m_{jx} \\ m_{jy} \end{pmatrix}$ are the x and y coordinates of the extracted landmark
- $\begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$ is the robot's state

More insight into the sensor model in (12), In Figure 6, it's clear that most of the error in the landmark reading is due to the uncertainty in the robot position in the environment which leads to having a wrong sensor reading. However, the relation between any two landmarks can be known with high accuracy even though the exact locations of both landmark is not accurate and this is known as joint probability where the joint probability density between landmarks mi and mj P (m_i, m_j) is high even though the individual marginal density of each landmarks P (m_i) or P (m_j) is low [3].

This insight shows that the correlation between the landmarks in monotonic as the number of observations done by the robot increase where the relative location of

landmarks always is in continuous improvement but not diverging independent of the robot position in the environment.

In order to have a better understanding of this concept lets return back to Figure 6, consider the robot starts at location x_k and observe the two landmarks mi and m_j, Afterwards, the robot moves to location x_{k+1} and it re-observe landmark m_j which will help both the robot's location and landmark's m_j location based on the robot's previous location x_k. In turn, the new update propagates back to update the landmark m_i even though it was not observed by the robot in the new location x_{k+1}. This occurs because both of the landmarks are highly correlated to each other as P (m_i, m_j) density is high so any update in m_j will propagate to update m_i as well [3].

To visualize the process illustrated earlier, conisder a robot moving in an enviroment with various landmarks. As the robot observes landmarks, a spring is drawn between them and the thicknes of the spring shows how much are the landmarks correlated. Thus, as the robot goes through the enviroment and observe the landmarks repeadetly a spiral network is formed as shown in Figure 9 between the landmarks and also the robot where the effect updating a node in the network will ripple to update all the other nodes depending on the connections strength or in other words how much are they correlated [3].

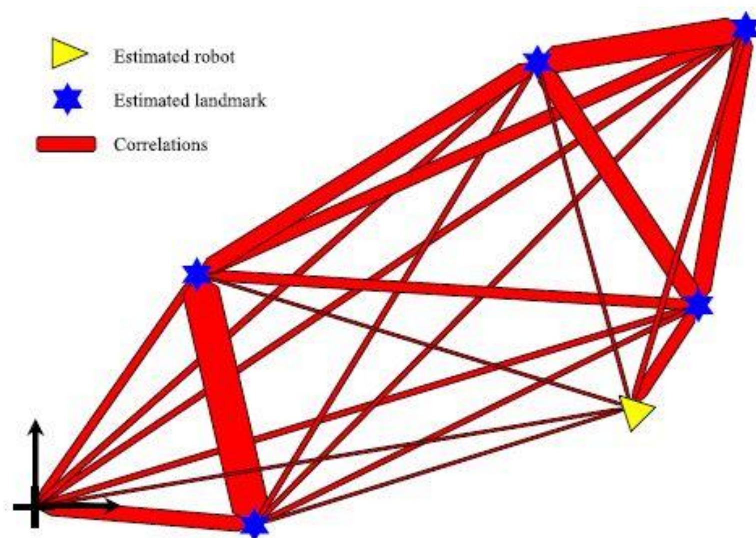

Figure 9: Spiral Network

Having derieved a representation for the motion model (7) and the observation model (12) we are now able to implement the Extended Kalman Filter.

2.3 EKF SLAM Steps

The EKF SLAM method mainly consists of two steps which are the Prediction Step and the Correction Step. The Algorithim that the EKF use to derieve both the Prediction and Correction steps is :

$$\mu_k' = g(u_k, \mu_{k-1}) \;\ldots.\; (14)$$

$$\varepsilon_k' = G_k \, \varepsilon_{k-1} G_k^{\;T} + R_k \;\ldots.\; (15)$$

$$K_k = \varepsilon_k' H_k^{\;T} \left(H_k \, \varepsilon_k' \, H_k^{\;T} + Q_k\right)^{-1} \ldots.\; (16)$$

$$\mu_k = \mu_k' + K_k(z_{\boldsymbol{k}} - h(\mu_k')) \;\ldots.\; (17)$$

$$\varepsilon_k = (I - K_k \, H_k) \, \varepsilon_k' \;\ldots.\; (18)$$

2.3.1 ***Predi*ction Step**

In the predection step, both the estimated Mean and the estimated Covariance are calculated. The estimated Mean is calculated as in equation (14) which is a function of the robot motion model discussed earlier either a velocity model or an odometry model.The calculated estimated Mean result is the estimated robot state and the estimated location of the estimated landmarks.

$$\mu_k' = \begin{pmatrix} x' \\ y' \\ \theta' \\ {m_{xi}}' \\ {m_{yi}}' \\ {\theta_i}' \\ {m_{xj}}' \\ \ldots \end{pmatrix} \ldots.\; (19)$$

On the other hand the estimated Covariance shows the relation between the landmarks and robot state and provide a representation as the spring network discussed earlier. The Estimated covariance in (15) is calculated by multiplting previous covariance by the Jacobian of the motion model G_k and adding non linear Gaussian noise of the motion model R_k. The final shape of the estimated covariance is shown in (20).

$$\varepsilon_k{}' = \begin{pmatrix} \alpha_{xx} & \alpha_{xy} & \alpha_{x\theta} & \alpha_{x\,m_{ix}} & \alpha_{x\,m_{iy}} \cdots & \alpha_{x\,m_{nx}} & \alpha_{x\,m_{ny}} \\ \alpha_{yx} & \alpha_{yy} & \alpha_{y\theta} & \alpha_{y\,m_{ix}} & \alpha_{y\,m_{iy}} \cdots & \alpha_{y\,m_{nx}} & \alpha_{y\,m_{ny}} \\ \alpha_{\theta x} & \alpha_{\theta y} & \alpha_{\theta\theta} & \alpha_{\theta\,m_{ix}} & \alpha_{\theta\,m_{iy}} \cdots & \alpha_{\theta\,m_{nx}} & \alpha_{\theta\,m_{ny}} \\ \alpha_{m_{ix}\,x} & \alpha_{m_{ix}\,y} & \alpha_{\theta} & \alpha_{m_{ix}\,m_{ix}} & \alpha_{m_{ix}\,m_{iy}} \cdots & \alpha_{m_{ix}\,m_{nx}} & \alpha_{m_{ix}\,m_{ny}} \\ \alpha_{m_{iy}\,x} & \alpha_{m_{iy}\,y} & \alpha_{\theta} & \alpha_{m_{iy}\,m_{ix}} & \alpha_{m_{iy}\,m_{iy}} \cdots & \alpha_{m_{iy}\,m_{nx}} & \alpha_{m_{iy}\,m_{ny}} \\ \vdots & \vdots & \vdots & & & & \\ \vdots & \vdots & \vdots & \alpha_{m_{nx}\,m_{ix}} & \alpha_{m_{nx}\,m_{iy}} \cdots & \alpha_{m_{nx}\,m_{nx}} & \alpha_{m_{nx}\,m_{ny}} \end{pmatrix} \quad (20)$$

2.3.2 Correction Step

The Correction step consist of three stages, first the calculation of the Kalman gain which is followed by the correction of the estimated Mean and finally the correction of the estimated Covariance.

In the first step as shown in equation (16) the Kalman gain is calculated based on the estimated covariance and the Jacobian of the function $h(\mu_k{}')$ which is the sensor model used for landmark extraction discussed earlier. The Kalman gain is the factor of correction as it depends on the sensor observation and the covariance to calculate the factor need to correct each estimated element with the addition of non linear Gaussian noise of the sensor model.

Afterwards, equation (17) use the Kalman gain to correct the predicted Mean $\mu_k{}'$. The amount of correction is the multiplication of the Kalman gain with

$(z_{\boldsymbol{k}} - h(\mu_k{}')$ where $z_{\boldsymbol{k}}$ is the predicted observations based on the predicted robot location calculated in the prediction step, in other words its considered what the robot is expected to see as it moved from K to K+1 .

Finally, equation (18) calculate the corrected Covariance using the Kalman gain and the Jacobian of the sensor model. These two steps will be done sequentially at certain time intervals as the robot moves through the enviroment and the result will be an acceptable estimate of the robot location at all points and a near accurate map of the enviroment, with this the SLAM problem is solved by implementing the Extended Kalman Filter.

CHAPTER 3
METHODOLOGY

3.1 Overview

The main goal of the project is to implement simultaneous localization and mapping in an indoor environment. This goal is achieved through two stages which are:

- Software Implementation
- Hardware Implementation

In the first stage which is software implemntation, a python code is written to implement the mathematical model of the Extended Klaman filter given inputs from wheel encoders and Laser range finder at various time intervals. Moving to the second stage, a manually driven robot is built and mounted with a laser range finder.The robot is manually driven through an indoor enviroment while capturing readings from the wheel encoders and the laser sensor.

The preceding sections will discuss more details about both stages and show how each stage was implementated.

3.1.1 Software Implementation

In this stage a python code is developed to implement the mathematical model of the Extended Kalman filter.The code recieves time spaced wheel encoders values and laser range finder readings then outputs a corrected robot path and landmarks locations .The results are then represented by a simulator devloped by python coding to show the robot corrected path and compare the result with the input path.

The input Laser range finder and encoder readings are provided by an online database which have the data gathered by a robot in an indoor enviroment at different time intervals.

The main code is *EKF_SLAM.py* which is included in Appendix A which contains the EKF both predection and corrction steps. This code uses a library included in Appendix B *EKF_Slam_Library.py* which contains various functions used through the code processing.

The *EKF_SLAM.py* consists mainly of the class **ExtendedKalmanFilterSLAM**

```
class ExtendedKalmanFilterSLAM
```

The class **ExtendedKalmanFilterSLAM** is provided with :

- Initial state of the robot.
- Initial covariance.
- Robot width
- Scanner displacement from the robot center.
- Control motion factor which represent the amount of wheel's deviation in straight line motion.
- Control turn factor which represent the amount of wheel's deviation in rotational motion.
- Measurement distance standard deviation which represent the amount of sensor's deviation in distance reading.
- Measurement angle standard deviation which represent the amount of sensor's deviation in angluar reading.

```
kf = ExtendedKalmanFilterSLAM(
                                initial_state, initial_covariance,
                                robot_width, scanner_displacement,
                                control_motion_factor,
                                control_turn_factor,
                                measurement_distance_stddev,
                                measurement_angle_stddev
                                )
```

The **ExtendedKalmanFilterSLAM** class consists of multiple functions that are implemented in sequence to provide the whole implemntation of the Extended Kalman filter.The first function and second functions are

- `def g(state, control, w):`
- `def h(state, landmark, scanner_displacement):`

The function `g` takes the encoder collected by the robot as it moves through the enviroment and applies the odometry motion model to generate x , y and heading for the robot at each time interval hence generate the full path of the robot.

As for the `h` function it takes the robot state, the observed landmarks and the scanner displacment from the robot center and provide the sensor model for the laser range finder as shown in equation (13).

The third , foueth and fifth functions are

- `def dg_dstate(state, control, w):`
- `def dg_dcontrol(state, control, w):`
- `def dh_dstate(state, landmark, scanner_displacement):`

The function `dg_dstate` takes the robot initial state and encoder values then use it to calculate the Jacobian matrix of the state which is G_k in equation (15) on the other hand the function `dg_dcontrol` takes the same input and provide the Jacobian matrix used to calculate the wheel encoders' noise R_k.Finally , the function `dh_dstate` is used to calculate the Jacobian matrix of the Sensor model H_k in equation (16) given the current robot state ,landmarks observed locations and the laser scanner's displacement from the centre of the robot.

Finally the two main functions of the **`ExtendedKalmanFilterSLAM`** class are the prediction and correction functions.

- `def predict(self, control):`
- `def correct(self, measurement, landmark_index):`

The `predict` function uses the robot state and the observed landmarks locations in order to calculate the predicted Mean $\mu_k{}'$and the predicted Covariance $\varepsilon_k{}'$ by applying both equations (14) and (15).

Afterwards, the function `correct` will get the predicted Mean and predicted Covariance and calculate the corrected Mean μ_k and the corrected Covariance ε_k by applying equations (16), (17) and (18).

Having the correction step succesfully implemented, a file is generated contating the corrected robot's state and the landmarks detected by using the following commands.

```
f.write("F %f %f %f\n" % \
            tuple(kf.state[0:3] +
            [scanner_displacement * cos(kf.state[2]),
            scanner_displacement*sin(kf.state[2]),0.0]))


write_cylinders(f, "D C", [(obs[2][0], obs[2][1])
                           for obs in observations])
```

In order to represent the output robot path and landmark locations a simulator is devloped using python programming *legofile_viewer.py* included in Appendix C which uses a library *lego_robot.py* included in Appendix D.

The simulator provides a GUI interface as shown in Figure 10 to represent the data where data text files are loaded through the interface then processed by the code and represented. Further information about the simulator will be provided in the results section.

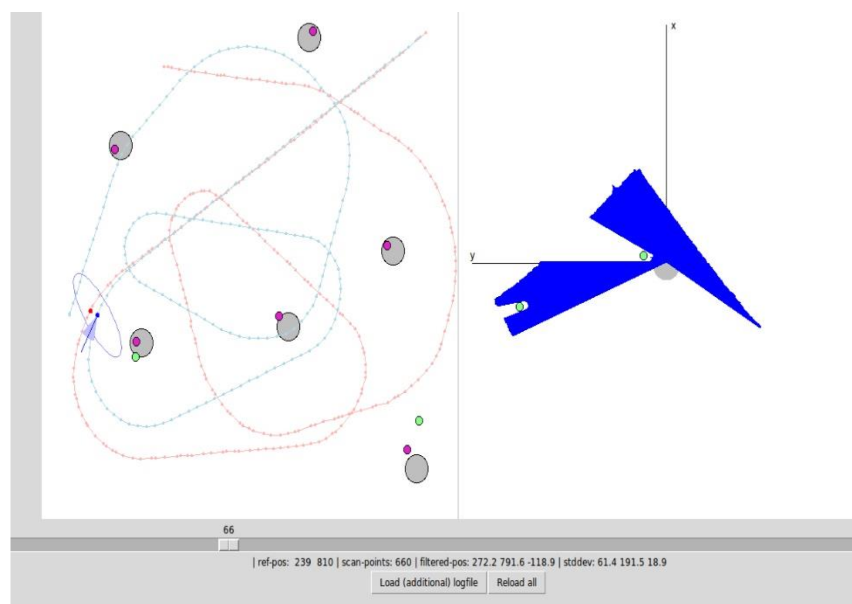

Figure 10 : Python Simulator GUI

3.1.2 Hardware Implementation

Having the python code for Extended Kalman filter finalized and tested, the final step is to implement the Extended Kalman filter algorithm on a hardware platform to prove it can be applied in real life applications. The goal of this section is to establish a test bench for the python code developed in the previous section. The inputs required for this test are the robot's odometry, laser range finder's scans and a reference robot's track at different intervals of time.

Having these requirements identified, the test bench is designed of two main parts. The first part is the arena or environment with various landmarks where the robot will travel through and collect its inputs. The second part is the robot platform to go through the arena and collect the data from the laser range finder and the robot's odometry sensor.

A. Arena

The arena dimensions, as shown in Figure 11, is 2 x 2 meters containing 5 landmarks of dimensions 0.6 x 0.3 meters. As the robot travels through the environment, it will observe the located landmarks and hence be able to localize itself and map the locations of these landmarks in the environment.

The real robot arena which is used in the testing is shown in Figure 12.

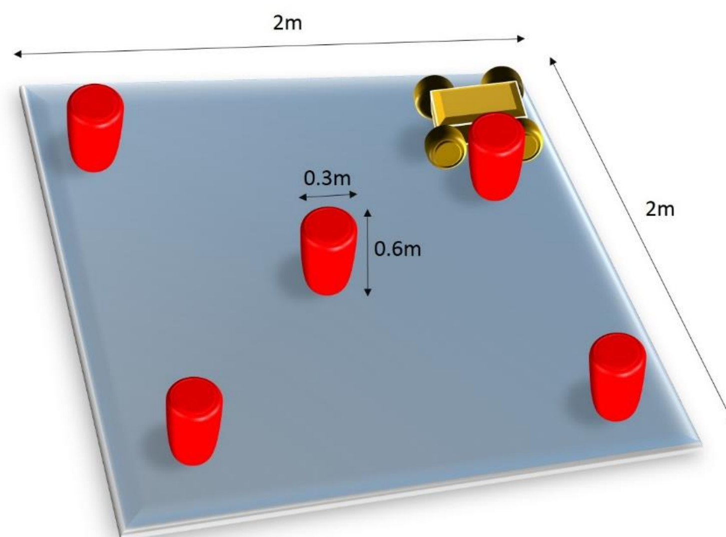

Figure 11: Arena presentation

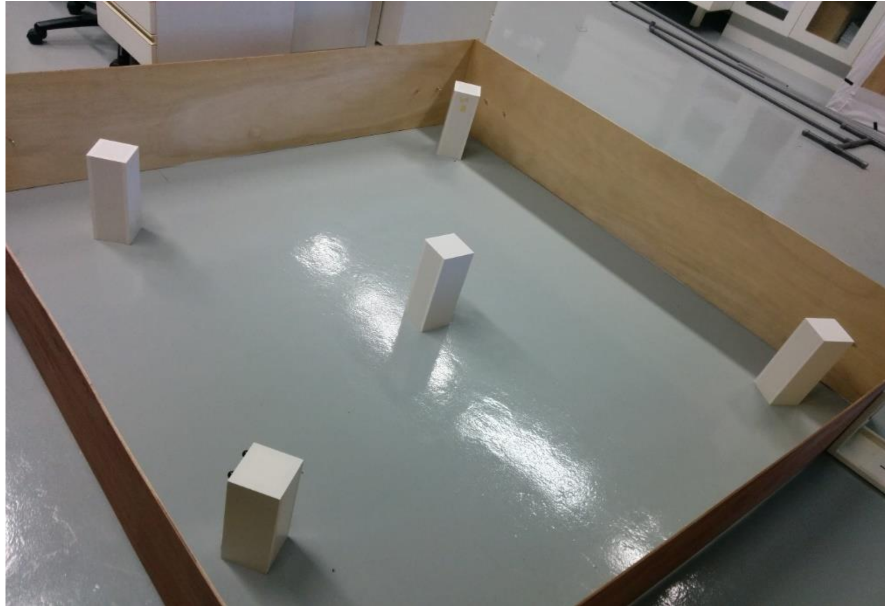

Figure 12 : Real Arena

Added to this arena setup, a static webcam is located above the arena to track the robot path using colour detection. The camera is connected to the laptop and using the opencv code in Appendix E the pixel locations of the robot at different intervals of time are logged.Having the robot pixel locations, a relation between the actual coordinates in meters and the pixel coordinates is formulated as follows:

$$X_{in\ meters} = \frac{Width_{of\ arena} * \ X_{in\ pixels}}{Width_{of\ image}} \quad (21)$$

$$Y_{in\ meters} = \frac{Length_{of\ arena} * \ Y_{in\ pixels}}{Length_{of\ image}} \quad (22)$$

Hence having the robot real path through the arena during the test run which will be used for verification in the results section. A snapshot of the overhead camera showing its field of view is shown in Figure 13.

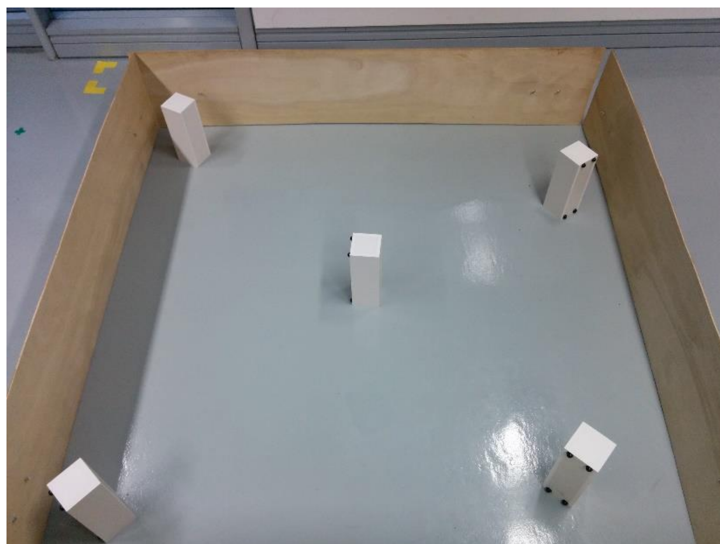

Figure 13 : Snapshot from the overhead camera

B. Robot Platform

The robot platform is designed for the purpose of data collection and logging where the data to be collected are the laser range finder scans and the wheel encoders at different intervals of time. The robot travels through the arena using manual controls given by the user and after the test run is finalized the data are fetched from the robot platform.

The robot platform consists of four main parts which are:

- 4WD Hercules mobile robotic platform.
- URG-04LX-UG01 Laser range finder.
- Raspberry Pi model B.
- Hercules Dual 15A 6-20V Motor Controller.

In the preceding sections, each of these hardware components will be explained briefly followed by how all of these components are integrated together.

C. 4WD Hercules mobile robotic platform

The Hercules mobile robotic platform, shown in Figure 14, is the chassis of the robot platform used in the test bench, the Hercules platform also contains four dc motors with magnetic wheel encoders installed.

Figure 14: 4WD Hercules mobile robotic platform

D. URG-04LX-UG01 Laser range finder

The Laser scanner URG-04LX-UG01, as shown in Figure 15, is a low voltage laser scanner designed for robotics application. The sensor has a range of vision of 240 degrees and measuring are in the range of 20 to 5600mm.

Figure 15: URG-04LX-UG01 Laser range finder

Model No.	URG-04LX-UG01
Power source	5VDC±5%(USB Bus power)
Light source	Semiconductor laser diode(λ =785nm), Laser safety class 1
Measuring area	20 to 5600mm(white paper with 70mm×70mm), 240°
Accuracy	60 to 1,000mm : ±30mm, 1,000 to 4,095mm : ±3% of measurement
Angular resolution	Step angle : approx. 0.36° (360° /1,024 steps)
Scanning time	100ms/scan
Noise	25dB or less
Interface	USB2.0/1.1[Mini B](Full Speed)
Command System	SCIP Ver.2.0
Ambient illuminance*1	Halogen/mercury lamp: 10,000Lux or less, Florescent: 6000Lux(Max)
Ambient temperature/humidity	-10 to +50 degrees C, 85% or less(Not condensing, not icing)
Vibration resistance	10 to 55Hz, double amplitude 1.5mm each 2 hour in X, Y and Z directions
Impact resistance	196m/s^2, Each 10 time in X, Y and Z directions
Weight	Approx. 160g

Table 1: URG-04LX-UG01 Laser scanner specifications

E. Raspberry Pi model B

Raspberry pi model B, shown in Figure 16, is one of a series of credit card-sized single-board computers developed by the Raspberry Pi foundation in UK. The board is based on Broadcom System on chip which contains a 700 MHZ ARM processor Video Core IV GPU, and RAM. It has a Level 1 cache of 16 KB and a Level 2 cache of 128 KB.

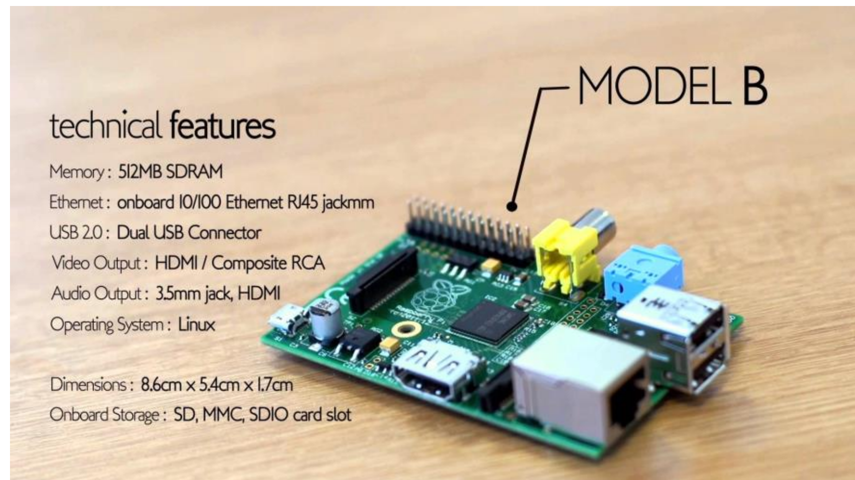

Figure 16 : Raspberry PI Model B specifications

F. Hercules Dual 15A 6-20V Motor Controller

The Hercules Dual 15A 6-20V Motor Controller, shown in Figure 17, is a high current motor driver with Arduino based micro controller and half bridge as motor drive circuit. The controller also support encoder and servo modules and can be programmed using Arduino IDE.

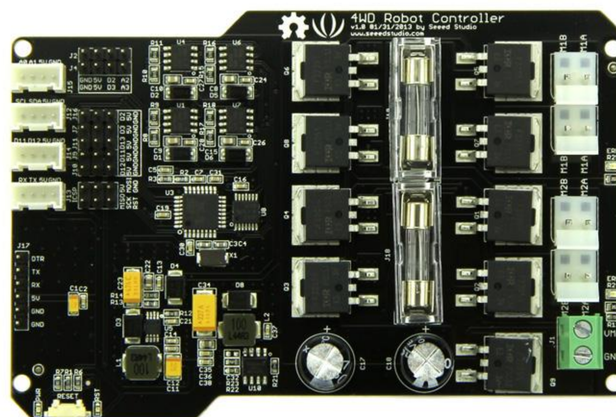

Figure 17 : Hercules Dual 15A 6-20V Motor Controller

3.2 Hardware Integration

All the hardware components explained in the previous sections are combined together to build the robot platform as shown in Figure 18. The main goal of building the robot platform is to have a manual control robot able to travel through the environment and log readings from laser scanner and wheel encoders at different time intervals.

In this robot platform, the raspberry pi is used as the main link between all the components where it is connected to the laptop using secure shell network protocol to manually control the robot's motion through the environment. The raspberry pi use serial communication to send the motion commands to the Hercules Dual 15A 6-20V Motor Controller and receive the encoder values of the wheels.

The Hercules Dual 15A 6-20V Motor Controller is programmed using Arduino programming language code, shown in Appendix F, which interpret keyboard inputs as motion commands and serially send the wheel encoder values at certain time intervals of time.

Finally, the raspberry pi serially communicate with the URG-04LX-UG01 Laser range finder using c coding , shown in Appendix G, to get the readings of the sensor in 240 degree range then log it on its memory. All of these parts are governed using shell scripts written to run those different codes at certain time intervals to collect the robot's readings in the environment.

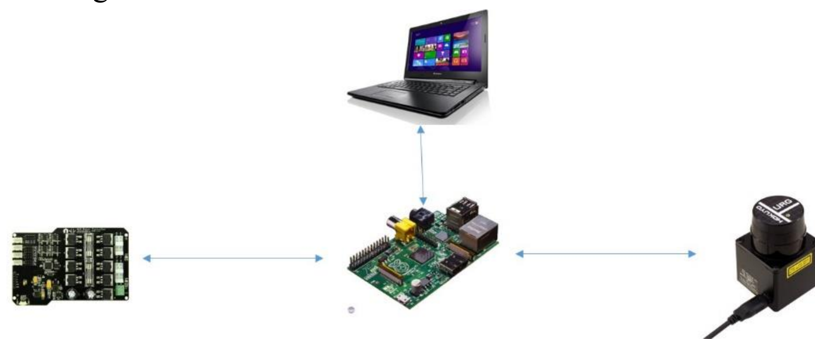

Figure 18 : Hardware Integration

Having the laser scans and the wheel encoder readings collected, the hardware implementation is finalized and these readings are provided as inputs to the python code then the results are compared with the robot real path, which is logged using the overhead camera, to see the effectiveness of the Extended Kalman filter implementation in real life applications.

CHAPTER 4
RESULTS & DISCUSSIONS

The Extended Kalman filter is developed as a solution for the SLAM problem as shown before and a python code is developed for the mathematical model whose inputs are the laser scans from the environment and the wheels odometry at different intervals of time. Using these inputs, the output corrected track and the corrected locations of the landmarks is generated by the code in a text file format.

In order to have a better representation of the results, a python simulation code *legofile_viewer.py* is used for better data analysis. The output of the simulator is shown in Figure 19 where:

- The **Red track** represent the actual track of the robot.
- The **Blue track** represent the estimated track from the encoder values.
- The **Arrow** represent the robot's orientation and the **Blue circle** represent the robot's uncertainty of its position/
- The **Grey cylinders** represents the arena landmarks' locations while the Red cylinders the expected landmarks' locations.
- The **Blue X-Y coordinates** represents the sensor's reading at each point.

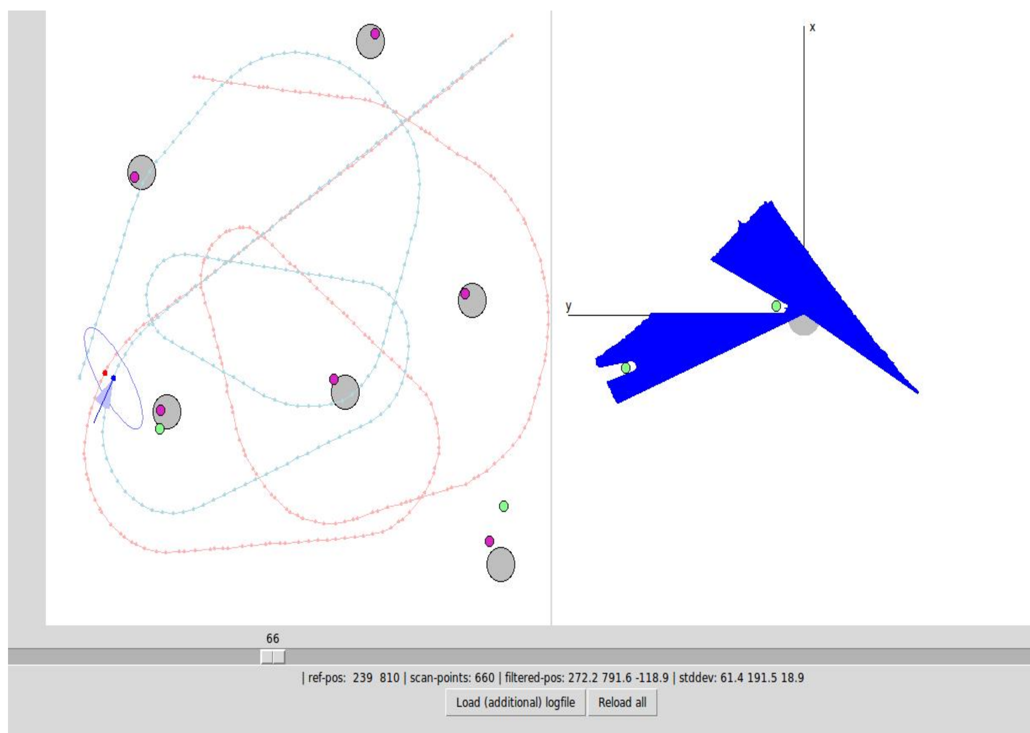

Figure 19: Python Logfile_viewer simulator output

In the preceding sections the results for both the software implementation and the hardware implementation will be discussed in details. The accuracy of the results will be decided based on two aspects which are:

- The degree of correction of the corrected path compared to the robot's odometry path.
- The degree of deviation of the corrected path from the robot's real path.

4.1 Software implementation results

In this section the input used is an online database. Based on the robot's odometry, Figure 20 shows the difference between the robot odometry path (blue path) and the robot's real path (red path).It's clear that the odometry path have a high degree of deviation in both shape and orientation than the real path.

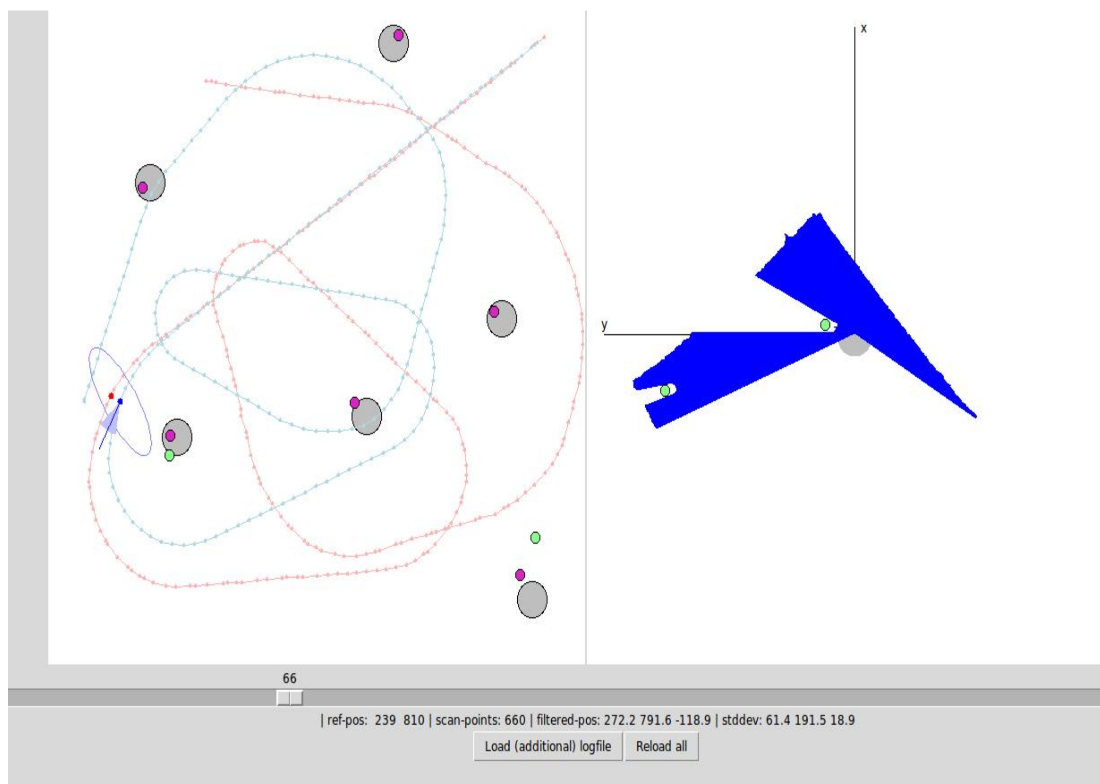

Figure 20 : Software implementation Before Correction

While after applying the correction, Figure 21 shows the corrected path (blue path) compared to the real path (red path). Compared to Figure 20 its clear the degree of correction in both shape and orientation between the odometry path and the corrected path. Finally, the result of the correction shows minimal deviation from the real robot path which is considered an acceptable result.

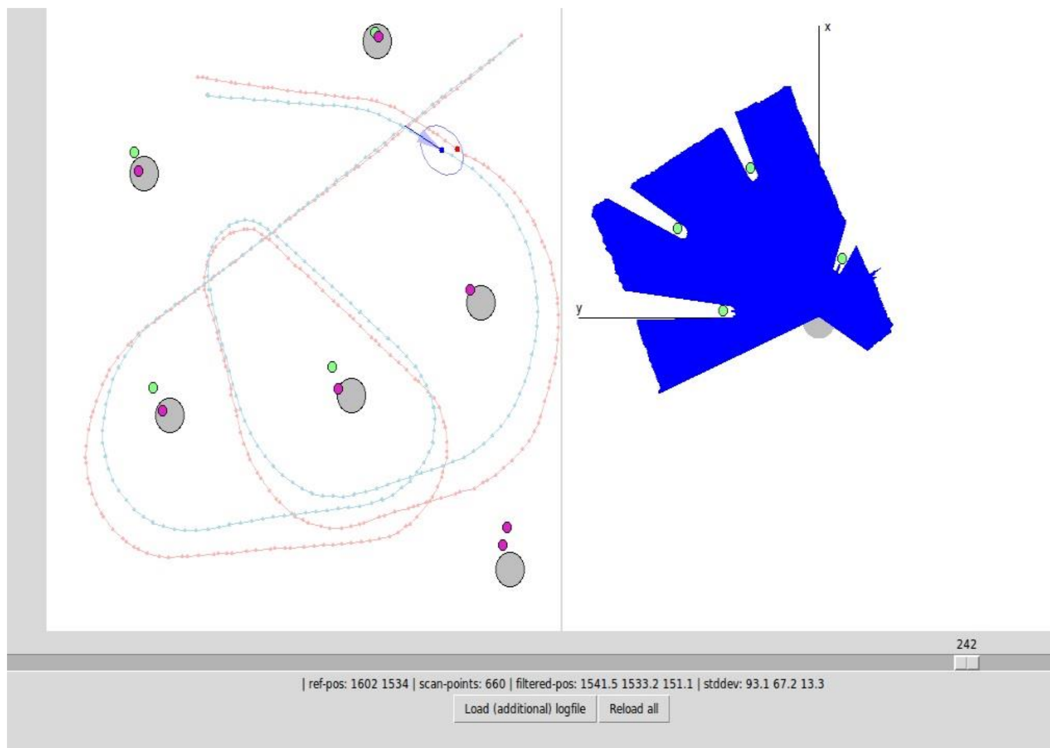

Figure 21 : Software implementation After Correction

4.2 Hardware Implementation Results

In this section, the inputs used are the data captured by the laser scanner, wheel encoders and overhead camera in the test bench designed for testing the Extended Kalman filter operation. By examining Figure 22, its clear the high amount of deviation of the odometry path (blue path) compared to the real robot's path (red path) as the result shows incomplete track due to data loss.

By comparing the odometry path shown in Figure 20 to that shown in Figure 22, its clear that the Hardware implementation odometry track has much more noise than that of the Software implementation which is due to the difference in encoder quality between both situations.

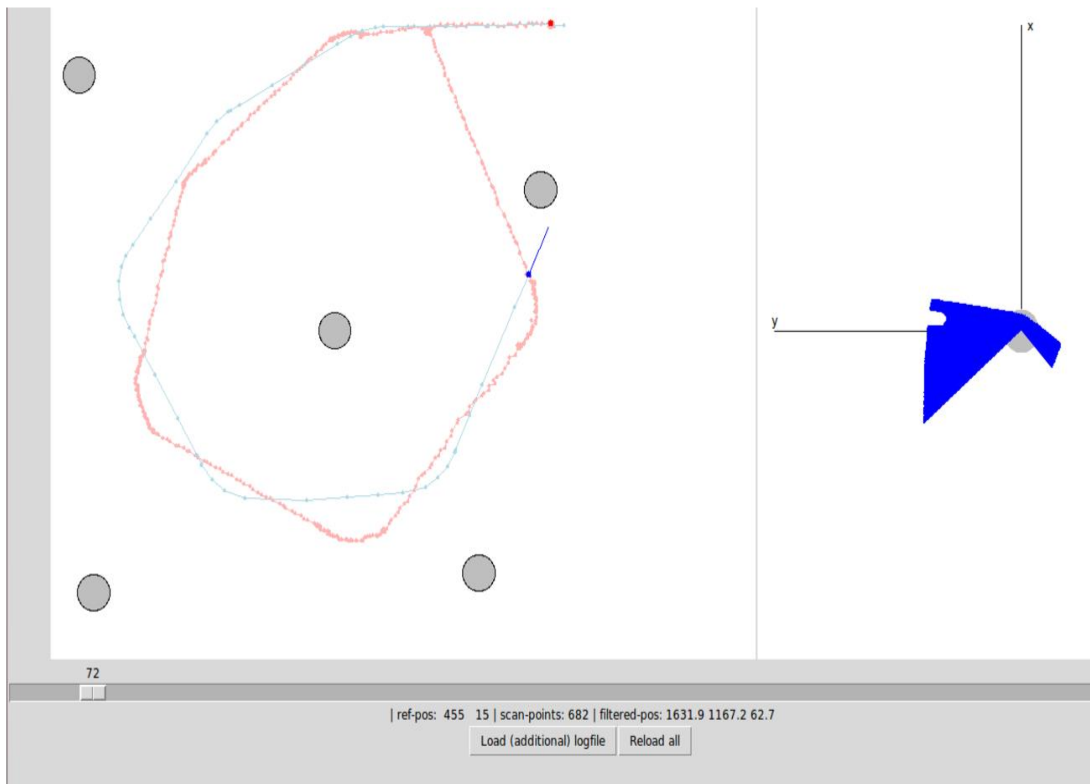

Figure 22 : Hardware implementation Before Correction

Finally, the result of the correction is shown in Figure 23 where the degree of correction is clear when compared to Figure 22. On the other hand, the corrected path (blue path) show a high degree of deviation if compared to the robot's real path (red path) but this deviation is acceptable given the fact that the odometry input was highly noisy and needed a high degree of correction.

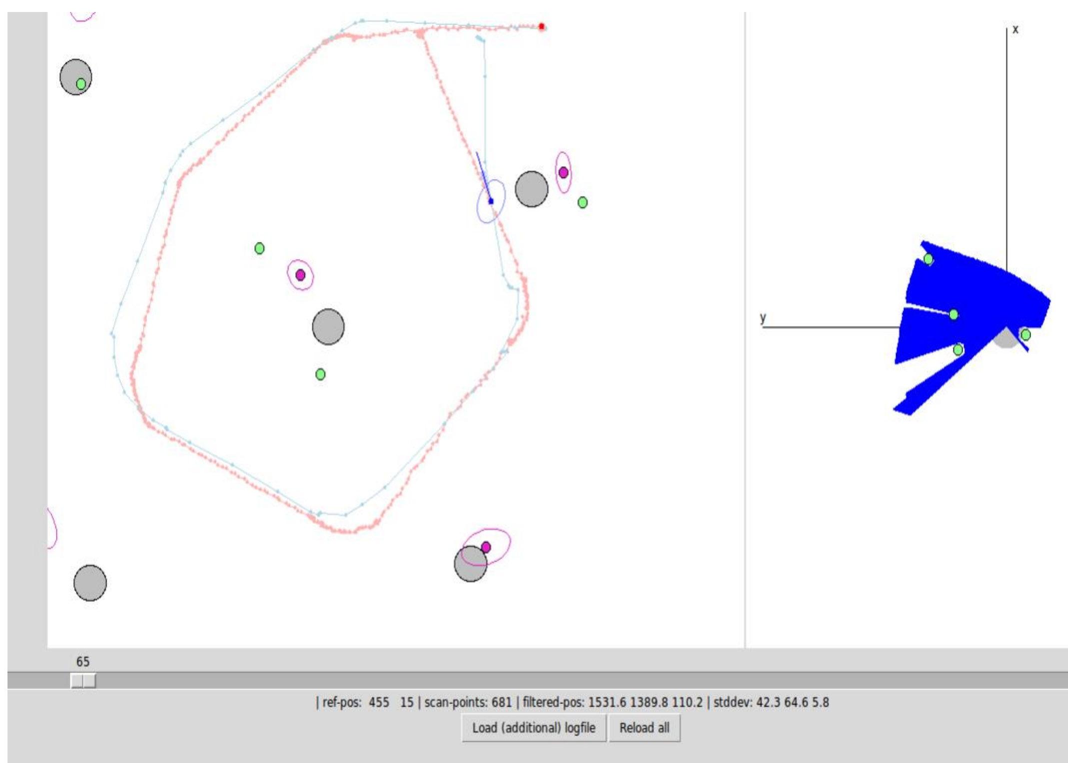

Figure 23 : Hardware implementation After Correction

CHAPTER 5
CONCLUSION & RECOMMENDATION

The aim of this project was to apply the Extended Kalman filter mathematical model on a mobile robot and test its efficiency in real life applications. This goal have been approached in two stages which are software implementation where a python code was written to emulate the EKF mathematical model and hardware implementation where a test bench was created to test the application of the model.

The results showed that the EKF is a viable solution for the SLAM problem and can be used in robotics applications taking in consideration that the results are highly affected by the amount of noise in the inputs either the wheel odometry or the laser scanner readings. Also the results is affected by the time interval between readings, as the time increase the accuracy of the model decrease as less samples are gathered providing less information for correction.

Therefore, it’s recommended in future applications to use high quality laser scanner and wheel encoders to avoid noise in the inputs and to decrease the time interval between the sensors readings in order to have the best results.

References

[1] O. Matsebe, M. Namoshe, and N. Tlale, *Basic extended Kalman filter: simultaneous localisation and mapping*: INTECH, 2010.
[2] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*: MIT Press, 2005.
[3] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I," *Robotics & Automation Magazine, IEEE,* vol. 13, pp. 99-110, 2006.
[4] S. Riisgaard and M. R. Blas, "SLAM for Dummies," *A Tutorial Approach to Simultaneous Localization and Mapping,* vol. 22, p. 126, 2003.

Appendix A
EKF_SLAM.py

```
from math import sin, cos, pi, atan2, sqrt
from numpy import *
from EKF_Slam_Library import encoder,write_error_ellipses,
write_cylinders, laser_scanner, get_observations

class ExtendedKalmanFilterSLAM:
    def __init__(self, state, covariance,
                 robot_width, scanner_displacement,
                 control_motion_factor, control_turn_factor,
                measurement_distance_stddev,
measurement_angle_stddev):

        # The state. This is the core data of the Kalman filter.
        self.state = state
        self.covariance = covariance

        # Some constants.
        self.robot_width = robot_width
        self.scanner_displacement = scanner_displacement
        self.control_motion_factor = control_motion_factor
        self.control_turn_factor = control_turn_factor
        self.measurement_distance_stddev =
measurement_distance_stddev
        self.measurement_angle_stddev = measurement_angle_stddev

        # Currently, the number of landmarks is zero.
        self.number_of_landmarks = 0

    @staticmethod
    def g(state, control, w):
        x, y, theta = state
        l, r = control
        if r != l:
            alpha = (r - l) / w
            rad = l/alpha
            g1 = x + (rad + w/2.)*(sin(theta+alpha) - sin(theta))
            g2 = y + (rad + w/2.)*(-cos(theta+alpha) + cos(theta))
            g3 = (theta + alpha + pi) % (2*pi) - pi
        else:
            g1 = x + l * cos(theta)
            g2 = y + l * sin(theta)
            g3 = theta

        return array([g1, g2, g3])

    @staticmethod
    def dg_dstate(state, control, w):
        theta = state[2]
        l, r = control
        if r != l:
            alpha = (r-l)/w
            theta_ = theta + alpha
            rpw2 = l/alpha + w/2.0
            m = array([[1.0, 0.0, rpw2*(cos(theta_) - cos(theta))],
                       [0.0, 1.0, rpw2*(sin(theta_) - sin(theta))],
                       [0.0, 0.0, 1.0]])
```

```
        else:
            m = array([[1.0, 0.0, -l*sin(theta)],
                       [0.0, 1.0,  l*cos(theta)],
                       [0.0, 0.0,  1.0]])
        return m

    @staticmethod
    def dg_dcontrol(state, control, w):
        theta = state[2]
        l, r = tuple(control)
        if r != l:
            rml = r - l
            rml2 = rml * rml
            theta_ = theta + rml/w
            dg1dl = w*r/rml2*(sin(theta_)-sin(theta))  -
(r+l)/(2*rml)*cos(theta_)
            dg2dl = w*r/rml2*(-cos(theta_)+cos(theta)) -
(r+l)/(2*rml)*sin(theta_)
            dg1dr = (-w*l)/rml2*(sin(theta_)-sin(theta)) +
(r+l)/(2*rml)*cos(theta_)
            dg2dr = (-w*l)/rml2*(-cos(theta_)+cos(theta)) +
(r+l)/(2*rml)*sin(theta_)

        else:
            dg1dl = 0.5*(cos(theta) + l/w*sin(theta))
            dg2dl = 0.5*(sin(theta) - l/w*cos(theta))
            dg1dr = 0.5*(-l/w*sin(theta) + cos(theta))
            dg2dr = 0.5*(l/w*cos(theta) + sin(theta))

        dg3dl = -1.0/w
        dg3dr = 1.0/w
        m = array([[dg1dl, dg1dr], [dg2dl, dg2dr], [dg3dl, dg3dr]])

        return m

    def predict(self, control):
        """The prediction step of the Kalman filter."""
        # covariance' = G * covariance * GT + R
        # where R = V * (covariance in control space) * VT.
        # Covariance in control space depends on move distance.
        G3 = self.dg_dstate(self.state, control, self.robot_width)
        left, right = control
        left_var = (self.control_motion_factor * left)**2 +\
                   (self.control_turn_factor * (left-right))**2
        right_var = (self.control_motion_factor * right)**2 +\
                    (self.control_turn_factor * (left-right))**2
        control_covariance = diag([left_var, right_var])
        V = self.dg_dcontrol(self.state, control, self.robot_width)
        R3 = dot(V, dot(control_covariance, V.T))


        G = eye(3+2*self.number_of_landmarks)
        G[0:3, 0:3] = G3

        R = zeros((3+2*self.number_of_landmarks,
3+2*self.number_of_landmarks))
        R[0:3, 0:3] = R3

        # new covariance matrix self.covariance.
        self.covariance = dot(G, dot(self.covariance, G.T)) + R  #
```

```
        # state' = g(state, control)
        self.state[0:3] = self.g(self.state[0:3], control,
self.robot_width)

    def add_landmark_to_state(self, initial_coords):

        self.number_of_landmarks += 1
        state_dash = zeros(3+2*self.number_of_landmarks)
        state_dash[0:3+2*(self.number_of_landmarks-1)] = self.state
        state_dash[3+2*self.number_of_landmarks-
2:3+2*self.number_of_landmarks] = initial_coords
        self.state = state_dash

        covariance_dash =
zeros((3+2*self.number_of_landmarks,3+2*self.number_of_landmarks))
        fill_diagonal(covariance_dash, 10**10)
        covariance_dash[0:3+2*(self.number_of_landmarks-1),
0:3+2*(self.number_of_landmarks-1)] = self.covariance
        self.covariance = covariance_dash

        return self.number_of_landmarks-1

    @staticmethod
    def h(state, landmark, scanner_displacement):
        """Takes a (x, y, theta) state and a (x, y) landmark, and
returns the
           measurement (range, bearing)."""
        dx = landmark[0] - (state[0] + scanner_displacement *
cos(state[2]))
        dy = landmark[1] - (state[1] + scanner_displacement *
sin(state[2]))
        r = sqrt(dx * dx + dy * dy)
        alpha = (atan2(dy, dx) - state[2] + pi) % (2*pi) - pi

        return array([r, alpha])

    @staticmethod
    def dh_dstate(state, landmark, scanner_displacement):
        theta = state[2]
        cost, sint = cos(theta), sin(theta)
        dx = landmark[0] - (state[0] + scanner_displacement * cost)
        dy = landmark[1] - (state[1] + scanner_displacement * sint)
        q = dx * dx + dy * dy
        sqrtq = sqrt(q)
        drdx = -dx / sqrtq
        drdy = -dy / sqrtq
        drdtheta = (dx * sint - dy * cost) * scanner_displacement /
sqrtq
        dalphadx =  dy / q
        dalphady = -dx / q
        dalphadtheta = -1 - scanner_displacement / q * (dx * cost +
dy * sint)

        return array([[drdx, drdy, drdtheta],
                      [dalphadx, dalphady, dalphadtheta]])
    @staticmethod
    def get_error_ellipse(covariance):
        """Return the position covariance (which is the upper 2x2
submatrix)
           as a triple: (main_axis_angle, stddev_1, stddev_2), where
```

```
            main_axis_angle is the angle (pointing direction) of the
main axis,
            along which the standard deviation is stddev_1, and
stddev_2 is the
            standard deviation along the other (orthogonal) axis."""
        eigenvals, eigenvects = linalg.eig(covariance[0:2,0:2])
        angle = atan2(eigenvects[1,0], eigenvects[0,0])
        return (angle, sqrt(eigenvals[0]), sqrt(eigenvals[1]))

    def correct(self, measurement, landmark_index):
        """The correction step of the Kalman filter."""

        landmark = self.state[3+2*landmark_index :
3+2*landmark_index+2]
        H3 = self.dh_dstate(self.state, landmark,
self.scanner_displacement)


        H = zeros((2,3+2*self.number_of_landmarks))
        H[0:2, 0:3] = H3
        H[0:2, 3+2*landmark_index:3+2*landmark_index+2] = -H3[0:2,
0:2]



        Q = diag([self.measurement_distance_stddev**2,
                  self.measurement_angle_stddev**2])
        K = dot(self.covariance,
                dot(H.T, linalg.inv(dot(H, dot(self.covariance,
H.T)) + Q)))
        innovation = array(measurement) -\
                     self.h(self.state, landmark,
self.scanner_displacement)
        innovation[1] = (innovation[1] + pi) % (2*pi) - pi
        self.state = self.state + dot(K, innovation)
        self.covariance = dot(eye(size(self.state)) - dot(K, H),
                              self.covariance)

    def get_landmarks(self):
        """Returns a list of (x, y) tuples of all landmark
positions."""
        return ([(self.state[3+2*j], self.state[3+2*j+1])
                 for j in range(self.number_of_landmarks)])

    def get_landmark_error_ellipses(self):
        """Returns a list of all error ellipses, one for each
landmark."""
        ellipses = []
        for i in range(self.number_of_landmarks):
            j = 3 + 2 * i
            ellipses.append(self.get_error_ellipse(
                self.covariance[j:j+2, j:j+2]))
        return ellipses
```

```
if __name__ == '__main__':
    # Robot constants.
    scanner_displacement = 0.0
    ticks_to_mm = 3.59
    robot_width = 250.0

    # Cylinder extraction and matching constants.
    minimum_valid_distance = 20.0
    depth_jump = 100.0
    cylinder_offset = 90.0
    max_cylinder_distance = 900.0

    # Filter constants.
    control_motion_factor = 0.3  # Error in motor control.
    control_turn_factor = 1.2 # Additional error due to slip when
turning.
    measurement_distance_stddev = 250.0  # Distance measurement
error of cylinders.
    measurement_angle_stddev = 10. / 180.0 * pi  # Angle measurement
error.

    # Arbitrary start position.
    initial_state = array([1720.0, 1929.0, 180.0 / 180.0 * pi])

    # Covariance at start position.
    initial_covariance = zeros((3,3))

    # Setup filter.
    kf = ExtendedKalmanFilterSLAM(initial_state, initial_covariance,
                                  robot_width, scanner_displacement,
                                  control_motion_factor,
control_turn_factor,
                                  measurement_distance_stddev,
measurement_angle_stddev)

    # Read data.
    ticks = encoder()

    f = open("//home//ahmed//work//SLAM//robot4_motors.txt")
    lines = len(f.readlines())
    f.seek(0.0)
    f.close()



    # This is the EKF SLAM loop.
    f = open("ekf_slam_correction.txt", "w")
    for t in range(lines-1):
        #Predection
        control = []
        control.append(ticks[(t, 0)]*ticks_to_mm)
        control.append(ticks[(t, 1)]*ticks_to_mm)
        kf.predict(control)

        # Correction.
        observations = get_observations(
            laser_scanner(t),
            depth_jump, minimum_valid_distance, cylinder_offset,
            kf, max_cylinder_distance)
```

```
        for obs in observations:
            measurement, cylinder_world, cylinder_scanner,
cylinder_index = obs
            if cylinder_index == -1:
                cylinder_index =
kf.add_landmark_to_state(cylinder_world)
            kf.correct(measurement, cylinder_index)

        # End of EKF SLAM - from here on, data is written.

        # Output the center of the scanner, not the center of the
robot.
        f.write("F %f %f %f\n" % \
            tuple(kf.state[0:3] + [scanner_displacement *
cos(kf.state[2]),
                                   scanner_displacement *
sin(kf.state[2]),
                                   0.0]))
        # Write covariance matrix in angle stddev1 stddev2 stddev-
heading form
        e =
ExtendedKalmanFilterSLAM.get_error_ellipse(kf.covariance)
        f.write("E %f %f %f %f\n" % (e +
(sqrt(kf.covariance[2,2]),)))
        # Write estimates of landmarks.
        write_cylinders(f, "W C", kf.get_landmarks())
        # Write error ellipses of landmarks.
        f.write("W E\t")
        for e in kf.get_landmark_error_ellipses():
            f.write("%.3f %.1f %.1f\t" % e)
        f.write("\n")
        # Write cylinders detected by the scanner.
        write_cylinders(f, "D C", [(obs[2][0], obs[2][1])
                                   for obs in observations])

    f.close()
```

Appendix B

EKF_Slam_Library.py

```
from math import sin, cos, pi
from pylab import *
from numpy import *
from lego_robot import LegoLogfile

def encoder():
    f = open("//home//ahmed//work//SLAM//robot4_motors.txt")
    lines = len(f.readlines())
    f.seek(0.0)
    left_list = []
    right_list = []
    encoder_ticks = zeros(shape=(lines, 2))
    for l in f:
        sp = l.split()
        left_list.append(int(sp[2]))
        right_list.append(int(sp[1]))

    for u in range(lines):
        if u == 0:
            encoder_ticks[(u, 0)] = (left_list[u]-left_list[0])
            encoder_ticks[(u, 1)] = (right_list[u]-right_list[0])
        else:
            encoder_ticks[(u, 0)] = (left_list[u]-left_list[u-1])
            encoder_ticks[(u, 1)] = (right_list[u]-right_list[u-1])
    f.close()
    return encoder_ticks


def filter_step(old_pose, motor_ticks, ticks_to_mm, robot_width,
scanner_displacement):

        if motor_ticks[0] == motor_ticks[1]:
            x =
old_pose[0]+(motor_ticks[0]*ticks_to_mm*cos(old_pose[2]))
            y =
old_pose[1]+(motor_ticks[0]*ticks_to_mm*sin(old_pose[2]))
            theta = old_pose[2]
            return(x, y, theta)

        else:
            old_theta = old_pose[2]
            old_x = old_pose[0]
            old_y = old_pose[1]

            old_x -= cos(old_theta) * scanner_displacement
            old_y -= sin(old_theta) * scanner_displacement

            l = motor_ticks[0] * ticks_to_mm
            r = motor_ticks[1] * ticks_to_mm
            alpha = (r - l) / robot_width
            R = l / alpha
            new_theta = (old_theta + alpha) % (2*pi)
            new_x = old_x + (R + robot_width/2.0) * (sin(new_theta)
- sin(old_theta))
            new_y = old_y + (R + robot_width/2.0) * (-cos(new_theta)
+ cos(old_theta))
```

```
            new_x += cos(new_theta) * scanner_displacement
            new_y += sin(new_theta) * scanner_displacement

            return (new_x, new_y, new_theta)

def laser_scanner(get):
    f = open("//home//ahmed//work//SLAM//robot4_scan.txt")
    line_no = len(f.readlines())
    f.seek(0)
    values = []

    for yu in range(line_no):
        readings = f.readline()
        if yu == get:
            values = readings.split()
    f.close()
    return values[1:682]

def compute_derivative(scan, min_dist):
    jumps = [0]
    for i in range(1, len(scan) - 1):
        l = float(scan[i-1])
        r = float(scan[i+1])
        if l > min_dist and r > min_dist:
            derivative = (r - l) / 2.0
            jumps.append(derivative)
        else:
            jumps.append(0)
    jumps.append(0)
    return jumps

def find_cylinders(scan, scan_derivative, jump, min_dist):
    cylinder_list = []
    on_cylinder = False
    sum_ray, sum_depth, rays = 0.0, 0.0, 0

    for i in range(len(scan_derivative)):
        if scan_derivative[i] < -jump:
            # Start a new cylinder, independent of on_cylinder.
            on_cylinder = True
            sum_ray, sum_depth, rays = 0.0, 0.0, 0
        elif scan_derivative[i] > jump:
            # Save cylinder if there was one.
            if on_cylinder and rays:
                cylinder_list.append((sum_ray/rays, sum_depth/rays))
            on_cylinder = False
        # Always add point, if it is a valid measurement.
        elif float(scan[i]) > min_dist:
            sum_ray += i
            sum_depth += float(scan[i])
            rays += 1
    return cylinder_list

def compute_cartesian_coordinates(cylinders, cylinder_offset,
mounting_angle):
    result = []
    for c in cylinders:
        bearing = (c[0] - 341.0) * 0.006275923151543 +
mounting_angle
        ranges = c[1] + cylinder_offset
```

```
        x_coordinate = ranges * cos(bearing)
        y_coordinate = ranges * sin(bearing)
        result.append((x_coordinate, y_coordinate))
    return result

def compute_scanner_cylinders(scan, depth_jump,
minimum_valid_distance, cylinder_offset, mounting_angle):
    der = compute_derivative(scan, minimum_valid_distance)
    cylinders = find_cylinders(scan, der, depth_jump,
minimum_valid_distance)
    scanner_cylinders = compute_cartesian_coordinates(cylinders,
cylinder_offset, mounting_angle)
    return scanner_cylinders

def write_cylinders(file_desc, line_header, cylinder_list):
    file_desc.write(line_header)
    file_desc.write("\t")
    for c in cylinder_list:
        file_desc.write("%.1f %.1f\t" % c)
    file_desc.write("\n")

def get_observations(scan, jump, min_dist, cylinder_offset,
                     robot,
                     max_cylinder_distance):
    der = compute_derivative(scan, min_dist)
    cylinders = find_cylinders(scan, der, jump, min_dist)
    # Compute scanner pose from robot pose.
    scanner_pose = (
        robot.state[0] + cos(robot.state[2]) *
robot.scanner_displacement,
        robot.state[1] + sin(robot.state[2]) *
robot.scanner_displacement,
        robot.state[2])

    # For every detected cylinder which has a closest matching pole
in the
    # cylinders that are part of the current state, put the
measurement
    # (distance, angle) and the corresponding cylinder index into
the result list.
    result = []
    for c in cylinders:
        # Compute the angle and distance measurements.
        angle = LegoLogfile.beam_index_to_angle(c[0])
        distance = c[1] + cylinder_offset
        # Compute x, y of cylinder in world coordinates.
        xs, ys = distance*cos(angle), distance*sin(angle)
        x, y = LegoLogfile.scanner_to_world(scanner_pose, (xs, ys))
        # Find closest cylinder in the state.
        best_dist_2 = max_cylinder_distance * max_cylinder_distance
        best_index = -1
        for index in range(robot.number_of_landmarks):
            pole_x, pole_y = robot.state[3+2*index : 3+2*index+2]
            dx, dy = pole_x - x, pole_y - y
            dist_2 = dx * dx + dy * dy
            if dist_2 < best_dist_2:
                best_dist_2 = dist_2
                best_index = index
        # Always add result to list. Note best_index may be -1.
        result.append(((distance, angle), (x, y), (xs, ys),
best_index))
```

```
    return result

def write_error_ellipses(file_desc, line_header,
error_ellipse_list):
    file_desc.write(line_header)
    file_desc.write("\t")
    for e in error_ellipse_list:
        file_desc.write("%.3f %.1f %.1f\n" % e)
```

Appendix C
legofile_viewer.py

```
# Python routines to inspect a ikg LEGO robot logfile.
# Author: Claus Brenner, 28 OCT 2012
from Tkinter import *
import tkFileDialog
from lego_robot import *
from math import sin, cos, pi, ceil

# The canvas and world extents of the scene.
# Canvas extents in pixels, world extents in millimeters.
canvas_extents = (600, 600)
world_extents = (1800.0, 2000.0)

# The extents of the sensor canvas.
sensor_canvas_extents = canvas_extents

# The maximum scanner range used to scale scan measurement drawings,
# in millimeters.
max_scanner_range = 4000.0

class DrawableObject(object):
    def draw(self, at_step):
        print "To be overwritten - will draw a certain point in
time:", at_step

    def background_draw(self):
        print "Background draw."

    @staticmethod
    def get_ellipse_points(center, main_axis_angle, radius1,
radius2,
                           start_angle = 0.0, end_angle = 2 * pi):
        """Generate points of an ellipse, for drawing (y axis
down)."""
        points = []
        ax = radius1 * cos(main_axis_angle)
        ay = radius1 * sin(main_axis_angle)
        bx = - radius2 * sin(main_axis_angle)
        by = radius2 * cos(main_axis_angle)
        N_full = 40  # Number of points on full ellipse.
        N = int(ceil((end_angle - start_angle) / (2 * pi) * N_full))
        N = max(N, 1)
        increment = (end_angle - start_angle) / N
        for i in xrange(N + 1):
            a = start_angle + i * increment
            c = cos(a)
            s = sin(a)
            x = c*ax + s*bx + center[0]
            y = - c*ay - s*by + center[1]
            points.append((x,y))
        return points


class Trajectory(DrawableObject):
    def __init__(self, points, canvas,
                 world_extents, canvas_extents,
                 standard_deviations = [],
```

```
                 point_size2 = 2,
                 background_color = "gray", cursor_color = "red",
                 position_stddev_color = "green", theta_stddev_color
= "#ffc0c0"):
        self.points = points
        self.standard_deviations = standard_deviations
        self.canvas = canvas
        self.world_extents = world_extents
        self.canvas_extents = canvas_extents
        self.point_size2 = point_size2
        self.background_color = background_color
        self.cursor_color = cursor_color
        self.position_stddev_color = position_stddev_color
        self.theta_stddev_color = theta_stddev_color
        self.cursor_object = None
        self.cursor_object2 = None
        self.cursor_object3 = None
        self.cursor_object4 = None

    def background_draw(self):
        if self.points:
            p_xy_only = []
            for p in self.points:
                self.canvas.create_oval(\
                    p[0]-self.point_size2, p[1]-self.point_size2,
                    p[0]+self.point_size2, p[1]+self.point_size2,
                    fill=self.background_color, outline="")
                p_xy_only.append(p[0:2])
            self.canvas.create_line(*p_xy_only,
fill=self.background_color)

    def draw(self, at_step):
        if self.cursor_object:
            self.canvas.delete(self.cursor_object)
            self.cursor_object = None
            self.canvas.delete(self.cursor_object2)
            self.cursor_object2 = None
        if at_step < len(self.points):
            p = self.points[at_step]
            # Draw position (point).
            self.cursor_object = self.canvas.create_oval(\
                p[0]-self.point_size2-1, p[1]-self.point_size2-1,
                p[0]+self.point_size2+1, p[1]+self.point_size2+1,
                fill=self.cursor_color, outline="")
            # Draw error ellipse.
            if at_step < len(self.standard_deviations):
                stddev = self.standard_deviations[at_step]
                # Note this assumes correct aspect ratio.
                factor = canvas_extents[0] / world_extents[0]
                points = self.get_ellipse_points(p, stddev[0],
                    stddev[1] * factor, stddev[2] * factor)
                if self.cursor_object4:
                    self.canvas.delete(self.cursor_object4)
                self.cursor_object4 = self.canvas.create_line(
                    *points, fill=self.position_stddev_color)
            if len(p) > 2:
                # Draw heading standard deviation.
                if at_step < len(self.standard_deviations) and\
                   len(self.standard_deviations[0]) > 3:
                    angle =
min(self.standard_deviations[at_step][3], pi)
```

```
                    points = self.get_ellipse_points(p, p[2], 30.0,
30.0,
                                                 -angle, angle)
                    points = [p[0:2]] + points + [p[0:2]]
                    if self.cursor_object3:
                        self.canvas.delete(self.cursor_object3)
                    self.cursor_object3 =
self.canvas.create_polygon(
                        *points, fill=self.theta_stddev_color)
                # Draw heading.
                self.cursor_object2 = self.canvas.create_line(p[0],
p[1],
                    p[0] + cos(p[2]) * 50,
                    p[1] - sin(p[2]) * 50,
                    fill = self.cursor_color)

class ScannerData(DrawableObject):
    def __init__(self, list_of_scans, canvas, canvas_extents,
scanner_range):
        self.canvas = canvas
        self.canvas_extents = canvas_extents
        self.cursor_object = None

        # Convert polar scanner measurements into xy form, in canvas
coords.
        # Store the result in self.scan_polygons.
        self.scan_polygons = []
        for s in list_of_scans:
            poly = [ to_sensor_canvas((0,0), canvas_extents,
scanner_range) ]
            i = 0
            for m in s:
                angle = LegoLogfile.beam_index_to_angle(i)
                x = m * cos(angle)
                y = m * sin(angle)
                poly.append(to_sensor_canvas((x,y), canvas_extents,
scanner_range))
                i += 1
            poly.append(to_sensor_canvas((0,0), canvas_extents,
scanner_range))
            self.scan_polygons.append(poly)

    def background_draw(self):
        # Draw x axis.
        self.canvas.create_line(
            self.canvas_extents[0]/2, self.canvas_extents[1]/2,
            self.canvas_extents[0]/2, 20,
            fill="black")
        self.canvas.create_text(
            self.canvas_extents[0]/2 + 10, 20, text="x" )
        # Draw y axis.
        self.canvas.create_line(
            self.canvas_extents[0]/2, self.canvas_extents[1]/2,
            20, self.canvas_extents[1]/2,
            fill="black")
        self.canvas.create_text(
            20, self.canvas_extents[1]/2 - 10, text="y" )
        # Draw big disk in the scan center.
        self.canvas.create_oval(
            self.canvas_extents[0]/2-20, self.canvas_extents[1]/2-
20,
```

```
            self.canvas_extents[0]/2+20,
self.canvas_extents[1]/2+20,
            fill="gray", outline="")

    def draw(self, at_step):
        if self.cursor_object:
            self.canvas.delete(self.cursor_object)
            self.cursor_object = None
        if at_step < len(self.scan_polygons):
            self.cursor_object =
self.canvas.create_polygon(self.scan_polygons[at_step], fill="blue")

class Landmarks(DrawableObject):
    # In contrast other classes, Landmarks stores the original world
coords and
    # transforms them when drawing.
    def __init__(self, landmarks, canvas, canvas_extents,
world_extents, color = "gray"):
        self.landmarks = landmarks
        self.canvas = canvas
        self.canvas_extents = canvas_extents
        self.world_extents = world_extents
        self.color = color

    def background_draw(self):
        for l in self.landmarks:
            if l[0] =='C':
                x, y = l[1:3]
                ll = to_world_canvas3((x - l[3], y - l[3]),
self.canvas_extents, self.world_extents)
                ur = to_world_canvas3((x + l[3], y + l[3]),
self.canvas_extents, self.world_extents)
                self.canvas.create_oval(ll[0], ll[1], ur[0], ur[1],
fill=self.color)

    def draw(self, at_step):
        # Landmarks are background only.
        pass

class Points(DrawableObject):
    # Points, optionally with error ellipses.
    def __init__(self, points, canvas, color = "red", radius = 5,
ellipses = [], ellipse_factor = 1.0):
        self.points = points
        self.canvas = canvas
        self.color = color
        self.radius = radius
        self.ellipses = ellipses
        self.ellipse_factor = ellipse_factor
        self.cursor_objects = []

    def background_draw(self):
        pass

    def draw(self, at_step):
        if self.cursor_objects:
            map(self.canvas.delete, self.cursor_objects)
            self.cursor_objects = []
        if at_step < len(self.points):
            for i in xrange(len(self.points[at_step])):
                # Draw point.
```

```
                c = self.points[at_step][i]
                self.cursor_objects.append(self.canvas.create_oval(
                    c[0]-self.radius, c[1]-self.radius,
                    c[0]+self.radius, c[1]+self.radius,
                    fill=self.color))
                # Draw error ellipse if present.
                if at_step < len(self.ellipses) and i <
len(self.ellipses[at_step]):
                    e = self.ellipses[at_step][i]
                    points = self.get_ellipse_points(c, e[0], e[1] *
self.ellipse_factor,
                                                     e[2] *
self.ellipse_factor)

self.cursor_objects.append(self.canvas.create_line(
                        *points, fill=self.color))

# Particles are like points but add a direction vector.
class Particles(DrawableObject):
    def __init__(self, particles, canvas, color = "red", radius =
1.0,
                 vector = 8.0):
        self.particles = particles
        self.canvas = canvas
        self.color = color
        self.radius = radius
        self.vector = vector
        self.cursor_objects = []

    def background_draw(self):
        pass

    def draw(self, at_step):
        if self.cursor_objects:
            map(self.canvas.delete, self.cursor_objects)
            self.cursor_objects = []
        if at_step < len(self.particles):
            for c in self.particles[at_step]:
                self.cursor_objects.append(self.canvas.create_oval(
                    c[0]-self.radius, c[1]-self.radius,
                    c[0]+self.radius, c[1]+self.radius,
                    fill=self.color, outline=self.color))
                self.cursor_objects.append(self.canvas.create_line(
                    c[0], c[1],
                    c[0] + cos(c[2]) * self.vector,
                    c[1] - sin(c[2]) * self.vector,
                    fill = self.color))

# World canvas is x right, y up, and scaling according to
canvas/world extents.
def to_world_canvas2(world_point, canvas_extents, world_extents):
    """Transforms a point from world coord system to world canvas
coord system."""
    x = int(world_point[0]*1.25)
    y = int(world_point[1]*1.24)
    return (x, y)

def to_world_canvas3(world_point, canvas_extents, world_extents):
    """Transforms a point from world coord system to world canvas
coord system."""
    x = int(world_point[0] / world_extents[0] * canvas_extents[0])
```

```
    y = int(world_point[1] / world_extents[1] * canvas_extents[1])
    return (x, y)

def to_world_canvas(world_point, canvas_extents, world_extents):
    """Transforms a point from world coord system to world canvas
coord system."""
    x = int(world_point[0] / world_extents[0] * canvas_extents[0])
    y = int(canvas_extents[1] - 1 - world_point[1] /
world_extents[1] * canvas_extents[1])
    return (x, y)

# Sensor canvas is "in driving direction", with x up, y left, (0,0)
in the center
# and scaling according to canvas_extents and max_scanner_range.
def to_sensor_canvas(sensor_point, canvas_extents, scanner_range):
    """Transforms a point from sensor coordinates to sensor canvas
coord system."""
    scale = canvas_extents[0] / 2.0 / scanner_range
    x = int(canvas_extents[0] / 2.0 - sensor_point[1] * scale)
    y = int(canvas_extents[1] / 2.0 - 1 - sensor_point[0] * scale)
    return (x, y)

def slider_moved(index):
    """Callback for moving the scale slider."""
    i = int(index)
    # Call all draw objects.
    for d in draw_objects:
        d.draw(i)

    # Print info about current point.
    info.config(text=logfile.info(i))

def add_file():
    filename = tkFileDialog.askopenfilename(filetypes = [("all
files", ".*"), ("txt files", ".txt")])
    if filename:
        # If the file is in the list already, remove it (so it will
be appended
        # at the end).
        if filename in all_file_names:
            all_file_names.remove(filename)
        all_file_names.append(filename)
        load_data()

def load_data():
    global canvas_extents, sensor_canvas_extents, world_extents,
max_scanner_range
    for filename in all_file_names:
        logfile.read(filename)

    global draw_objects
    draw_objects = []
    scale.configure(to=logfile.size()-1)

    # Insert: landmarks.
    draw_objects.append(Landmarks(logfile.landmarks, world_canvas,
canvas_extents, world_extents))

    # Insert: reference trajectory.
    positions = [to_world_canvas2(pos, canvas_extents,
world_extents) for pos in logfile.reference_positions]
```

```
    draw_objects.append(Trajectory(positions, world_canvas,
world_extents, canvas_extents,
        cursor_color="red", background_color="#FFB4B4"))

    # Insert: scanner data.
    draw_objects.append(ScannerData(logfile.scan_data,
sensor_canvas,
        sensor_canvas_extents, max_scanner_range))

    # Insert: detected cylinders, in scanner coord system.
    if logfile.detected_cylinders:
        positions = [[to_sensor_canvas(pos, sensor_canvas_extents,
max_scanner_range)
                     for pos in cylinders_one_scan ]
                     for cylinders_one_scan in
logfile.detected_cylinders ]
        draw_objects.append(Points(positions, sensor_canvas,
"#88FF88"))

    # Insert: world objects, cylinders and corresponding world
objects, ellipses.
    if logfile.world_cylinders:
        positions = [[to_world_canvas(pos, canvas_extents,
world_extents)
                      for pos in cylinders_one_scan]
                      for cylinders_one_scan in
logfile.world_cylinders]
        # Also setup cylinders if present.
        # Note this assumes correct aspect ratio.
        factor = canvas_extents[0] / world_extents[0]
        draw_objects.append(Points(positions, world_canvas,
"#DC23C5",
                                   ellipses =
logfile.world_ellipses,
                                   ellipse_factor = factor))

    # Insert: detected cylinders, transformed into world coord
system.
    if logfile.detected_cylinders and logfile.filtered_positions and
\
        len(logfile.filtered_positions[0]) > 2:
        positions = []
        for i in xrange(min(len(logfile.detected_cylinders),
len(logfile.filtered_positions))):
            this_pose_positions = []
            pos = logfile.filtered_positions[i]
            dx = cos(pos[2])
            dy = sin(pos[2])
            for pole in logfile.detected_cylinders[i]:
                x = pole[0] * dx - pole[1] * dy + pos[0]
                y = pole[0] * dy + pole[1] * dx + pos[1]
                p = to_world_canvas((x,y), canvas_extents,
world_extents)
                this_pose_positions.append(p)
            positions.append(this_pose_positions)
        draw_objects.append(Points(positions, world_canvas,
"#88FF88"))

    # Insert: particles.
    if logfile.particles:
        positions = [
```

```
            [(to_world_canvas(pos, canvas_extents, world_extents) +
(pos[2],))
             for pos in particles_one_scan]
             for particles_one_scan in logfile.particles]
        draw_objects.append(Particles(positions, world_canvas,
"#80E080"))

    # Insert: filtered trajectory.
    if logfile.filtered_positions:
        if len(logfile.filtered_positions[0]) > 2:
            positions = [tuple(list(to_world_canvas(pos,
canvas_extents, world_extents)) + [pos[2]]) for pos in
logfile.filtered_positions]
        else:
            positions = [to_world_canvas(pos, canvas_extents,
world_extents) for pos in logfile.filtered_positions]
        # If there is error ellipses, insert them as well.
        draw_objects.append(Trajectory(positions, world_canvas,
world_extents, canvas_extents,
            standard_deviations = logfile.filtered_stddev,
            cursor_color="blue", background_color="lightblue",
            position_stddev_color = "#8080ff",
theta_stddev_color="#c0c0ff"))

    # Start new canvas and do all background drawing.
    world_canvas.delete(ALL)
    sensor_canvas.delete(ALL)
    for d in draw_objects:
        d.background_draw()


# Main program.
if __name__ == '__main__':

    # Construct logfile (will be read in load_data()).
    logfile = LegoLogfile()

    # Setup GUI stuff.
    root = Tk()
    frame1 = Frame(root)
    frame1.pack()
    world_canvas = Canvas(frame1,width=800,height=600,bg="white")
    world_canvas.pack(side=LEFT)
    sensor_canvas = Canvas(frame1,width=400,height=600,bg="white")
    sensor_canvas.pack(side=RIGHT)
    scale = Scale(root, orient=HORIZONTAL, command = slider_moved)
    scale.pack(fill=X)
    info = Label(root)
    info.pack()
    frame2 = Frame(root)
    frame2.pack()
    load = Button(frame2,text="Load (additional)
logfile",command=add_file)
    load.pack(side=LEFT)
    reload_all = Button(frame2,text="Reload all",command=load_data)
    reload_all.pack(side=RIGHT)

    # The list of objects to draw.
    draw_objects = []

    # Ask for file.
```

```
all_file_names = []
add_file()
root.mainloop()
root.destroy()
```

Appendix D

lego_robot.py

```
# Python routines useful for handling ikg's LEGO robot data.
# Author: Claus Brenner, 28.10.2012
from math import sin, cos, pi

# In previous versions, the S record included the number of scan
points.
# If so, set this to true.
s_record_has_count = True

# Class holding log data of our Lego robot.
# The logfile understands the following records:
# P reference position (of the robot)
# S scan data
# I indices of poles in the scan data (determined by an external
algorithm)
# M motor (ticks from the odometer) data
# F filtered data (robot position, or position and heading angle)
# E (error) standard deviation in position, or position and heading
angle
# L landmark (reference landmark, fixed)
# D detected landmark, in the scanner's coordinate system. C is
cylinders
# W something to draw in the world coordinate system.
#   C is cylinders, E is 2D error ellipses
# PA list of particles (x, y, heading).
#
class LegoLogfile(object):
    def __init__(self):
        self.reference_positions = []
        self.scan_data = []
        self.pole_indices = []
        self.motor_ticks = []
        self.filtered_positions = []
        self.filtered_stddev = []
        self.landmarks = []
        self.detected_cylinders = []
        self.world_cylinders = []
        self.world_ellipses = []
        self.particles = []
        self.last_ticks = None

    def read(self, filename):
        """Reads log data from file. Calling this multiple times
with different
           files will result in a merge of the data, i.e. if one
file contains
           M and S data, and the other contains M and P data, then
LegoLogfile
           will contain S from the first file and M and P from the
second file."""
        # If information is read in repeatedly, replace the lists
instead of appending,
        # but only replace those lists that are present in the data.
        first_reference_positions = True
        first_scan_data = True
        first_pole_indices = True
```

```
        first_motor_ticks = True
        first_filtered_positions = True
        first_filtered_stddev = True
        first_landmarks = True
        first_detected_cylinders = True
        first_world_cylinders = True
        first_world_ellipses = True
        first_particles = True
        f = open(filename)
        for l in f:
            sp = l.split()
            # P is the reference position.
            # File format: P timestamp[in ms] x[in mm] y[in mm]
            # Stored: A list of tuples [(x, y), ...] in
reference_positions.
            if sp[0] == 'P':
                if first_reference_positions:
                    self.reference_positions = []
                    first_reference_positions = False
                self.reference_positions.append( (int(sp[1]),
int(sp[2])) )

            # S is the scan data.
            # File format:
            #  S timestamp[in ms] distances[in mm] ...
            # Or, in previous versions (set s_record_has_count to
True):
            #  S timestamp[in ms] count distances[in mm] ...
            # Stored: A list of tuples [ [(scan1_distance,... ),
(scan2_distance,...) ]
            #   containing all scans, in scan_data.
            elif sp[0] == 'S':
                if first_scan_data:
                    self.scan_data = []
                    first_scan_data = False
                if s_record_has_count:
                    self.scan_data.append(tuple(map(int, sp[1:])))
                else:
                    self.scan_data.append(tuple(map(int, sp[1:])))

            # I is indices of poles in the scan.
            # The indices are given in scan order
(counterclockwise).
            # -1 means that the pole could not be clearly detected.
            # File format: I timestamp[in ms] index ...
            # Stored: A list of tuples of indices (including empty
tuples):
            #  [(scan1_pole1, scan1_pole2,...),
(scan2_pole1,...)...]
            elif sp[0] == 'I':
                if first_pole_indices:
                    self.pole_indices = []
                    first_pole_indices = False
                self.pole_indices.append(tuple(map(int, sp[2:])))

            # M is the motor data.
            # File format: M timestamp[in ms] pos[in ticks]
tachoCount[in ticks] acceleration[deg/s^2] rotationSpeed[deg/s] ...
            #   (4 values each for: left motor, right motor, and
third motor (not used)).
```

```
            # Stored: A list of tuples [ (inc-left, inc-right), ...
] with tick increments, in motor_ticks.
            # Note that the file contains absolute ticks, but
motor_ticks contains the increments (differences).
            elif sp[0] == 'M':
                ticks = (int(sp[2]), int(sp[1]))
                if first_motor_ticks:
                    self.motor_ticks = []
                    first_motor_ticks = False
                    self.last_ticks = ticks
                self.motor_ticks.append(
                    tuple([ticks[i]-self.last_ticks[i] for i in
range(2)]))
                self.last_ticks = ticks

            # F is filtered trajectory. No time stamp is used.
            # File format: F x[in mm] y[in mm]
            # OR:          F x[in mm] y[in mm] heading[in radians]
            # Stored: A list of tuples, each tuple is (x y) or (x y
heading)
            elif sp[0] == 'F':
                if first_filtered_positions:
                    self.filtered_positions = []
                    first_filtered_positions = False
                self.filtered_positions.append( tuple( map(float,
sp[1:])) )

            # E is error of filtered trajectory. No time stamp is
used.
            # File format: E (angle of main axis)[in radians] std-
dev1 std-dev2
            # OR:          The same format but with std-dev-heading
appended.
            # Note: std-dev1 is along the main axis, std-dev2 is
along the
            # second axis, which is orthogonal to the main axis.
            # Stored: A list of tuples, each tuple is
            # (angle, std-dev1, std-dev2) or
            # (angle, std-dev1, std-dev2, std-dev-heading).
            elif sp[0] == 'E':
                if first_filtered_stddev:
                    self.filtered_stddev = []
                    first_filtered_stddev = False
                self.filtered_stddev.append( tuple( map(float,
sp[1:])) )

            # L is landmark. This is actually background
information, independent
            # of time.
            # File format: L <type> info...
            # Supported types:
            # Cylinder: L C x y diameter.
            # Stored: List of (<type> info) tuples.
            elif sp[0] == 'L':
                if first_landmarks:
                    self.landmarks = []
                    first_landmarks = False
                if sp[1] == 'C':
                    self.landmarks.append( tuple(['C'] + map(float,
sp[2:])) )
```

```
            # D is detected landmarks (in each scan).
            # File format: D <type> info...
            # Supported types:
            # Cylinder: D C x y x y ...
            #   Stored: List of lists of (x, y) tuples of the
cylinder positions,
            #   one list per scan.
            elif sp[0] == 'D':
                if sp[1] == 'C':
                    if first_detected_cylinders:
                        self.detected_cylinders = []
                        first_detected_cylinders = False
                    cyl = map(float, sp[2:])
                    self.detected_cylinders.append([(cyl[2*i],
cyl[2*i+1]) for i in range(len(cyl)/2)])

            # W is information to be plotted in the world (in each
scan).
            # File format: W <type> info...
            # Supported types:
            # Cylinder: W C x y x y ...
            #   Stored: List of lists of (x, y) tuples of the
cylinder positions,
            #   one list per scan.
            # Error ellipses: W E angle axis1 axis, angle axis1
axis2 ...
            #   where angle is the ellipse's orientations and axis1
and axis2 are the lenghts
            #   of the two half axes.
            #   Stored: List of lists of (angle, axis1, axis2)
tuples.
            #   Note the ellipses can be used only in combination
with "W C", which will
            #   define the center point of the ellipse.
            elif sp[0] == 'W':
                if sp[1] == 'C':
                    if first_world_cylinders:
                        self.world_cylinders = []
                        first_world_cylinders = False
                    cyl = map(float, sp[2:])
                    self.world_cylinders.append([(cyl[2*i],
cyl[2*i+1]) for i in range(len(cyl)/2)])
                elif sp[1] == 'E':
                    if first_world_ellipses:
                        self.world_ellipses = []
                        first_world_ellipses = False
                    ell = map(float, sp[2:])
                    self.world_ellipses.append([(ell[3*i],
ell[3*i+1], ell[3*i+2]) for i in xrange(len(ell)/3)])

            # PA is particles.
            # File format:
            #  PA x0, y0, heading0, x1, y1, heading1, ...
            # Stored: A list of lists of tuples:
            #  [[(x0, y0, heading0), (x1, y1, heading1),...],
            #   [(x0, y0, heading0), (x1, y1, heading1),...], ...]
            # where each list contains all particles of one time
step.
            elif sp[0] == 'PA':
                if first_particles:
                    self.particles = []
```

```
                first_particles = False
            i = 1
            particle_list = []
            while i < len(sp):
                particle_list.append(tuple(map(float,
sp[i:i+3])))
                i += 3
            self.particles.append(particle_list)

        f.close()

    def size(self):
        """Return the number of entries. Take the max, since some
lists may be empty."""
        return max(len(self.reference_positions),
len(self.scan_data),
                   len(self.pole_indices), len(self.motor_ticks),
                   len(self.filtered_positions),
len(self.filtered_stddev),
                   len(self.detected_cylinders),
len(self.world_cylinders),
                   len(self.particles))

    @staticmethod
    def beam_index_to_angle(i, mounting_angle = -
0.06981317007977318):
        """Convert a beam index to an angle, in radians."""
        return (i - 341.0) * 0.006765923151543 + mounting_angle

    @staticmethod
    def scanner_to_world(pose, point):
        """Given a robot pose (rx, ry, heading) and a point (x, y)
in the
           scanner's coordinate system, return the point's
coordinates in the
           world coordinate system."""
        dx = cos(pose[2])
        dy = sin(pose[2])
        x, y = point
        return (x * dx - y * dy + pose[0], x * dy + y * dx +
pose[1])

    def info(self, i):
        """Prints reference pos, number of scan points, and motor
ticks."""
        s = ""
        if i < len(self.reference_positions):
            s += " | ref-pos: %4d %4d" % self.reference_positions[i]

        if i < len(self.scan_data):
            s += " | scan-points: %d" % len(self.scan_data[i])

        if i < len(self.pole_indices):
            indices = self.pole_indices[i]
            if indices:
                s += " | pole-indices:"
                for idx in indices:
                    s += " %d" % idx
            else:
                s += " | (no pole indices)"
```

```
if i < len(self.motor_ticks):
    s += " | motor: %d %d" % self.motor_ticks[i]

if i < len(self.filtered_positions):
    f = self.filtered_positions[i]
    s += " | filtered-pos:"
    for j in (0,1):
        s += " %.1f" % f[j]
    if len(f) > 2:
        s += " %.1f" % (f[2] / pi * 180.)

if i < len(self.filtered_stddev):
    stddev = self.filtered_stddev[i]
    s += " | stddev:"
    # Print stddev in both axes, and theta, if present.
    # Don't print the orientation angle stddev[0].
    for j in (1,2):
        s += " %.1f" % stddev[j]
    if len(stddev) > 3:
        s += " %.1f" % (stddev[3] / pi * 180.)

return s
```

Appendix E

track.cpp

```
//g++ track.cpp -o track -lopencv_core -lopencv_imgproc -
lopencv_objdetect -lopencv_highgui


#include <iostream>
#include <stdio.h>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"

using namespace cv;
using namespace std;

 int main( int argc, char** argv )
 {

FILE *f = fopen("reference.txt","w");
if (f == NULL)
{
printf("Error opening file\n");
exit(1);
}

VideoCapture cap(0); //capture the video from webcam
//cap.open(0);
    if ( !cap.isOpened() )  // if not success, exit program
    {
         cout << "Cannot open the web cam" << endl;
         return -1;
    }

    namedWindow("Control", CV_WINDOW_AUTOSIZE); //create a window
called "Control"

int iLowH = 0;
int iHighH = 179;

int iLowS = 150;
int iHighS = 255;

int iLowV = 0;
int iHighV = 255;

//Create trackbars in "Control" window
createTrackbar("LowH", "Control", &iLowH, 179); //Hue (0 - 179)
createTrackbar("HighH", "Control", &iHighH, 179);

createTrackbar("LowS", "Control", &iLowS, 255); //Saturation (0 -
255)
createTrackbar("HighS", "Control", &iHighS, 255);

createTrackbar("LowV", "Control", &iLowV, 255);//Value (0 - 255)
createTrackbar("HighV", "Control", &iHighV, 255);

int iLastX = -1;
int iLastY = -1;
```

```
//Capture a temporary image from the camera
Mat imgTmp;
cap.read(imgTmp);
imgTmp = imgTmp(Rect(157,1,482,479));
//cap>>imgTmp;

//Create a black image with the size as the camera output
Mat imgLines = Mat::zeros( imgTmp.size(), CV_8UC3 );;

    while (true)
    {
        Mat imgOriginal;

        bool bSuccess = cap.read(imgOriginal); // read a new frame
from video
    imgOriginal = imgOriginal(Rect(157,1,482,479));


         if (!bSuccess) //if not success, break loop
        {
             cout << "Cannot read a frame from video stream" <<
endl;
             break;
        }

Mat imgHSV;

cvtColor(imgOriginal, imgHSV, COLOR_BGR2HSV); //Convert the captured
frame from BGR to HSV

Mat imgThresholded;

inRange(imgHSV, Scalar(iLowH, iLowS, iLowV), Scalar(iHighH, iHighS,
iHighV), imgThresholded); //Threshold the image

//morphological opening (removes small objects from the foreground)
erode(imgThresholded, imgThresholded,
getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );
dilate( imgThresholded, imgThresholded,
getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );

//morphological closing (removes small holes from the foreground)
dilate( imgThresholded, imgThresholded,
getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );
erode(imgThresholded, imgThresholded,
getStructuringElement(MORPH_ELLIPSE, Size(5, 5)) );

//Calculate the moments of the thresholded image
Moments oMoments = moments(imgThresholded);

double dM01 = oMoments.m01;
double dM10 = oMoments.m10;
double dArea = oMoments.m00;

// if the area <= 10000, I consider that the there are no object in
the image and it's because of the noise, the area is not zero
if (dArea > 10000)
{
//calculate the position of the ball
int posX = dM10 / dArea;
int posY = dM01 / dArea;
```

```
fflush(f);
fprintf(f,"P\t%d\t%d\n",posX,posY);
printf("P\t%d\t%d\n",posX,posY);
if (iLastX >= 0 && iLastY >= 0 && posX >= 0 && posY >= 0)
{
//Draw a red line from the previous point to the current point
line(imgLines, Point(posX, posY), Point(iLastX, iLastY),
Scalar(0,0,255), 2);
}

iLastX = posX;
iLastY = posY;
}
imshow("Thresholded Image", imgThresholded); //show the thresholded
image

imgOriginal = imgOriginal + imgLines;
imshow("Original", imgOriginal); //show the original image

        if (waitKey(30) == 27) //wait for 'esc' key press for 30ms.
If 'esc' key is pressed, break loop
        {
            cout << "esc key is pressed by user" << endl;
            break;
        }

    }
fclose(f);
   return 0;
}
```

Appendix F

motor_Control.ino

```
#include <Hercules.h>
#include "Timer.h"
int count_new;
int count_old = 1;
int counter = 0;
int distance1=0;
Timer t;
Timer t2;
int count_new2;
int count_old2 = 1;
int counter2 = 0;
int distance2=0;

int distance=0;
int incomingByte=0;

void forward()
{
  //Serial.println("Forward");
  MOTOR.setSpeedDir1(20, DIRR);
  MOTOR.setSpeedDir2(20, DIRF);
}

void right()
{
  //Serial.println("Right");
  MOTOR.setSpeedDir1(20, DIRR);
  MOTOR.setSpeedDir2(0, DIRF);
}

void left()
{
 // Serial.println("Left");
  MOTOR.setSpeedDir1(0, DIRR);
  MOTOR.setSpeedDir2(20, DIRF);
}

void sstop()
{
  //Serial.println("Stop");
  MOTOR.setSpeedDir1(0, DIRR);
  MOTOR.setSpeedDir2(0, DIRF);
}
void setup()
{
    MOTOR.begin();
    Serial.begin(9600);
    t.every(10,encoder);
   // t2.every(100,dist);
}

void loop()
{
    t.update();
    t2.update();
    if (Serial.available() > 0) {
    incomingByte = Serial.read();
```

```
    if (incomingByte == 119)
    {
      forward();
    }
    if (incomingByte == 100)
    {
      left();
    }
    if (incomingByte == 97)
    {
      right();
    }
    if (incomingByte == 32)
    {
      sstop();
    }
    if (incomingByte == 113)
    {
      counter=0;
      counter2=0;
      count_old=1;
      count_old2=1;
    }
}
}


void encoder()
{
 count_new =  digitalRead(2);
 count_new2 =  digitalRead(3);
 if(count_new-count_old != 0)
 {
   counter=counter+1;
 }

 if(count_new2-count_old2 != 0)
 {
   counter2=counter2+1;
 }
 count_old2=count_new2;
 count_old=count_new;
      Serial.print("a");
     Serial.println(counter);
     Serial.print("b");
    Serial.println(counter2);

}
void dist()
{
     distance1 = counter;
    distance2 = counter2;
    //distance = (distance1+distance2)/2;
   // Serial.print("h");
 //   Serial.print(distance1);
 //   Serial.print(distance2);
 //   Serial.println("t");
}
```

Appendix G

calculate_xy.c

```
#include "urg_sensor.h"
#include "urg_utils.h"
#include "open_urg_sensor.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char *argv[])
{
    urg_t urg;
    long *data;
    long max_distance;
    long min_distance;
    long time_stamp;
    int i;
    int n;

    if (open_urg_sensor(&urg, argc, argv) < 0) {
        return 1;
    }

    data = (long *)malloc(urg_max_data_size(&urg) *
sizeof(data[0]));
    if (!data) {
        perror("urg_max_index()");
        return 1;
    }

    urg_start_measurement(&urg, URG_DISTANCE, 1, 0);
    n = urg_get_distance(&urg, data, &time_stamp);
    if (n < 0) {
        printf("urg_get_distance: %s\n", urg_error(&urg));
        urg_close(&urg);
        return 1;
    }


    urg_distance_min_max(&urg, &min_distance, &max_distance);
    for (i = 0; i < n; ++i) {
        long distance = data[i];
        double radian;
        long x;
        long y;

        if ((distance < min_distance) || (distance > max_distance))
{
            continue;
        }

        radian = urg_index2rad(&urg, i);
        x = (long)(distance * cos(radian));
        y = (long)(distance * sin(radian));

        printf("%ld, %ld\n", x, y);
```

```
    }
    printf("\n");


    free(data);
    urg_close(&urg);

#if defined(URG_MSC)
    getchar();
#endif
    return 0;
}
```