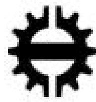

TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ANDREAS UWAOMA

ON PARTICLE FILTER LOCALIZATION AND MAPPING FOR NAO ROBOT

Master of Science thesis

Examiner: prof. Risto Ritala
Examiner and topic approved by the
Faculty Council of the Faculty of
Engineering Sciences
on 6th May 2015

ABSTRACT

ANDREAS UWAOMA: On particle filter localization and mapping for Nao robot.

Tampere University of Technology
Master of Science Thesis, 58 pages, 20 Appendix pages
June 2015
Master's Degree Program in Machine Automation
Major: Mechatronics and Micromachines
Examiner: Professor Risto Ritala

Keywords: Humanoid robot, Nao, Localization, Particle filters, Path planning.

The performance of autonomous mobile robots within an indoor environment relies on an effective detection and localization system. Self-Localization within an indoor environment has been studied and tested experimentally on humanoid robot Nao. The solution utilizes a pre-existing map with known and unknown features.

The aim of this thesis is to utilize map of visual features and the Monte-Carlo Scheme (particle filters) in localization and navigation. Nao robot cameras has been used for detection Naomarks, the detection of these features provides an estimation of the relative distances of features to current robot position. These measurements are applied to a visual localization algorithm that uses a pair of known feature to localize the robot, furthermore the measurements is fused to a particle filter algorithm for estimating the pose of the robot within the map. The particle filter implementation was based on the C++ programming language. A simple path planning scheme was implemented for continuous localization while navigating a paths with obstacles.

The algorithms has been tested with reference to measurements provided by an external sensor. The results of the implementations indicates that the robot can effectively navigate from a start position to a predefined location while avoiding obstacles on its path.

PREFACE

This thesis work has been realized at the department of Automation Science and Engineering with the help of a number of people, whom I would like to thank.

First, special thanks to my supervisor, Professor Risto Ritala for providing the resources and the much needed guidance through this thesis.

I also like to thank Mikko Lauri, Joonas Melin and everyone who provided technical support and suggestions through the implementation of this work.

Lastly, express my gratitude to my family for their love and encouragement through the challenging times of my studies.

Tampere, 30.06.2015

Uwaoma Andreas E.

CONTENTS

1. INTRODUCTION 1
1.1 Objectives 1
1.2 Contribution 2
1.3 Thesis structure 2
2. REVIEW OF MOBILE ROBOTICS 4
2.1 Autonomous mobile robots 4
2.2 Autonomous navigation 5
2.3 Path planning and obstacle avoidance 6
2.4 Localization 8
2.4.1 Particle filter based localization 8
2.4.2 Kalman filter based localization 10
2.5 SLAM implementation on Nao robot 11
2.5.1 Visual SLAM 12
2.5.2 Monocular SLAM 12
3. THE ROBOTIC PLATFORM – NAO 14
3.1 The Nao robot 14
3.2 Nao sensors 15
3.3 Programming 18
3.3.1 Naoqi framework 18
3.3.2 Device communication manager (DCM) 21
3.3.3 Nao Simulation Environment 21
3.4 Motion 23
3.5 Vision 23
4. METHODODOLOGY AND IMPLEMENTATION
2.1 Measurements 25
4.1.1. Distance measurements 25
4.1.2. Turn measurements 26
4.1.3. Straightness measurements 27
2.2 Visual markers 28
4.2.1. Naomarks 28
4.2.2. Limitations of the landmark module 29

4.2.3. Landmark detection info 30
4.2.4. Marker coordinates 31
2.3 Environment map 33
2.4 Visual localization 33
2.5 Implementation of particle filter based localization 38
4.5.1. Motion model 38
4.5.2. Measurement and updates 40
4.5.3. Resampling 41
2.6 Motion planning 41
5. EXPERIMENTS AND RESULTS 45
5.1 Initial tests 45
5.2 Particle filter localization 46
5.3 Localization while robot moves 52
6. CONCLUSSION 58

LIST OF FIGURES

Figure 2.1. Robo-tuna biomimetic robot....5
Figure 2.3. Pose data comparison; augmented MCL, top camera localization and odometry data....9
Figure 2.2. A typical Kalman filter application....10

Figure 3.1. The Nao robot....14
Figure 3.2. Nao sensors and joints....15
Figure 3.3. Head tactile sensors: A: front, B: middle, and C: rear sensors....16
Figure 3.4. Nao's cameras field of view....17
Figure 3.5. Nao Software Interaction....19
Figure 3.6. Naoqi Framework....20
Figure 3.7. Screenshot of the Choregraphe software....24

Figure 4.1. Walk path of Nao robot for 200cm command along x-axis....28
Figure 4.2. Naomarks samples....29
Figure 4.3. Naomark dimension 31
Figure 4.4. Nao Camera angles 32
Figure 4.5. Nao Frame and global frame 34
Figure 4.6. Representation of marker locations in global and robot frame 35
Figure 4.7. Illustration of robot and with a specified target coordinate 42
Figure 4.8. Flow-chart of the navigation and localization program 44
Figure 4.9. Class Structure of the implementation 45

Figure 5.1. Distance and lateral displacement for 200 cm straight walk 46
Figure 5.2. Distance and lateral displacement for 100 cm straight walk 47
Figure 5.3. Visual localization result 48
Figure 5.4. Particle pose estimate after move [0.8, 30^0] 50
Figure 5.5. Particle pose estimate for target coordinates [1.3, 1.0] 52
Figure 5.6. Particle pose estimate for target coordinates [0.5, 0.6]....53
Figure 5.7. Particle pose estimate for (a) an obstacle at [0.6, 0.6]....54
Figure 5.8. Estimate of robot position (b) obstacle at [0.6 0.6] evaded 55
Figure 5.9. Estimate of robot position at (c) final target coordinates [1.3, 1.2]....56
Figure 5.10. Estimate of robot path to target....57

LIST OF TABLES

Table 4.1. Walk measurements. 26

Table 4.2. Robot turn measurements for angles 180, 90 and 45 degrees. 27

Table 4.3. Global coordinate of markers on map. 33

Table 5.1. Localization experiment using known start points, distance and direction Command. 49

Table 5.2. Localization experiment using known start points and target coordinates..51

Table 5.3. Results of autonomous localization with obstacles. 56

1 INTRODUCTION

The Intelligent Sensing Laboratory of Automation Science and Engineering (ASE) aims at developing methods for autonomous mobile robots to interact and navigate within a specified environment. The long-term goal of the research in intelligent sensing is to develop processes whereby the robots can focus their attention so that they optimally perform task such as autonomous navigation, obstacle detection and avoidance and social interaction. Over the years, research and development of mobile robots has produced promising results with the advent of Honda's ASIMO and Sony's AIBO. Recently a new platform Nao developed by Aldebaran robotics has become very popular amongst researchers. Nao platform has gained a lot of interest due to its relatively low-cost, array of available sensors and relative ease of programming.

A common task for a mobile robot is to follow an object at a specific distance while simultaneously localizing itself in the environment. In order to perform this function, the robot must be able recognize the object and also identify specific features in the environment. Using the detected features the robot should be able to guide itself towards a specified target while avoiding bumping into obstacles on its path.

1.1 Objective

The objective of this thesis is to implement a particle filter based solution for localization and mapping of mobile robot provided by the Nao platform. The work is divided into the following tasks:

- Create map of known environment based on features.
- Implement self-localization on Nao using monocular vision.
- Implement a sequential Monte-Carlo scheme to verify the performance of the localization algorithm.

The self-localization on Nao robot involves definition of the map based on features that can be easily identified by the robot. Applying a set of actions that is required for the robot to reach a set goal point within the map and determining the robot's pose at each state by matching actions and new observations to previous states.

1.2 Contribution

The major contributions to this thesis can be summarized as follows:

- A sequential Monte-Carlo Scheme (SMC) is implemented for Nao localization based on C++ programming language.
- A feature based map is developed using Nao markers for an environment with incomplete information about obstacles.
- Visual localization is implemented using information from the map.

The implementation of SMC undertaken in this thesis is a verification of the Nao robot's capabilities in the C++ programming environment. This implementation can be further explored within capabilities of the platform for a more robust scheme for tasks such as path planning etc.

1.3 Thesis structure

The structure of the thesis is as follows;

Chapter 2 contains an overview of mobile robots, and a brief timeline of developments in mobile robotics. The chapter also discusses the main ideas in mobile robotics. Key concepts, such as navigation, path planning, active sensing, vision and localization are briefly explained. In addition previous works based on biped robots are presented.

Chapter 3 introduces the robotic platform Nao. The hardware and software will be described in detail. The chapter also describes the sensors available on the platform, the mode of operation of the sensors, actuators and the available programming methods. The rest of the chapter will be focused on a thorough examination of modules pertinent to the thesis.

Chapter 4 describes the methods used in the implementation. The SMC, the visual localization and the Nao visual sensing information obtained from features will be covered here. Furthermore, the chapter presents the approach to programming, challenges encountered during implementation, an analysis of the decision taken during the implementation and the factors that necessitated such decisions.

Chapter 5 presents the results of the implementation. The chapter discusses the result of the SMC and visual localization.

A summary of the works completed is presented in Chapter 6. Suggestion on future works based on the experience gained from this thesis and the uses of the platform is presented.

2 REVIEW OF MOBILE ROBOTICS

Operation of an autonomous mobile robot in complex locations such as factory floors, homes or generally within maze-like environment requires a dependable 'self-localization' system. This section will present a general overview of autonomous mobile robots, past development in mobile robotics and an explanation of key concepts such as autonomous navigation, path planning, obstacle avoidance and sensing. The rest of the section will review the earlier works based on the biped robot platforms.

2.1 Autonomous mobile robots

Autonomous mobile robots have become an interesting discourse for several reasons. First, mobile robots are less viewed as mere computers on wheels having ability to detect some physical properties in their environment using inbuilt sensors. A mobile robot is an intelligent agent that combines a host of computers, actuators and an array of sensors whereby it is able to detect features, identify patterns and build a knowledge base of its operational environment. With this knowledge base, the robot is able to navigate, and perform required tasks in the learned environment and adapt the knowledge to environment not previously learned.

In earlier developments in mobile robots, robots were designed to mimic functions of humans and other species. One of such robots is Robotuna developed by David Barret [1] of MIT as part of his doctoral thesis in 1995. The robot is a biomimetic robot that moves through water by swimming like a biological fish. His design has been adopted and further developed by Boston engineering as a 4 foot long undersea vehicle that can blend with marine life and perform both civilian and military missions. Honda's "Prototype model 2" humanoid robot that was first shown in 1996 had the ability to stand like a human. The most progress came with Sony's robotic dog Aibo introduced in 1999 and Honda's Humanoid robot Asimo introduced a year after. Both platforms had the ability to walk, run, communicate with humans and interact with the environment.

Figure 2.1 Robo-tuna developed by Boston engineering [14]

Autonomous robots has since developed into various types of land based robots, underwater robot, and aerial robots. Each of these having characteristics that best suits their operational environment.

2.2 Autonomous navigation

Navigation is “the process of accurately [2] determining position and velocity relative to a known reference”. It is a goal orientated behavior that moves an agent between its present location and the desired location. Autonomous navigation is when the agent exhibits this behavior with little or no human intervention.

Autonomous navigation is one of the most challenging competences required of a mobile robot. For a robot to successfully navigate an environment, the robot must be capable of the following key functions.

- Perception
- Cognition
- Localization
- Motion control.

Perception requires that the robot is able to extract useful data about its environment using an array of sensors. The data extracted is processed by the onboard computers and the robot is able to make decision based on the data.

Cognition requires that the robot can decide on how to achieve its goals. The robot must be able to decide the specific actions from a set of possible actions that will take it from its present state to the goals.

Localization requires that the robot is able to determine its own location with respect to some external reference. The challenge here is to actively combine data from sensors such as cameras and time of flight sensors with the odometry data such that the mobile robot becomes aware of its state in the environment. Identifying its absolute location locally or globally is not enough for localization. Determining its relative position to humans, objects or other robot that might be operating in same space is equally important for the appropriate performance of task.

Motion control requires that the robot can regulate its actuators' output in order to attain the desired motion trajectory.

Amongst the four critical functions described, extensive attention has been devoted to localization and this has produced several new and evolving approaches to how autonomous mobile robots operate. Some of the approaches will be discussed in subsequent sections of this chapter.

2.3 Path planning and obstacle avoidance

Path planning, also referred to as motion planning, involves finding a sequence of actions that transforms an agent from an initial state to the desired goal state. In path planning, the states represent the location of the agents while the actions the agent can take, each having an associated cost attached is the transition. The path is optimal if the sum of associated transition costs across the possible paths leading from the initial location to the goal position is minimized. When planning paths, for example the completeness of the path and the optimality of the path need to be considered.

A path planning algorithm is *complete* if the agent is guaranteed to find a path in a finite time when one exists and will let us know if no path exists. Similarly, the planning algorithm is *optimal* if it is guaranteed to find the optimal path.

In mobile robotics, topological path planning consists of representing the environment of the robot as a graph with nodes and edges. The cost of each edge represents the cost

of transiting between two nodes. Path planning can then be treated as a simple search of sequence of nodes that connects the start node to the desired end node at an admissible cost. A number of methods have been developed for computing least cost path. Two common methods are Dijkstra's algorithm (Dijkstra 1959) and the A*(Hart, Nelson & Rafael 1968).

Obstacle avoidance is a key factor for the successful operation of an autonomous mobile robot. Virtually all mobile robots feature some form of collision avoidance ranging from primitive algorithms to well-developed algorithms that manage detection using sensors and stop the robot short of an obstacle or enables the robot to detour around the obstacle to avoid collision.

Common obstacle avoidance methods include edge detection, occupancy grids and virtual force fields (VFF) [3-4].

In the edge detection method, the avoidance algorithm attempts to determine the position of vertical edges and consequently steers around the detected edge. The lines connecting detected edges are considered as obstacle boundaries. The edge detection method though popular, has several limitations. The efficacy of this method depends on the sensitivity and accuracy of the sensors. In the case of sonar sensors which are commonly used in mobile robotics, there are many shortcomings, some of which are explained as follows.

The poor directionality limits the ability to detect the spatial position of the obstacle in ranges between 10cm - 40cm.

Specular reflections which occur when the angle of incidence between the wave front and a smooth surface is large, results in the surface reflecting the incoming ultrasonic beam away from the sensors. Furthermore, Ultrasonic noise form external sources often cause the robot to detect non exiting edges.

In the occupancy grid method, the robot's work space is divided into small square cells of fixed sizes to form a grid. Each cell is assigned a certainty values that indicates the measure of confidence that an obstacle exists within that cell. The greater the certainty values the more likely the cell is occupied by an obstacle. As the robot moves within its work space while sampling continuously for obstacles, a stationary obstacle gives more count of echo readings while the incorrect readings are minimal due to randomness.

The virtual force field (VFF) method relies on the assumption that obstacles conceptually generate some potential field that repulses the robot as the robot approaches the obstacle. The closer the robot is to an obstacle the stronger the repulsive field.

The basic VFF method is a combination of the certainty grid described above with a potential field. While the mobile robot transverses the workspace, range readings are recorded and projected to a certainty grid. The algorithm scans small's cells within the workspace that represents the possible locations of the robot. Each cell applies a repulsive force to the robot.

The magnitude of the repulsive field indicates the measure of certainty in the presence of an obstacle in the proximity of the cell. The magnitude of the certainty is inversely proportional to the square of the distance between the cell and the robot. The VFF method has a clear advantage over the edge detection method because incorrect readings are eliminated since VFF does not utilize sharp edges but responses to a cluster of densities. Furthermore, the grid representation allows the integration of data from different types of sensors for example vision, proximity and contact sensors.

2.4 Localization

2.4.1 Particle filter based localization

Particle filter based localization describes a probabilistic scheme that tracks a robot's belief state using an arbitrary probability density function to represent the robot's position. This scheme approximates a state and its variance by a set of samples – called particles - comprising possible states and weightings representing the probability of each state. The algorithm starts with a uniform random distribution over the configuration space, indicating the lack of information about robot's initial location. Each point or position in the robot's space is equally likely at this stage. When the position of the robot changes by a specified value, each particle is updated through the motion model. The update is performed according to the last control input to reflect the change in position of the robot. Weighting value is computed for each particle by considering the likelihood of data when observing specified landmarks. The particles are resampled with respect to their weights. Thus resampling is carried out such that particles which are consistent with sensor readings are more likely selected. After

several observations the particles converge to reflect a better estimate of the robots' pose.

Predictive estimation of robot's camera position and an implementation of the kinematic model based on the odometry system were proposed by Eshan Hashemi, et al [5]. Their work focused on the application of the Augmented Monte Carlo localization on the landmarks, lines, and points and optimized filtering parameters of robot state estimation. The current set of particles is obtained by an application of the motion model on the previous set according to the last control action issued to the robot. A weighting value is computed for each landmark by considering the observations of landmarks. The final weighting of particles is a numerical product of detect landmark probabilities.

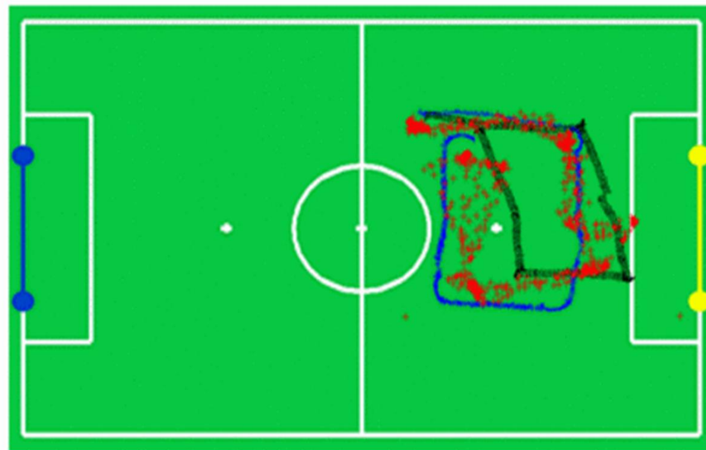

Figure 2.2. *Pose data comparison; augmented MCL (red pluses), top camera localization [5] (blue lines), and odometry data (black lines)*

In their implementation, the robot's head was mounted with a colored outline to enable the tracking of poses and comparison with perceived positions and orientations. This tracking system utilizes an external Wi-Fi camera mounted above the work space. Independent tests were carried out on the robot for three different maneuvers both for simulated and empirical data. The result, see Fig. 2.2 shows ± 10 cm error in position and ± 5 degree error in orientation for the augmented MCL compared with the perceived position and orientation by the external reference camera.

2.4.2 Kalman filter based localization

The Kalman filter (KF) is a mathematical algorithm for estimating the state of a noisy linear dynamic system. The state refers to a vector of variables that describe the system. In the case of a mobile robot, the state vector is [x, y, ϴ], where (x, y) is the coordinate of the location on a plane and theta is the orientation of the robot with respect to a reference. The KF produces an optimal estimate of a system's state based on the knowledge of the system and measuring device, the description of the system noise and measurement errors and the uncertainty in the dynamics of the system.

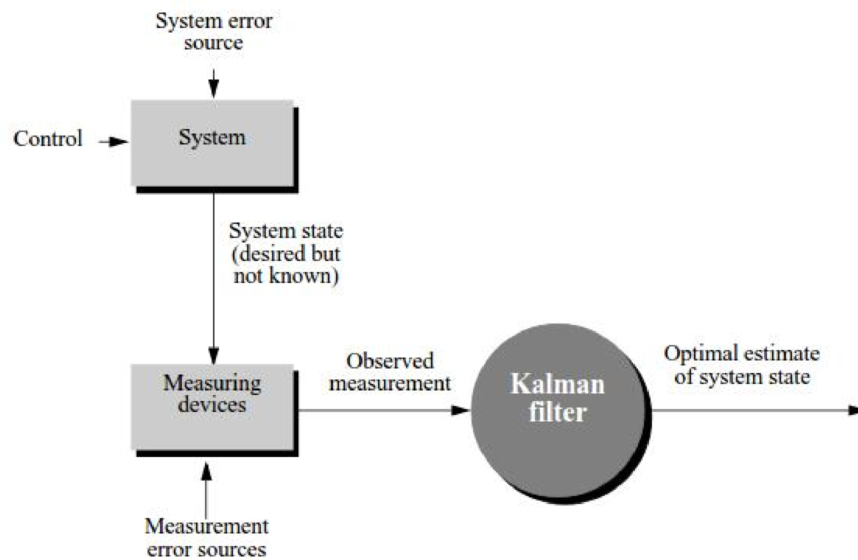

Figure 2.3: A typical Kalman filter application [2]

The Kalman filter assumes that the system is linear, measurement noise and the model noise are independent, and all other noise in the system are white noise and can be modelled with a Gaussian distribution. Mobile robot localization commonly meets all these assumptions except that the trajectory of mobile robot is non-linear. This problem has been solved by a modification of the Kalman filter call extended Kalman filter (EKF). The EKF places the linear trajectory of the KF with an estimated trajectory that models the non-linearity of the system.

The Kalman filter is broken into two steps. The first is the prediction step or time update. At this step the state of the system is predicted based of the corresponding system kinematics. The second step is the correction or measurement update. During the

correction step, the state of the system is updated to reflect the data from sensors measurements.

The prediction step can be written as:

$$\hat{X}_{k|k-1} = A\hat{X}_{k-1|k-1} + Bu_k \quad (1)$$

$$P_{k|k-1} = AP_{k-1|k-1}A^T + Q \quad (2)$$

For equations above, X is the state vector, which is $(x \quad y \quad \theta)^T$ in the case of a mobile robot. k is the time step denoting the time of measurement and estimates. The control vector u_k represents the odometer readings from the robot. A and B are matrices that relate the input vector to the state vector. P is a matrix representing the error covariance and Q is the noise covariance matrix.

The correction step can be written as:

$$J_k = P_{k|k-1}C^T(\,C\,P_{k|k-1}\,C^T + R)^{-1} \quad (3)$$

$$\hat{X}_{k|k} = \hat{X}_{k|k-1} + J_k\left(Y_k - C\hat{X}_{k|k-1}\right) \quad (4)$$

$$P_{k|k} = (I - J_k C)P_{k|k-1} \quad (5)$$

Y represents the measurement data vector, C is a matrix relating the measurement vector to the state vector, R is the noise covariance matrix for the measurement model and J is the Kalman gain.

The Kalman filter, though very powerful, suffers same downsides as other localization algorithms. It depends on information about previous state and this implies that if the initial pose is unknown, the localization would have a large error in its initial guess. This degrades its performance. Furthermore, since each step is dependent on the previous state, small error at each update will propagate leading to state where the robot is unable to recover an accurate pose estimate.

2.5 SLAM implementations on Nao robot

Several attempts have been made to implement simultaneous localization and mapping (SLAM) on Nao robot with varying success. The following sections briefly describe the visual compass and Monocular SLAM. Since the Nao robot cameras do not have an overlapping field of view, it is impossible to utilize the benefits of stereo vision for SLAM implementation.

2.5.1 Visual SLAM

E. Wirbel et al [6] attempted an implementation of visual SLAM on the Nao robot but achieved a visual compass instead. A visual compass provides an improved estimate of the change in robot's orientation when robot is rotated about its z-axis and the new orientation with respect to a reference point, provided the reference is detectable in both images. The approach used was tracking of key points between image frames. Key points are extracted from a panorama of images captured as the robot rotates about the z-axis. This was accomplished using features from accelerated segment test (FAST) [7] and speed-up robust features (SURF) descriptors [8]. Kalman filter approach was not used because of the limited processing power of the robot. Instead a notion of observation lines was introduced. In case where a key point is detected once, the key point will lie on the line. When the key point is detected more than once, its position is given as an average of intersections of observation lines fitting the number of detections. This rule holds provided the observation lines are not collinear.

2.5.2 Monocular SLAM

Simon Fojtu, et al [9] applied structure from motion (SfM), which is a technique for matching key points in a sequence of image frames as the camera moves. First, a map of the 3D environment is built using SfM-Seqv2, with some modifications that permitted iterative update. The modification ensures that the map of camera poses is iteratively built in real-time. Using the map, a world coordinate is defined by the association of at least three 3D points selected from the model to global coordinates. All other points in the cloud are mapped to the global frame with respect to the selected coordinate using similarity transforms.

An estimate of the robot pose is obtained in an image of the mapped scene by applying image processing algorithms. First, SURF features detection is applied to eliminate image distortion. Then feature matching between the image and 3D point cloud is carried out followed by the solution of a 3-point pose problem for a calibrated camera within a random sample consensus [RANSAC] loop [10]. The resulting data is classified as true pose estimate if the number of inliers is above a set threshold.

The odometry data was derived from robot's step length and walk angle. Given a start point, the robot pose after walk or turn action is obtained relative to previous poses.

Fusion of odometry data and visual localization data is performed by application of a weighting to both data sources. Where one of the data sources is deemed unreliable, the system falls back to a single data source as a measure of true pose estimates. The pose estimate is computed as:

$$p_e(k) = W.p_v(k) + (1 - W)(p_e(k - 1) + p_0(k)) \qquad (6)$$

where W is the weight assigned to the data source, p_e p_v and p_0 are the pose estimates at time epoch k, visual localization pose estimate, and Odometry pose estimate respectively. The weighting is biased such that estimates relies more visual localization when data is available. The confidence level of the pose estimate is determined using a Bayes filter according to the following update rule.

$$c(k + 1) = \frac{p.c(k)}{p.c(k) + (1 - p)(1 - c(k))} \qquad (7)$$

Where confidence $c = 1$ if the robot is confident about its pose and $c = 0$ when it is unsure. The parameter p is set to 0.8 when both visual data and odometry is used and 0.2 when odometry only is used.

Their approach was validated by real and simulated data and the results obtained showed the error of determining robot pose from visual odometry to be a normal distribution around the true pose and this error complements the result from robot's odometry.

3 THE ROBOTIC PLATFORM - NAO

This chapter introduces the Nao robot. It describes hardware and the software modules, the inbuilt sensors, the network equipment and the accompanying operating system. The rest of the chapter focuses on the software for programing and a detailed description of the modules that are utilized for performing the tasks in the thesis.

3.1 The Nao Robot

The Nao humanoid robot [11] was developed by Aldebaran robotics, a company based in France. The Nao model H25 V4, is a 58 cm tall robot equipped with an onboard Intel Atom 1.6 GHz CPU. It has 25 degrees of freedom (DOF).

The head of the robot has two degrees of freedom which are the head pitch and yaw. Both arms have four degrees of freedom and both legs five degrees of freedom.

Figure 3.1. *The Nao robot*

There are two joints in the pelvis coupled together and actuated by a single motor, such that the pelvis joints cannot move independent of the other.

Located within the torso is an Inertial Monitoring Unit (IMU) with its own processor. This unit enables the estimation of torso speed and attitude. Communication with Nao is

enabled either by an Ethernet port at the back of Nao's head or through Wi-Fi. The network is IEEE compatible, it uses 802.11g standard and can use both WEP and WPA security protocols. The robot also includes infrared transceivers. These components are installed on the robot's eye and they enable communication between the Nao robot and other robots in its vicinity and other infrared emitters.

NAO is powered by a lithium ion 27.6Wh battery located at the back of its torso. The robot documentation claims that the robot can withstand 60 minutes of active use, and 90 minutes in normal operational mode, but experience has shown the active use being limited to a maximum of 30 minutes.

3.2 Nao Sensors

The Nao Robot is equipped with a variety of sensors that enables information gathering from itself and its immediate environment.

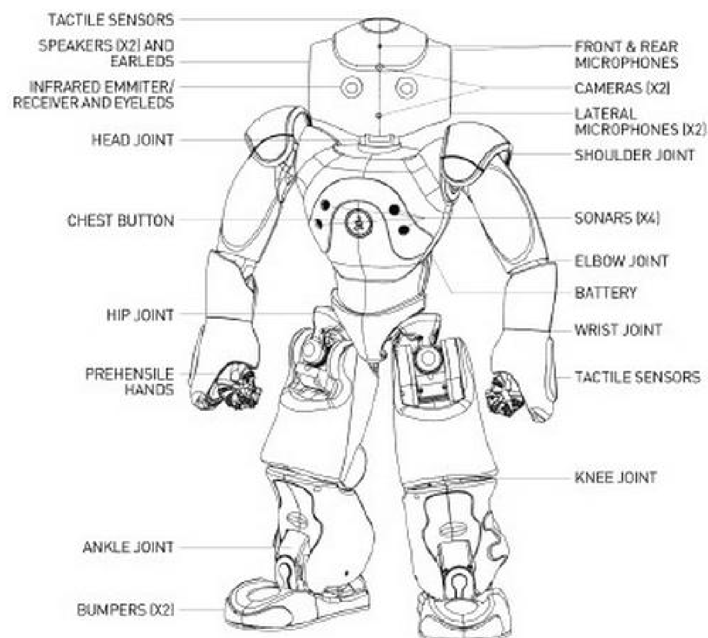

Figure 3.2. Nao sensors and joints

The sensor classified into three types: The proprioceptive, exteroceptive and exproprioceptive sensors.

The *proprioceptive sensors* measure signals originating within the robot. They are responsible for self-maintenance and controlling the internal status of the robot. These

include the IMU, magnetic rotary encoders, battery level sensor, and the joint motor temperature sensors.

The *exteroceptive sensors* are proximity sensors. These sensors determine the measurements of objects relative to a robot's frame of reference. They provide information about the robots environment. Examples of these sensors are cameras, ultrasound sonars, tactile sensor, and infrared sensor.

The *exproprioceptive sensors* use a combination of proprioceptive and exteroceptive monitoring. These sensor measure the difference between an internal state and an external state, for example, the temperature of the robots motors relative to the environment temperature. The Nao includes one exproprioceptive sensor which is the force sensitive resistor (FSR).

The tactile sensors are a set of capacitive sensors positioned on the head, the chest and on the arm of the robot. The first set of tactile sensors is in three sections of the robot's head, i.e. the front, middle and rear of the head. The head tactile sensor provides a programmable touch interface for commanding the robot.

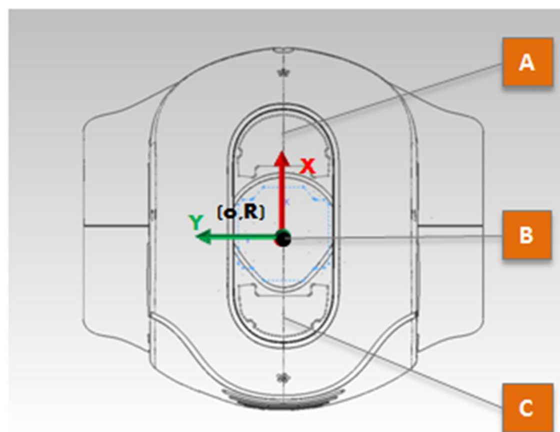

Figure 3.3.Head tactile sensors: A: front, B: middle, and C: rear sensors

The chest tactile sensor is used as the power on/off button. The button is also used in disabling stiffness of the robot joints when pressed twice in rapid succession. This set of tactile sensor also includes LED lights that blink to indicate the state of the tactile sensor when triggered.

Two contact sensors are located in the foot bumpers. These sensors are triggered whenever the feet of the robot collides with an object. The purpose of this sensor is to detect objects and to raise an event or initiate an action when an object is detected.

The robot is equipped with two ultrasound channels comprising two transmitters and two receivers. These sensors enable the robot to estimate distances to obstacles in its environment. The detection range is between 1cm and 300cm. When an object is position at a distance less than 15cm relative to the robot, the robot is only capable of detecting its presence, not to measure the distance. The ultrasound sonars are capable of measuring distance at range of a maximum distance of 300cm and a minimum distance of 15 cm.

Two CMOS VGA 1.22Mpix cameras are installed on NAO's head. Both cameras have 60.97 degree horizontal field of view (HFOV) and 47.64 degree vertical field of view (VFOV) and are capable of high quality resolution at rates slightly over 15 frames per second on a Gigabit Ethernet connection. The lower camera is tilted to view an area close to the robot's feet, while the top camera is focused on the plane where Nao is facing. Figure 3.4 shows the robot camera field of view, and location of the camera on Nao's head.

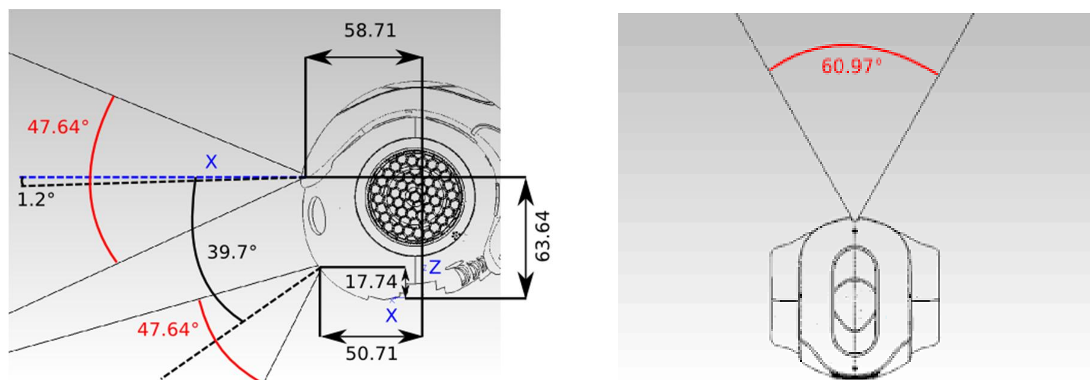

Figure 3.4. Nao's cameras field of view

The geometric location of the cameras on the robot head is such that the FOVs do not overlap. This design limits the capability of the robot for stereo vision tasks.

The force sensitive resistors (FSR) are located under each foot of the robot. These sensors indicate a change in resistance proportional to the amount of force exerted on them by the floor. This information is used during the walk, where at least one foot must maintain contact with the ground while walking. The value of the resistance enables the robot determine when the foot makes proper contact with the floor and ensures the stability of the robot. There are 8 FSRs in Nao, each foot has 4 FSRs located under its sole.

The robot is equipped with 36 magnetic rotary encoders (MREs), which provide information on all the joints of the robot. The MRE utilizes a change in magnetic field to determine the state of the motor shaft position. This is a feedback mechanism that measures the angular displacement of the robots rotary actuators controlling the movement of each joint.

The robot incorporates an inertial monitoring unit. The unit is composed of two axis gyroscope that provides an estimate for the torso orientation i.e. the yaw, roll and pitch angles in the world coordinates. IMU also includes a three axis accelerometer that provides information on the robot's motion in the world frame. The measurement of the inertia monitoring unit is quite noisy and thus the estimates differ significantly from the ground truth.

3.3 Programming

This section describes the software architecture and tools available for programing the robot. Aldebaran robotics provides a comprehensive software development kit that enables the development of applications for the robot by experts and novices. Nao SDK packages provide a means for professionals to develop applications for the Nao robot using supported programming languages. For the ease of programming, Aldebaran robotics provides a visual programming interface named Choregraphe. This GUI application includes the libraries of predefined blocks for simple robot tasks. It also enables users to control the robot by combining a set predefined and easily customizable blocks to form compound blocks that make up a complete and functional program sequence.

The software for Nao robot can be classified into two types: embedded software and desktop software. The embedded software runs on the motherboard located in the head of the robot and it is responsible for Nao's autonomous behavior. The operating system

of the robot is referred to as OpenNao. It is a Gentoo Linux distribution specifically developed for the Nao robot's needs. It provides all the libraries and modules required by Naoqi, another software that governs the behavior of the robot.

The desktop software runs on the user's computer. This software enables the programmer to create new behaviors and to control the robot remotely. It provides a link between the user's developed application and the Naoqi software running on the robot.

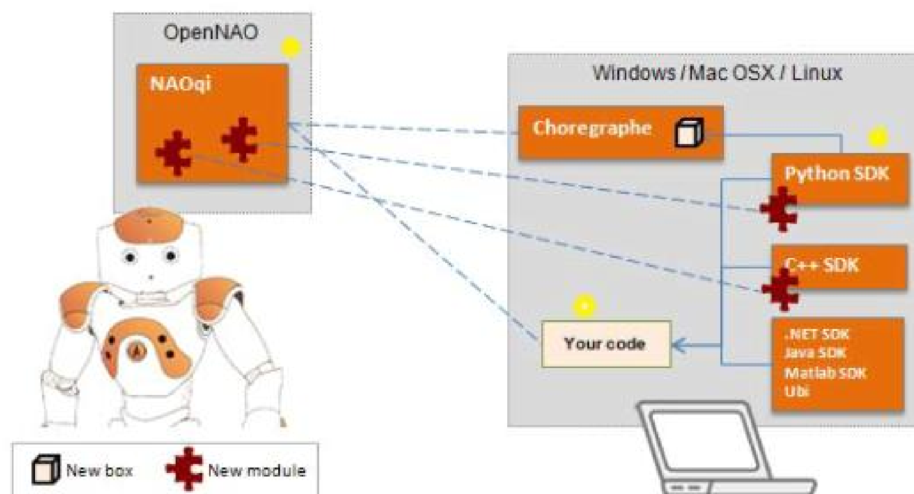

Fig 3.5. *Nao Software Interaction*

3.3.1 Naoqi framework

Naoqi is a distributed software framework that governs the behavior of the robot. It runs on the OpenNao operating system and it can also be run on the computer in a virtual simulator of the Nao robot, called webots. Robot functionality is encapsulated in software modules, so users can communicate to specific modules in order to access sensors and actuators. Communication between user defined modules and inbuilt modules are provided by the framework.

Naoqi framework currently supports five programming languages: C++, Python, URBI, Java and Matlab. It has also been tested in the Microsoft .Net framework for C#, F# and Visual Basic programming language. Amongst the specified programming languages, Python and C++ are the most developed for Nao. Programs written in C++ or Python

can be installed and run directly on the robot and remotely, whereas all other supported languages are supported only on the desktop computer.

This framework comprises a set of modules, e.g. memory, motion, vision, sonar and device communication manager (DCM). It functions as a broker by allowing homogenous communication between modules and sharing of information on resource availability. Thus the modules can access methods from other modules across the network. This communication enables capability for parallelism and synchronization. Naoqi also provides capability for monitoring the state of memory values. When the value at a memory location is changed, an event is raised and the appropriate action specified in the case of this specific event is carried out.

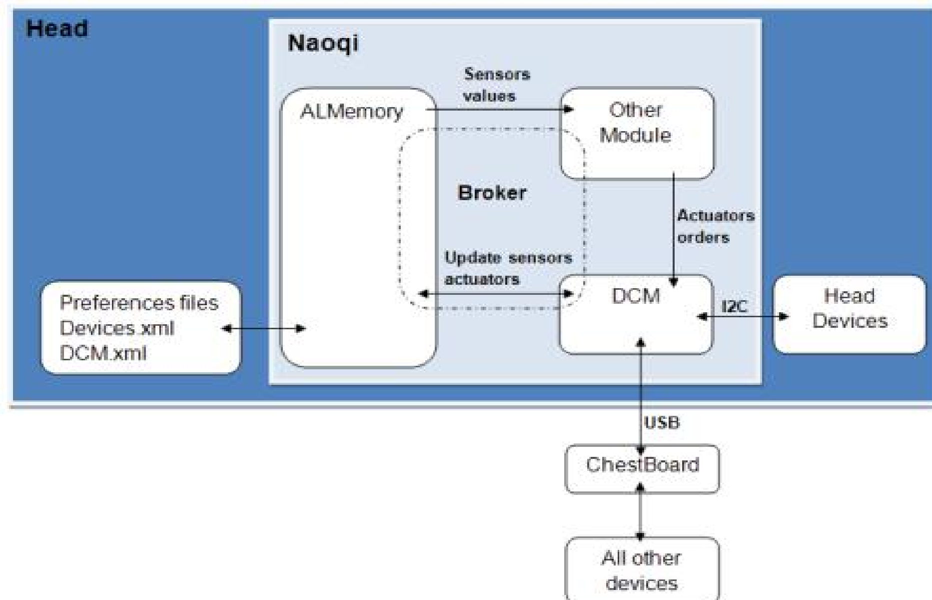

Figure 3.6. *Naoqi Framework*

Although Naoqi is a very useful framework, the software is still under development and several of the provided modules lack adequate documentation on programming.

In the implementations carried out in this thesis, an attempt was made to use a third party integrated development environment (IDE) for programming. The result was disappointing because some predefined methods within Naoqi modules were unresponsive. The support provided by the Aldebaran robotics community online was insufficient for resolving all the problems.

3.3.2 Device communication manager (DCM)

The device communication manager is a software module that manages communication with all electronic devices in the robot. It controls the robot directly by sending command calls to the robot's ARM controller, a type of processor based on a reduced instruction set computing architecture. The DCM is a link between the higher level architecture Naoqi and low level devices such as actuators and sensors. This includes electronic boards, joint position sensors and actuators. Devices such as the microphone, speakers, and camera are excluded from the DCM tasks. These devices are connected directly to head's on-board system. The DCM is essential for real-time processes such as access to image data, when a new image is captured by robot's camera, or for generating walk sequences. Modules for motion are designed to interact directly with actuators using the DCM while the extractor modules retrieve values from memory through the DCM. One pitfall in using the DCM is that robot stability is disabled when interacting directly and as such stability has to be ensured by the programmer.

3.3.3 Nao Simulation Environment

The GUI programming environment Choregraphe is also useful when a real robot is not available. It comes with a simulated Nao having features of the real robot, excluding functions that require the exteroceptive sensors are not available on the simulated robot. Programming is done through drag-and-drop, and by connecting graphically the function blocks.

An alternative robot simulation environment is webots, a third party application developed specifically for Nao robot. This offers simulation in a customizable virtual world, where all sensor, also the ones excluded in Choregraphe, are available to the user.

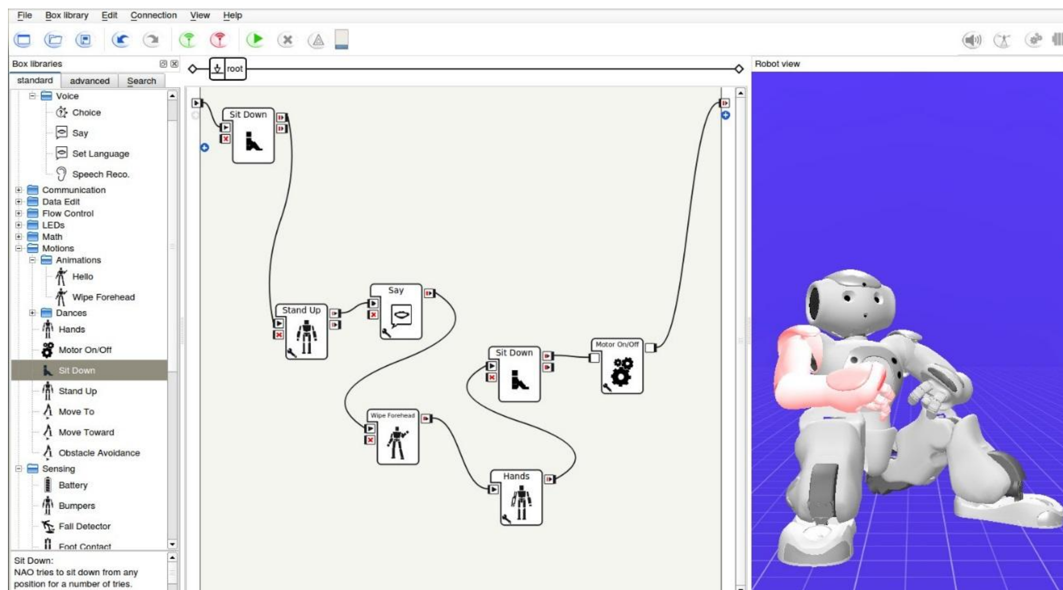

Figure 3.7. Screenshot of the Choregraphe software.

3.4 Motion

The motion module provides methods that enable the robot to move. It creates a "motion task" anytime a call to produce a movement is made through the API. The motion task computes the elementary commands to change motor angles and stiffness. The commands are scheduled such that they are performed only when the requested resources are available. The methods are categorized into four major groups: joint stiffness control, joint position control, locomotion control and Cartesian control methods. Within the motion module, safety measures such as self-collision avoidance, fall manager and smart stiffness are implemented.

In order to apply the motion module, the user is required to create a proxy for the module using Naoqi brokers. This proxy makes all methods within the module available to the user.

The joint stiffness control determines the torque to be generated when the robot is initialized for a motion task. The value of stiffness ranges between 0 and 1. A joint is compliant when the stiffness is set to 0 and it rigid when the stiffness value is 1. One important aspect of the motion module is that without setting stiffness "ON" any motion command specified to the module will remain unresponsive.

The joint position control comprises dedicated methods for controlling the position of Nao's joint. Each joint can be controlled individually by specifying the joint name or in parallel with other joints by specifying a chain name, where a chain would represent a group of joints such as "Body".

The Cartesian control is dedicated to controlling the effectors of the robot in Cartesian space using an inverse kinematics solver.

The Locomotion control comprises methods which make the robot move to places. Some of the methods for locomotion control are listed below.

- ALMotionProxy::moveTo (x, y, Θ), to set a target pose relative to the present pose, that Nao will walk to. The robot computes the required sequence of actions to reach the target.
- ALMotionProxy::move (direction, intensity) is used to set Nao's instantaneous velocity in SI units. This is usually used to control the walk from a loop with an external input such as visual tracker.
- ALMotionProxy::moveToward (direction, intensity) is used to set Nao's instantaneous normalized velocity. It is typically used to control the robot from a joystick.
- ALMotionProxy::setWalkTargetVelocity (direction, intensity) is used to set Nao's instantaneous normalized step length and frequency, and thus control its velocity indirectly.

3.5 Vision

NAO's vision system has modules for face detection, movement detection, landmark detection, visual compass, photo capture and red ball detection. The landmark detection module is utilized for this thesis. This module enables the robot to recognize special landmarks referred to as Naomarks.

Naomark consists of black circles with white triangle fans centered within the circle. The size, orientation and location of each fan on the triangle are used as a distinguishing feature for each Naomark. The range of detection for Naomark is accurate to about 2 meters and when the angle of view is less than 60 degrees.

Once the landmark detection module is subscribed to, the robot camera is activated. The detection of Naomark by the robot camera produces the landmark information that is stored in the robot memory. Each detection provides the following information; Marker ID, angle alpha and beta in radians representing the location of the center of the detected marker in terms of camera angles measured from the center of the field of view, and size - x and size - y i.e. the marker size in camera angles.

The landmark detection module is useful for distance measurements. Since the other available sensors are not suited for distance measurements, landmark detection provides an alternative approach for measuring the distance to a reference point relative to the robot. This capability makes it vital for localization.

4 METHODOLOGY AND IMPLEMENTATION

This chapter describes the methods for data acquisition, interpretation and transformation into useful information for robot localization. It also covers the algorithms and techniques used for determining the path which the robot uses in reaching its set objective.

4.1 Motion tests

A set of tests was carried out to ascertain the behavior of the Nao robot given a walk or turn command. The Nao robot was commanded to turn an angle (φ) and move a specific distance *(r)*. The start pose of the robot (x, y, φ) and the end pose after each action were measured. The data from the experiment provides a means to determine the walk length of Nao within which odometer measurements can be relied on and the nature of the uncertainty in walk. These measurements provide data about the behavior of the robot when commanded to walk or turn a certain value. From the measurements, the process/motion noise values were determined.

4.1.1 Distance Measurements

Measurements were carried out for walks of distances 50 cm, 100 cm and 200 cm respectively, starting from a point (x, y) chosen as the origin (0, 0). The robot was commanded to the specified distances in the robot's x-direction after which the covered distance and displacement from the walk path was measured. The outcome of these measurements showed that the lateral displacement mostly tends to the positive y-direction and the distance covered in the x-direction is lower when the lateral displacement is high. The x-coordinate and y-coordinate of the robot and finally the heading of the robot phi (ϕ) was recorded. Table 4.1 shows some of the recorded values.

Table 4.1. Walk measurements

Distance along x	$50.6 \pm 0.36\,cm$	$100.4 \pm 1.3\,cm$	$201.2 \pm 8.3\,cm$
1	50.0	99.2	188.5
2	50.5	100.8	185.5
3	50.4	99.2	205.2
4	50.5	100.4	200.0
5	50.5	102.7	203.7
6	51.0	100.8	201.4
7	50.5	99.6	201.8
8	50.3	99.2	201.8
9	51.4	99.6	194.9
10	50.8	101.3	201.3

4.1.2 Turn Measurements

Turn measurements were carried out by making the robot turn an angles *pi/4, pi/2* and *pi* at a point and the value of the output was recorded for twenty trials. The amount of overturn or under turn, and the robots displacement from the center point were recorded. It was observed during measurements that the robot mostly overturns for each of the specified turn angles. Furthermore, the turn increases in proportion to the specified turn angle. This provides some idea for determining a suitable value for bias and uncertainty in turn. Even so, the uncertainty in turn was determined to be a Gaussian distributed random value within means specified as the commanded turn angle and a variance given by the range of the error in turn. Table 4.2 shows the some of the values for turn measurements in each case.

Table 4.2: Robot turn measurements for angles 180, 90 and 45 degrees

No /Turn angle	*Mean: 193.7⁰ std.dev:5.3⁰*	*Mean: 99.8⁰ std.dev: 4.0⁰*	*Mean: 46.5 std.dev:1.6⁰*
1.	193	94	47
2.	198	98	45
3.	199	102	48
4.	198	96	46
5.	197	99	46
6.	195	98	45
7.	194	95	49
8.	193	100	47
9.	193	109	44
10.	198	102	48

4.1.3 Straightness measurements

Further test was carried out to determine the walk path for a 200 cm straight walk command. The (x, y) positions of the robot on a plane were recorded for every 20cm walk sequence for a total distance of 200 cm. The straightness measurements were repeated four times to illustrate how much the robot deviates from specified walk path at each walk. The result of the above measurements is vital for determining the optimal walk distance that limits the accumulated error in walk.

Figure 4.1 below shows a plot of the straightness measurement for four walks. The requested path is shown with the blue markings dotted line while actual paths are shown as 1st, 2nd, 3rd and 4th.

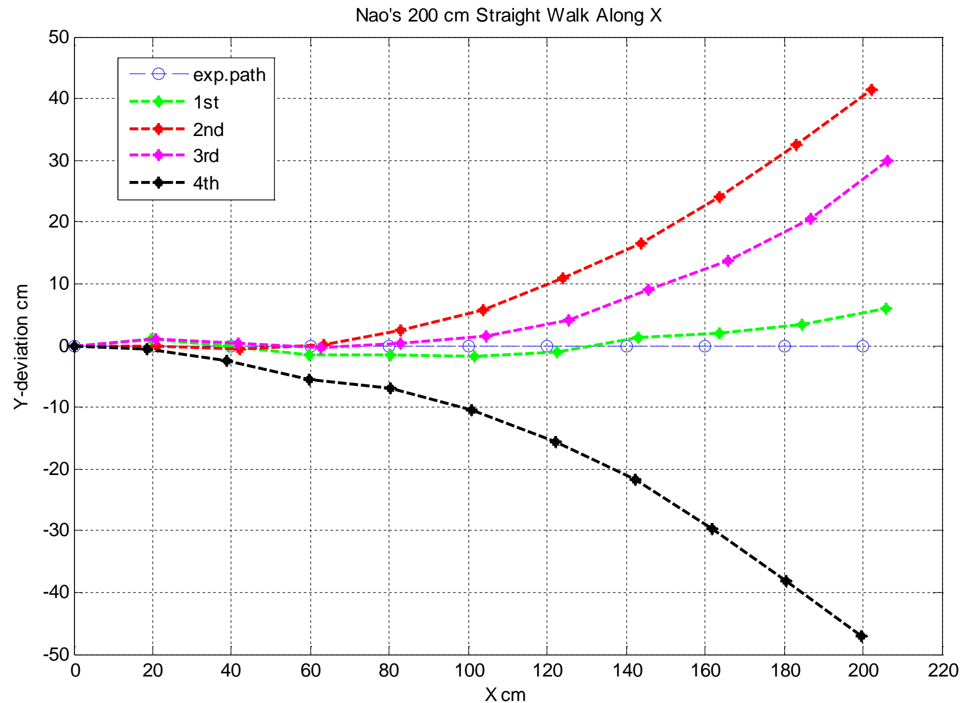

Figure 4.1. Walk path of Nao robot for 200cm command along x-axis.

4.2 Visual markers

The vision aspect of this work utilizes naomarks. The vision module of the robot already provides ability to recognize naomarks, red balls and faces. For localization, the landmark detection module is applied for determining the location of the naomark relative to the robot. The following sections describe the approach.

4.2.1 Naomarks

Aldebaran robotics provides a total of 29 markers. Each of these markers has a unique shape that distinguishes it. A marker is a black circle with a white pattern it. Encoded in the shape of the white pattern is the unique identity of the marker. The landmark detection module provides a means to acquire information from these markers when detected by nao camera. Practically, the robot can detect shape of the markers, the distance of the marker from the robot camera, and the unique ID of the marker. Using

the associated desktop monitor software, users can see the marker and the ID as detected by the robot.

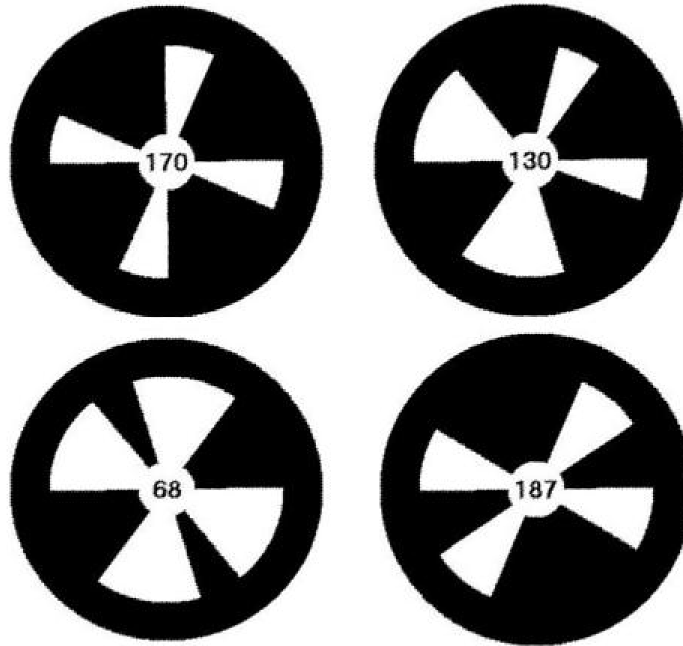

Figure 4.2. Naomarks samples

4.2.2 Limitations of the landmark module

Although the landmark detection module offers a simple approach to visual localization, there exist several limitations to its application. This module is plagued by the quality of the images acquired by the camera. As stated in the documentation, the first key requirement is sufficient illumination. The detection relies on contrast differences in the image. The proper illumination must be between 100 Lux and 500 Lux. Lighting conditions below this range often results in misidentification of markers or no detection in the worst case.

Secondly the tilt of the marker's plane relative to the camera must be between +/- 60 degrees for detectability. For optimal performance, the naomark must be in the direct line of sight of the robot. Experiment performed on the robot using both cameras showed no detection for landmark placed on the floor plane, while the landmark placed on the walls were detected albeit some misclassifications.

The third limitation is the size of the marker within the image and the range of detection by the camera. The minimum size is approximate 0.035 rad which corresponds to 14 Pixels in a QVGA image, while the maximum size is approximately 0.40 rad or 160 pixels within the QVGA image. At this marker image size ranges and marker real size being 108.54 mm, the distance range for detection is from 30 cm to about 200cm.

4.2.3 Landmark detection info

The data for detected landmark is obtained directly from the robot's memory using the memory proxy's getData ( ) method. The data for any observation is structured as follows.

```
ALLandMarkDetectionInfo
 {
        TimeStamp, MarkInfo [N], CameraPoseInNaoSpace,
        CameraPoseInWorldSpace, CurrentCameraName
}
```

The Timestamp field contains the time at which a landmark was detected in the image from the robot camera. The MarkInfo [N] is the list of N landmarks detected and it contains detailed information such as shape information and Marker Id. The MarkerInfo field is structured as follows:

```
MarkInfo
{
        ShapeInfo,    MarkerId
}
```

The MarkerId is the number written on naomark which corresponds to its pattern.

```
ShapeInfo
 {
        heading, alpha, beta, sizeX, sizeY
}
```

The shape info field contains the heading angle this describes the orientation of the naomark about the vertical axis. The field alpha and beta represents the naomark's center in terms of camera angles in radian while sizeX and sizeY are the camera angles.

The CameraPoseInNaoSpace and CameraPoseInWorldSpace expresses the 3d vector and pose angles of the camera with respect to the robot and to the world when the image was captured. Finally the CurrentCameraName can be either the "CameraTop" or "CameraBottom" indicating which camera was used for image acquisition.

4.2.4 Marker coordinates

The coordinates of the marker acquired by the robot camera are computed using the shape info. First we need to know the physical dimension of the naomark detected. With this we can calculate the distance of the marker from the robot using the following steps.

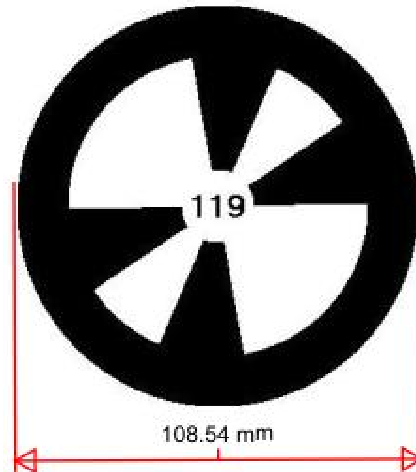

Figure 4.3. Naomark dimension

The variable (*S*) is the distance of the marker from the camera can be calculated using the angular size (*a*) and the marker size (*m*) as show in the following equation. The marker size is the size of the printed marker. For this experiment, the size is 108mm. The parameters *sizeX* and *sizeY,* for which identical values are given in the image of detected markers, are the dimension from center-most to the edge on the horizontal and vertical image axis respectively.

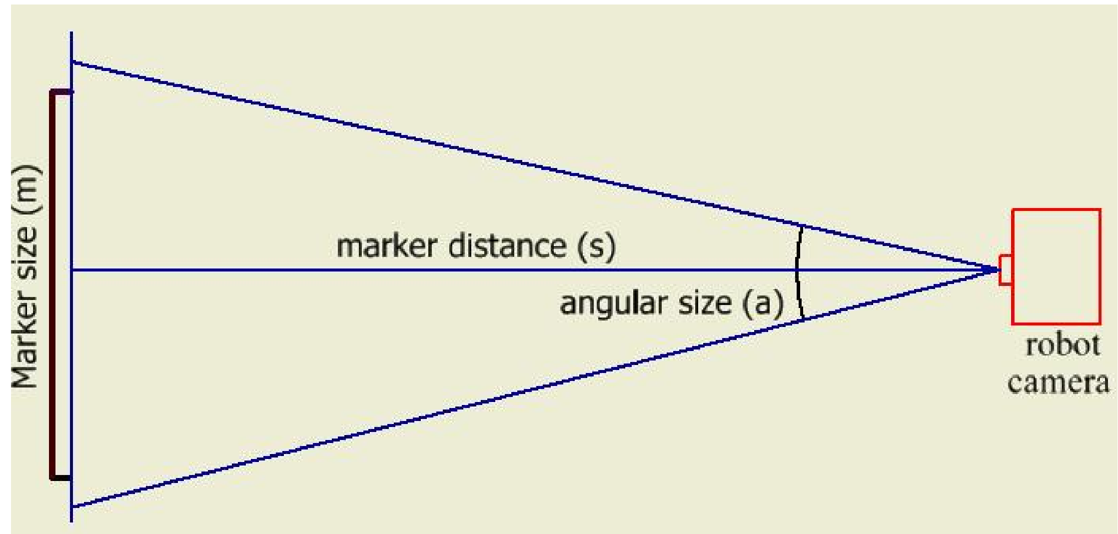

Figure 4.4. Nao Camera angles

$$S = \frac{m/2}{\mathrm{atan}(a/2)} \quad (1)$$

The angles alpha and beta are used to obtain the transformation from the robot frame to the landmark. To obtain the coordinate of the marker in the robot frame, a transformation from camera frame to the robot frame is required. This transformation is performed using methods defined in the transform class of the implementation.

The computation for this transformation is presented in equation (2)

$$RobotToLandmark = landmarkToCameraRotationalTransform * landmarkToCameraTranslationalTransform * cameraToRobot \quad (2)$$

The result is a transformation matrix which includes the (x, y, z) coordinates of the landmark in the robot frame. Since we are concerned only with the position horizontal distance of the landmark from the robot, the z-coordinate is ignored in future applications.

4.3 Environment map

The environment map was built using a set of 13 known markers. The environment is designed as a 2 m x 2 m area. Each marker was located in the map at specific coordinates, as shown in the table below.

Table 4.3. Global coordinate of markers on map

Marker ID	x-position (cm)	y-position (cm)
170	200	55
141	125	0
112	0	50
117	0	100
131	200	118
130	70	0
138	0	150
108	0	180
175	200	40
125	200	90
127	145	200
109	200	166
146	26	200
80	60	200
143	102	200
124	180	200
119	175	0
171	25	0

4.4. Visual localization

The location and orientation of the robot in the global frame are determined based on the 2D relative coordinates of markers detected by robot in robot frame and the coordinates of same markers in global frame. Earlier experiments carried out by [12] provide a proof of the approach. This section describes the approach developed by [12] and how the pose of the robot is determined.

First the transformation between the global and local frame of the robot is outlined. The robot has its X axis from the robot pointing forwards and Y axis direction pointing to the left of the robot. Figure 4.5 shows NAO’s frame and the global frame together.

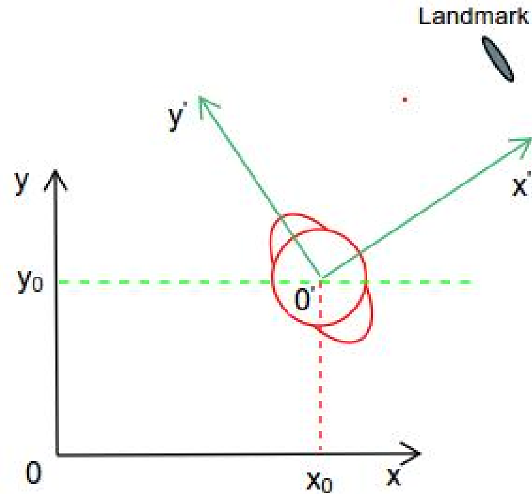

Figure 4.5. Nao Frame and global frame

The 2D transformation between the robot frame of marker and the global frame is a combination of rotation and translation. This is shown in equation (3).

$$\begin{pmatrix} X_{global} \\ Y_{global} \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} X_{robot} \\ Y_{robot} \end{pmatrix} + \begin{pmatrix} X_0 \\ Y_0 \end{pmatrix} \qquad (3)$$

The subscripts “global” indicates coordinates of the marker in the global frame, “robot” indicates coordinate of the marker in the robot frame while X_0 and Y_0 are the robot location in the global frame. The angle φ denotes the orientation of the robot in the global frame. The equation can be rewritten in a more compact form as:

$$\begin{pmatrix} X_{global} \\ Y_{global} \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & X_0 \\ \sin\varphi & \cos\varphi & Y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_{robot} \\ Y_{robot} \\ 1 \end{pmatrix} \qquad (4)$$

Since equation (3) has three unknowns the knowledge of relative coordinates and global coordinates of one marker alone is not sufficient to determine the location and orientation of the robot in the global map. If the coordinates of at least two markers are known and assuming that the position of the robot remains unchanged during detection of the markers, two sets of equations for the two detected marker coordinates would provide four equations with three unknowns, the robot's coordinates and orientation. An illustration of is shown in figure 4.6. The equations for two marker case read as:

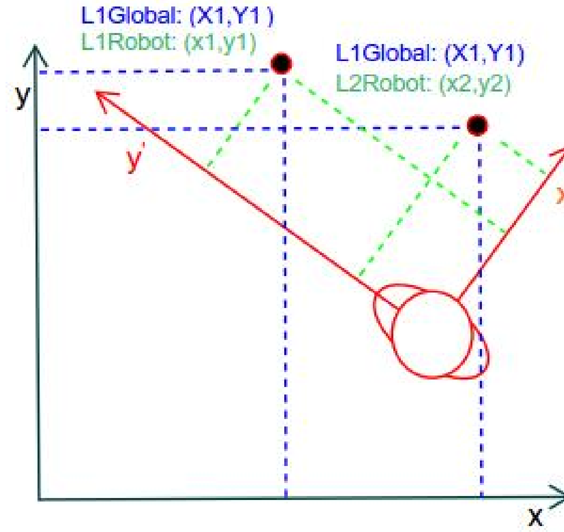

Figure 4.6. Representation of marker locations in global and robot frame

$$\begin{pmatrix} X1_{global} \\ Y1_{global} \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & X_0 \\ \sin\varphi & \cos\varphi & Y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X1_{robot} \\ Y1_{robot} \\ 1 \end{pmatrix} \quad (5)$$

$$\begin{pmatrix} X2_{global} \\ Y2_{global} \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi & X_0 \\ \sin\varphi & \cos\varphi & Y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X2_{robot} \\ Y2_{robot} \\ 1 \end{pmatrix} \quad (6)$$

These four equations have three unknowns and thus the system is overdetermined. One simple way to deal with this is consider $cos\varphi$ and $\sin\varphi$ as independent variables, and the check their constraint after solution. The two landmarks for localization should be at

a considerable distance from each other, and the markers must not be coplanar otherwise the errors in pose estimation would be significant. These conditions present a challenge to the localization, since the landmark detection range is limited by the camera field of view and the marker tilt range. A solution to these is to rotate the head of the robot by a specific angle $(0 \pm \theta)$ during marker detection and then carry out a transformation of the relative marker coordinates by angle $(0 \mp \theta)$ during localization. The localization scheme is presented in Pseudo code 4.1

```
1 Begin
2    Obtain a list of marker coordinate in robot frame.
3    For any pair of markers:
4        If they have different X and Y with each other (non-coplanar)
5                Solve coupled equations and get the pose of robot for the two markers.
6                If pose is within limits of map
7                        Update pose list
8                End if
9        else if they have the same X and Y (coplanar)
10               Transform coordinates of marker, go to line 5
11       End if
11 End
12 determine average over robot pose list
13 End
```

Pseudo code 4.1. Determining robot pose

The implementation in line (4 – 6) deals with non-coplanar marker conditions required for accurate pose estimation and the solution the set of equations for the selected markers. Lines (7 - 9) checks that the computed pose is within specified limits in terms of map area and for the orientation between $-pi < value < pi$. If the condition is met, the pose is update to a list of poses from which an average pose is determined in line (11).

The parameters X_0, Y_0 and φ from equations 5 and 6 can be solved either by using C++ library for symbolic named “symbolic C++” or analytically. For this work, the analytic approach was chosen due to missing dependencies in Linux GCC library required by Symbolic C++. The expression for X_0, Y_0 and $cos\varphi$ and $sin\varphi$ is shown in equations 7, 8, 9a and 9b. Note that variables $X_1, Y_2, X_2,$ $and\ Y_2$ represent the global coordinates

of the landmarks while x_1, y_1, x_2 and y_2 the represents coordinates of landmarks in robot frame.

$$X_0 = \frac{-(x_1x_2X_1 - X_1x_2^2 - X_2x_1^2 + x_1x_2X_2 - X_2y_2^2 + x_2y_1Y_1 + X_1y_1y_2 + X_2y_1y_2 - x_1y_2Y_1 - X_1y_2^2 + x_2y_1Y_1 + x_1y_2Y_2)}{((x_1 - x_2)(x_1 - x_2) + (y_1 - y_2)(y_1 - y_2))} \quad (7)$$

$$Y_0 = \frac{-(-X_1x_2y_1 + x_2y_1X_2 + x_1x_2Y_1 - x_2^2Y_1 + x_1y_2X_1 - x_1y_2X_2 + Y_1y_1y_2 - Y_1y_2^2 - x_1^2Y_2 + x_1x_2Y_2 + Y_2y_1^2 + y_1y_2Y_2)}{((x_1 - x_2)(x_1 - x_2) + (y_1 - y_2)(y_1 - y_2))} \quad (8)$$

$$cos\varphi = \frac{-(X_1(x_2 - x_1) + X_2(x_1 - x_2) + Y_1(y_2 - y_1) + Y_2(y_1 - y_2))}{((x_1 - x_2)(x_1 - x_2) + (y_1 - y_2)(y_1 - y_2))} \quad (9a)$$

$$sin\varphi = \frac{-(y_1(X_1 - X_2) + Y_1(x_2 - x_1) + y_2(X_2 - X_1) + Y_2(x_1 - x_2))}{((x_1 - x_2)(x_1 - x_2) + (y_1 - y_2)(y_1 - y_2))} \quad (9b)$$

For a planar robot the orientation in world frame can be described as $0 \leq \varphi \leq \pi$ or $-\pi \leq \varphi \leq 0$ Since arc-cosine provides results in this range, the cosine results from equation (9a) is relied upon for orientation values.

An important observation made in the course of experiments was that, when the robot is at very close proximity (< 30 cm) to landmarks the visual localization the estimates of robots position and orientation are largely inconsistent.

4.5. Implementation of Particle filter based localization

This section describes application of the particle filter for localization on the Nao robot. The basic particle filter has four steps.

- Initialization
- Prediction
- Measurement update
- Resampling

First a set of samples or random particles representing the beliefs of the robot state is created within the confines of the map. For the prediction step, a motion model that simulates the movement of the robot on each particle is determined. The particles in this step are evolved based on the robots motion model. The next step is the measurement update. Here weights are assigned to each particle based on information from sensors. The weights are normalized such that particles well-compatible with the sensor data – in this case marker-based localization – are highly weighted, while particles less compatible with data are assigned low weights. The last step is the resampling. The idea of the resampling step is simply that particles with very low weights are abandoned, while particles with high weights are retained and replicated. In order that the total number of particles is maintained, identical copies of high-weighted particles are formed. This is referred to as sampling with replacement.

4.5.1. Motion Model

At the prediction stage, the effect of control or command on the pose of the robot is modelled by applying the control action on Nao's motion model. First, consider the control or action required to produce a motion on the robot. Given Nao's pose on the plane as $[x, y, \theta]^T$ where (x, y) is the position of Nao in world coordinates and θ is the orientation of the robot coordinate system. The effect of changes $[\Delta x\ \Delta y]^T$ in the robots position on the plane can be described by a rotation followed by a translation.

The robot rotates $\Delta\theta = \theta(k) - \theta(k-1)$. Where $\theta(k) = \arctan(\Delta y / \Delta x)$ and then translates the distance $r = \sqrt{\Delta x^2 + \Delta y^2}$ to its destination. The location and orientation of the robot after every control input k is given by:

$$x(k) = x(k-1) + r(k)\cos\Delta\theta(k)$$
$$y(k) = y(k-1) + r(k)\sin\Delta\theta(k) \qquad (11)$$
$$\theta(k) = \theta(k-1) + \Delta\theta(k)$$

In order to predict the probability distribution of the pose of the robot after each motion, the effect of noise on the process must be modelled. For both the translational and rotational motion robot the noise are modelled as an additive Gaussian noise.

The turn noise is modelled as a Gaussian $N(n_t, \sigma_t)$ with mean with the men rotational error and variance determined through the experiments in section 4.1.2

The translational noise arises from two sources. The first is the error in distance travelled and the second is the changes in orientation during translational motion. The changes in orientation during translation are responsible for robot's deviation from the desired direction of translation (lateral translation). The error due to distance is travelled was determined through experiments in section 4.1.1. Analytical modelling of the deviation of the second noise parameter is difficult, so the approach chosen was to limit the distance moved at each step to 50cm and then model the noise as a Gaussian with $N(n_d, \sigma_d)$. So at each walk of 50cm the robot is assumed to have deviated with a mean value of n_d, and a variance σ_d.

Thus the motion model including the noise is given by:

$$x(k) = x(k-1) + [r + N(n,\sigma)](k)\cos(\varphi + N(n_t,\sigma_t)))(k)$$
$$y(k) = y(k-1) + [r + N(n,\sigma)](k)\sin(\varphi + N(n_t,\sigma_t))(k) \qquad (12)$$
$$\theta(k) = \theta(k-1) + [\varphi + N(n_t,\sigma_t)+](k)$$

4.5.2. Measurement and updates

To determine the weight of each sample in the particle filter, a measurement of the robot's location and orientation is obtained from observation of landmarks. The observations are coordinates of landmarks in robot frame. The visual localization process described in Section 4.4 provides the measured location of the robot in the global map. The measurement from the sensor is assumed to be noisy and thus the measured location and orientation of the robot is subject to measurement noise.

The probability of each sample particle is computed using the difference in Cartesian coordinates and orientation of the particles and the measured location.

The probability is computed as shown:

$$P(X_i^{k+1}|X_m) = \frac{1}{\sqrt{2\pi\sigma_d}} e^{\frac{-(\Delta x_i)^2}{2\sigma_d^2}} \frac{1}{\sqrt{2\pi\sigma_d}} e^{\frac{-(\Delta y_i)^2}{2\sigma_d^2}} \frac{1}{\sqrt{2\pi\sigma_\emptyset}} e^{\frac{-(\Delta \theta_i)^2}{2\sigma_\emptyset^2}} \quad (13)$$

Where X_i^{k+1} represent the probability of *i-th* particle given the measurement X_m.

$\Delta x_i = x_i - x_m$, $\Delta y_i = y_i - y_m$, and $\Delta\theta_i = \theta_i - \theta_m$

The constants σ_d and $\sigma_\emptyset$ are the measurement noise and they indicate the confidence with which we weight each measurement in terms of the terms of position and orientation.

4.5.3. Resampling

The resampling step adopts sampling with replacement. The process of resampling is described in the Algorithm 1. First the cumulative sum of particle weights is computed, then a random numbers selected from a uniformly distributed set in the range [0, 1]. The next step applies the resampling algorithm described in [13].

Algorithm 1 below presents a formal description of the "select with replacement" algorithm. The resampling produces a new set of particles that describes the next state of the robot.

Algorithm 1: Select with replacement sampling algorithm

1: Input: floats $W[N]$, $P[N]$
2: $Q = cumsum(W)$ {Cumulative sum of weights $Q_i = \sum_i^j W_i$ }
3: $index = rand([0,1]) * N$ {Select a random index from the set of particles}
4: $beta = 0.0$
5: $mw = max(W)$ {Maximum weight}
6: $for\ i\ in\ N$
7: $beta = beta + rand([0,1]) * 2 * mw$
8: $while(beta > w(index))\ do$
9: $beta = beta - w[index]$
10: $index = (index + 1)\ \%\ N$
11: Output: $return\ P[index]$

4.6. Motion planning

This section deals with robots movement from its initial position to a specified target position with obstacle present in unknown locations. Motion planning utilizes Nao sensors to determine how the Nao would reach its target. The bumper sensors on the Nao's feet are used to detect obstacles at collision. Initially the path is assumed to be free from obstacles and so the initial plan of the robot is a straight walk to the target. Figure 4.7 shows the robot and the target in 2D plane.

First, the direction of the target is determined based on the robot orientation and the target coordinates on the map. The turn angle required to align the robots heading to target direction is determined. Next the shortest angle to achieve the alignment is computed. The difference in heading between the target and the robot is specified as the turn angle of the robot.

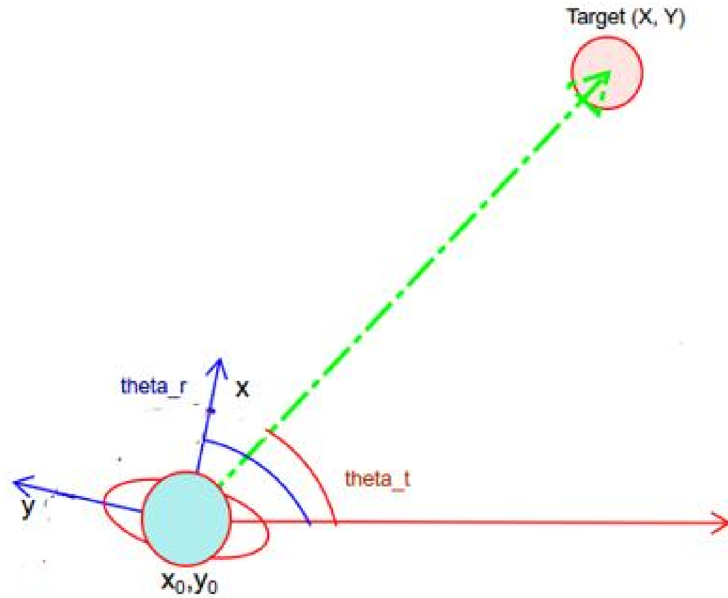

Figure 4.7. Illustration of robot and with a specified target coordinate.

$$\varphi = \arctan\left(\frac{Y_{target} - Y_0}{X_{target} - X_0}\right) \quad (15)$$

The Euclidean distance between the robot and the target is computed using equation 16.

$$R = \sqrt{\left(X_{target} - X_0\right)^2 + (Y_{target} - Y_0)^2} \quad (16)$$

The robot moves in the set direction until it collides with an obstacle. On collision, the robot stops and computes the distance covered prior to collision. The location of the obstacle is determined using odometry readings. Then the robot evades the obstacle by taking the following actions. First the robot steps back a few steps, randomly turns an angle pi/6 either to the left or right direction, then moves forwards a distance equivalent to twice the backward steps. Next the robot re-localizes and computes new turn angle and distance to target. If the robot is at target it stops; if the robot is still away from the target, then a turn and move action is perfumed to guide robot from present state to target. The program flow is described by the flow chart in Figure 4.8.

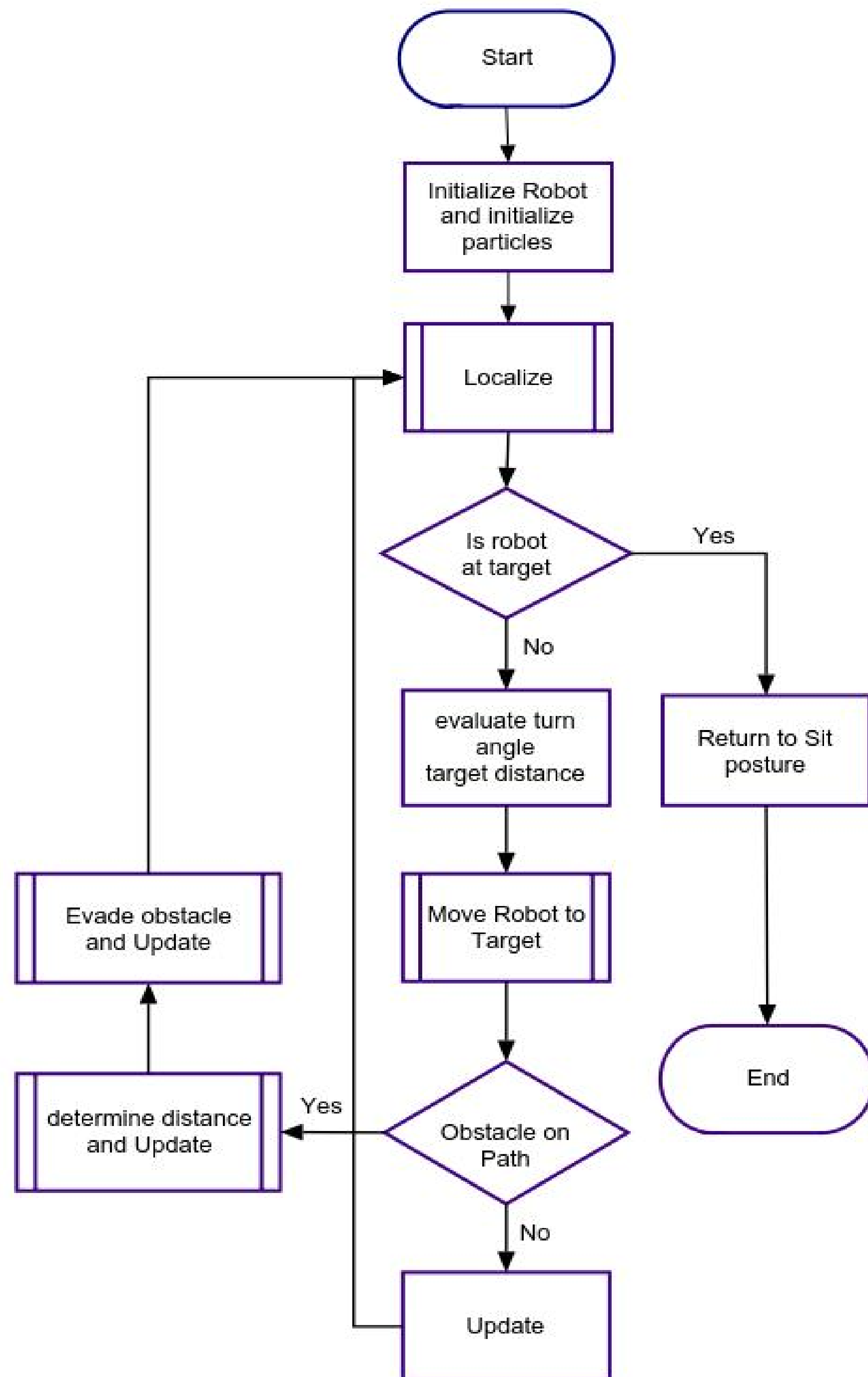

Figure 4.8. Flow-chart of the navigation and localization program

At initialization, Nao is set to a stand posture then particles are created and the target point is specified. Next the robot localizes by calling the landmark detection module and using equations (15) and (16). Then the robot calculates the difference between its pose and the target specified. If the difference greater than a specified tolerance, the

robot turns to align itself to target then attempts movement in a straight path to target. The particles are updated at after each move action and resampled to obtain particle estimates of robot pose within the map.

The figure below shows the class structure and the list of methods for the implementation:

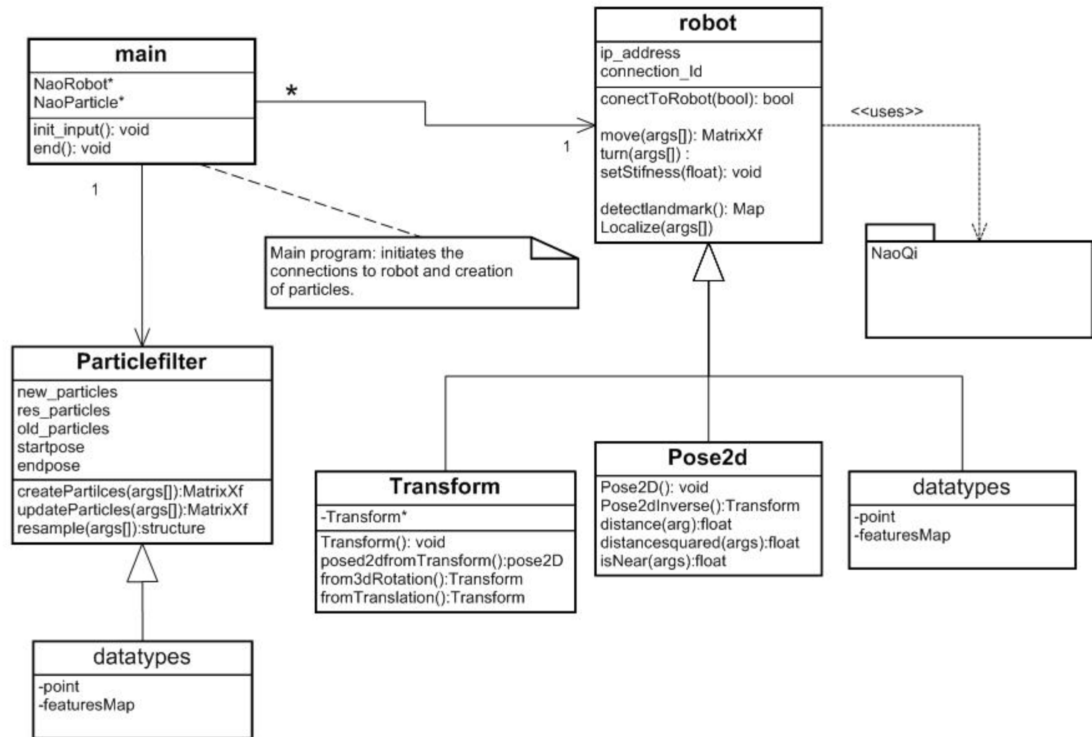

Figure 4.9. Class Structure of the implementation

The class diagram shows the main class which includes the map definition and all variables used during runtime, the Robot class and the Particlefilter class. Also shown are helper classes like transform and Pose2d and the Aldebaran naoqi package which provides functionalities like proxies for connection to modules for motion, posture and general hardware management. The code written for the classes is provided in the Appendix section.

5 EXPERIMENTS AND RESULTS

This section describes the results of the experiments performed and the particle filter localization implementation. It also discusses the result from the planning aspect.

5.1. Initial Tests

These test consist of data collection from the odometry module, see Chapter 4. The purpose of this test was to determine the deviation in walks and the error in the direction perpendicular to the walk. The uncertainty in the robot's motion is modelled using the result of these tests.

The plot shows the error walk in x-axis, and the deviation in direction perpendicular to the specified walk path. The plot shows output for 200 cm, 100cm and 50 cm respectively. The walk for each distance specified was repeated 20 times.

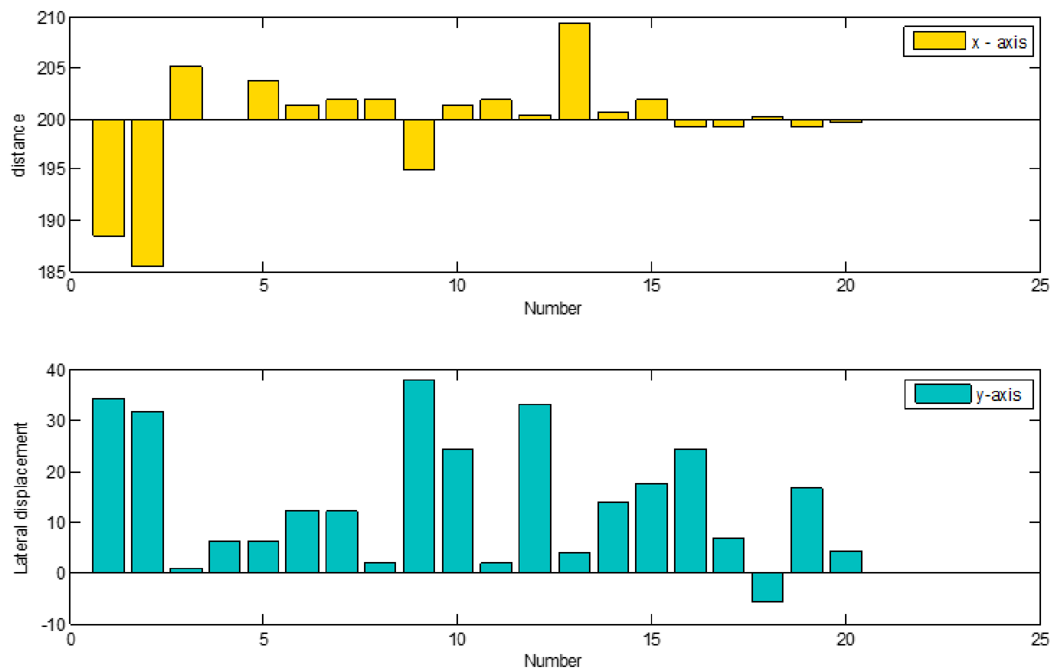

Figure 5.1. Distance and lateral displacement for 200 cm straight walk.

From the data, it can be observed that the deviation in the direction perpendicular to walk path is minimal for small values of specified walk direction, thus only short walks was used during goal to goal behavior and the particle filter implementation.

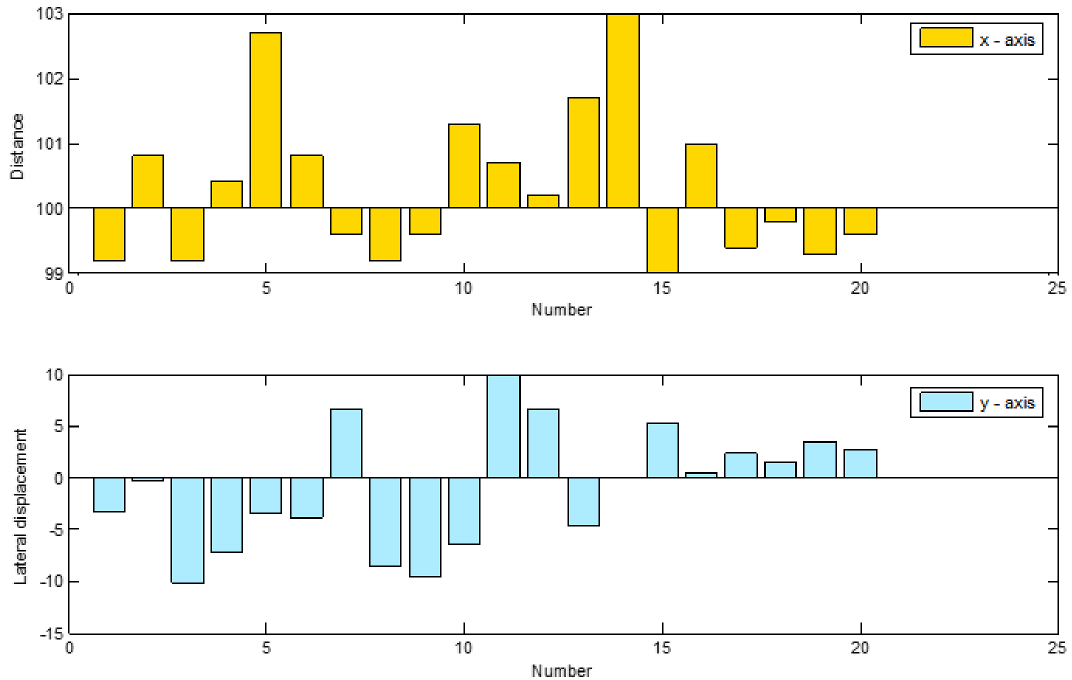

Figure 5.2. Distance and lateral displacement for 100 cm straight walk.

5.2. Particle filter localization

First the robot is placed at pose $[0.4, 0.4, 90^0]$ then it was commanded to localize using the landmarks. The perceived pose of the robot is $[0.35, 0.48, 96]$. The focus of the robot was directed at markers at the top and right of the map. The origin of the map being the bottom left with coordinates $(0,0)$.

As an evaluation metric, the pose error of the particle estimate relative to the true pose of the robot in the x, y plane was considered. First, the position error is considered to determine how well the particle filter estimates true position of the robot within the map. Next the heading error which is important if the robot is to navigate through a path with obstacles to reach a target position.

Figure 5.3 shows the real location of the robot and the location determined from visual localization. The error in the pose estimate as observed from visual localization is quite tolerable. The difference in heading is 6 degrees while the differences in the both coordinates are less than 0.1m.

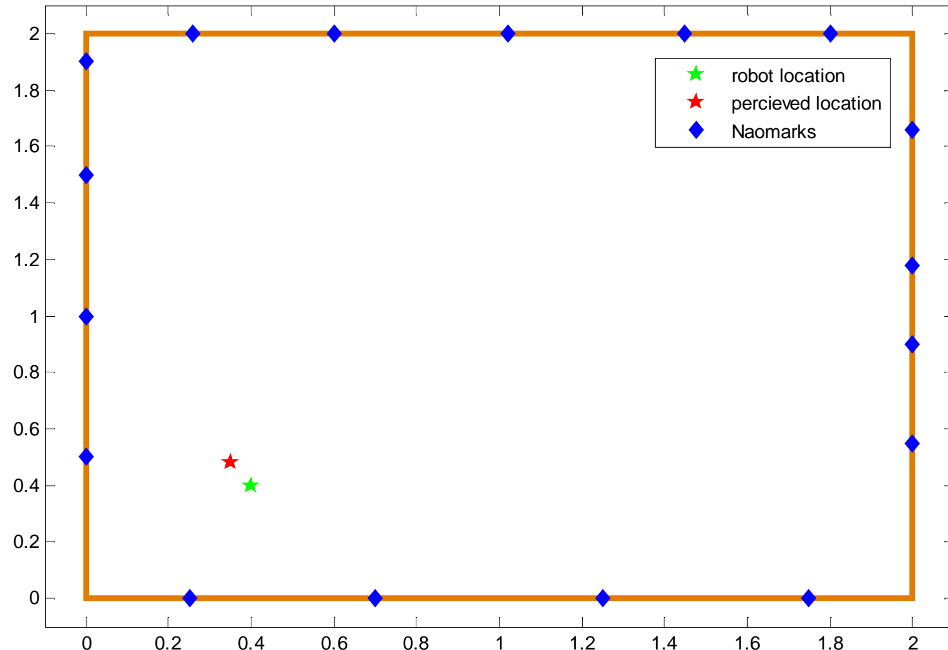

Figure 5.3. Visual localization result

Cases were observed where the robot detected a non-existing marker with number 13 and coordinates [-infinity, -2]. The condition occurred when the robot is at a position less than 30cm from the marker and when the marker tilt angle is greater than 60^0. This is likely as a result of limitations in the detection range and the marker tilt angles.

Thus, the visual landmark localization only provides an estimate of the robots pose within the map. The next step is to apply particle filter localization, the approach used for the particle selection is to select particles in the neighborhood of values obtained from visual localization algorithm. Using this criterion, the particle filter performs better in terms of convergence towards the true pose of the robot.

Two simple localization experiments were carried out using known start poses. In the first case, the robot is given walk and turn commands and in the second case the robot was given the coordinates of a target to reach. The robot was set at a predetermined pose and commanded to move a distance and turn a specified angle. The robot is set at the poses given in leftmost column of Table 5.1 and the robot is commanded to localize using the landmarks. The map area is 2.0 m by 2.0 m. The second column in Table 5.1 shows the distance and angle specified. The third column shows the visual estimate of the robot's pose after the robot has completed its walk. The fourth column shows the

particle estimate of the robot pose at the end of the walk and the last column shows the true pose of the robot.

Table 5.1. Localization experiment using known start points, distance and direction command

Initial pose [x, y, theta]	Move Command [distance, angle]	Visual estimate [x, y, theta]	Particles estimate [x, y, theta]	Real pose [x, y, theta]
[0.4, 0.4, 45^0]	[0.8, 30^0]	[1.04, 0.85, 82^0]	[0.97, 0.78, 78^0]	[1.04, 0.80, 75^0]
[0.4, 0.4, 60^0]	[1.0, 60^0]	[1.06, 0.98, 121^0]	[1.10, 1.22, 118^0]	[1.01,1.18, 124^0]
[0.4, 0.3,-20^0]	[1.2, 90^0]	[0.76, 1.08, 65^0]	[0.80, 1.27, 72^0]	[0.83,1.32, 68^0]
[0.4, 0.3,-45^0]	[1.4, 120^0]	[1.42, 1.11, 78^0]	[1.48, 1.25, 81^0]	[1.46, 1.20, 82^0]

To apply the particle filter algorithm, the robot was commanded to move from its initial pose of **[0.4,0.4,45]** a distance 0.8m and turn an angle 30^0. Figure 5.4 shows the particle distribution after the move command of (0.8, 30^0). Initially, the particles are randomly distributed on the map. The particle estimates represents the distribution of particles after the movement update, while the residual particles represent the particles that survived after the resampling stage. The pose of the robot is determined by averaging over residual particles.

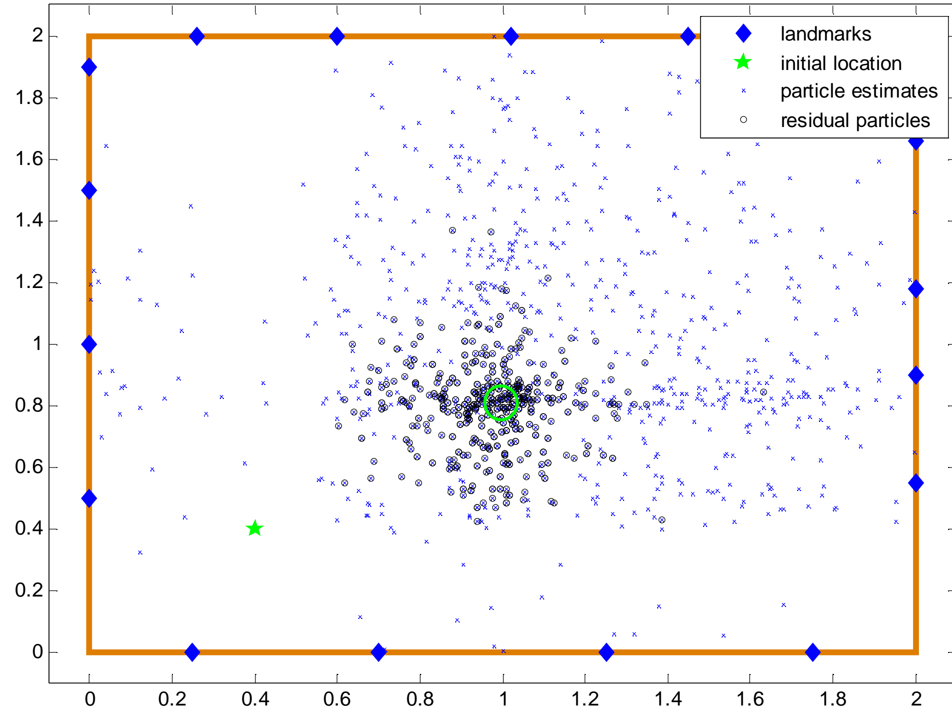

Figure 5.4. Particle pose estimate after move [0.8, 30^0]

To obtain the estimated pose from the particles, an average of residual particles' coordinates was computed. The average values for the pose was estimated as **[0.97, 0.78, 78^0]**. The green ellipse shows the possible position of the robot as obtained by the averaging of residual particles.

The subsequent rows in Table 5.1 shows the result for the next three trials. The robot was commanded to move [1.0, 60^0], [1.2, 90^0], and [1.4, 120^0] respectively. Comparing the visual estimates, real pose and the particle filter estimated pose for each trial, the errors in position of the robot is less than 0.15 m for both x and y coordinates while the error in the direction was within 0 - 5 degrees. The directional error though substantial, is still within acceptable limits considering that the robot was limited short walks only. A longer walk distance would result in a substantially high deviation from expected end position on the map.

Next the experiment was repeated, this time, the command was specified as coordinates within the map. The robot was given a target coordinate and it was made to determine

the angle and required distance to move in reaching the target. The leftmost column of Table 5.2 shows the start pose, the next column shows the target coordinates.

At the first run, the robot is set at **[0.4, 0.4, 45^0]** and was commanded to move to coordinates **[1.3, 1.0]**, the particle estimate of robot location is shown in the 4th column of table 5.2. The visual estimate of robot pose indicates that the robot has a pose [1.42, 1.11, 44^0] while the particle filter estimate was computed as the average of residual particles is **[1.37, 1.02, 39^0]**. The experiment was repeated thrice with different start positions and target coordinates, the results are shown in the subsequent rows of the table. The real pose of the robot is shown in rightmost column of Table 5.2. The error in the pose estimates when compared with the real pose is typically less than 0.15 for both x and y coordinates, while the direction is less than 10^0. The worst result obtained is a position error of 0.18m for the y-coordinate of the robot for target coordinates [1.3, 1.0].

Table 5.2. Localization experiment using known start points and target coordinates

Initial pose [x, y, theta]	Target coordinates [x, y]	Visual estimate [x, y, theta]	Particles estimate [x, y, theta]	Real pose [x, y, theta]
[0.4, 0.4, 45^0]	[1.3, 1.0]	[1.42, 1.11, 44^0]	[1.37,1.02, 39^0]	[1.46, 1.20, 35^0]
[0.3, 0.3, 60^0]	[1.0, 0.8]	[1.24, 0.40, 38^0]	[1.04, 0.71, 42^0]	[1.10, 0.62, 38^0]
[1.1, 1.2, 60^0]	[0.5, 0.6]	[0.45,0.54, -112^0]	[0.52, 0.55,-126^0]	[0.50, 0.50,-120^0]

Figure 5.5 shows the particle distribution when the robot was set at [0.4, 0.4, 45^0] and the target given as [1.3, 1.0]. The green ellipse shows an estimated location of the robot.

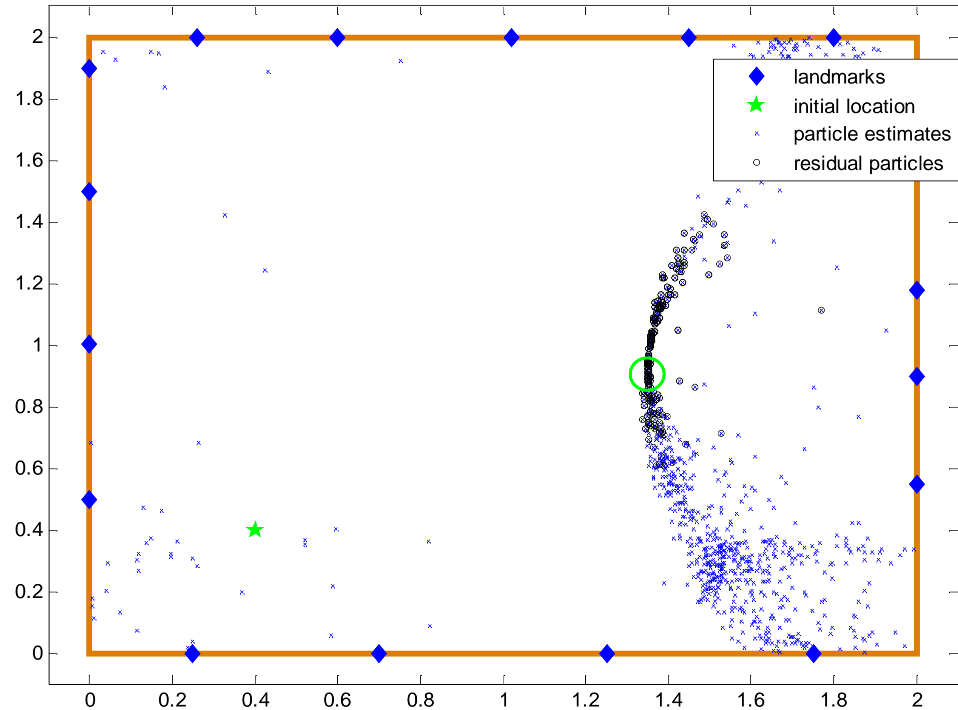

Figure 5.5 Particle pose estimate for target coordinates [1.3, 1.0]

Figure 5.6 shows the result of localization when the robot was set at pose $[1.1, 1.2, 60^0]$ and was commanded to walk to target coordinates [0.5, 0.6]. The green ellipse indicates the possible location of the robot from the particle estimates. The average of poses at this location was $[0.52, 0.55, -126^0]$, while the visual localization result indicates the robot pose as $[0.45, 0.54, -112^0]$. At each run the robot mostly ended its walk within a circle of 0.15m radius around the target coordinates. When the robot stops at any point within this circle, we assume that the robot has successfully reached the specified target.

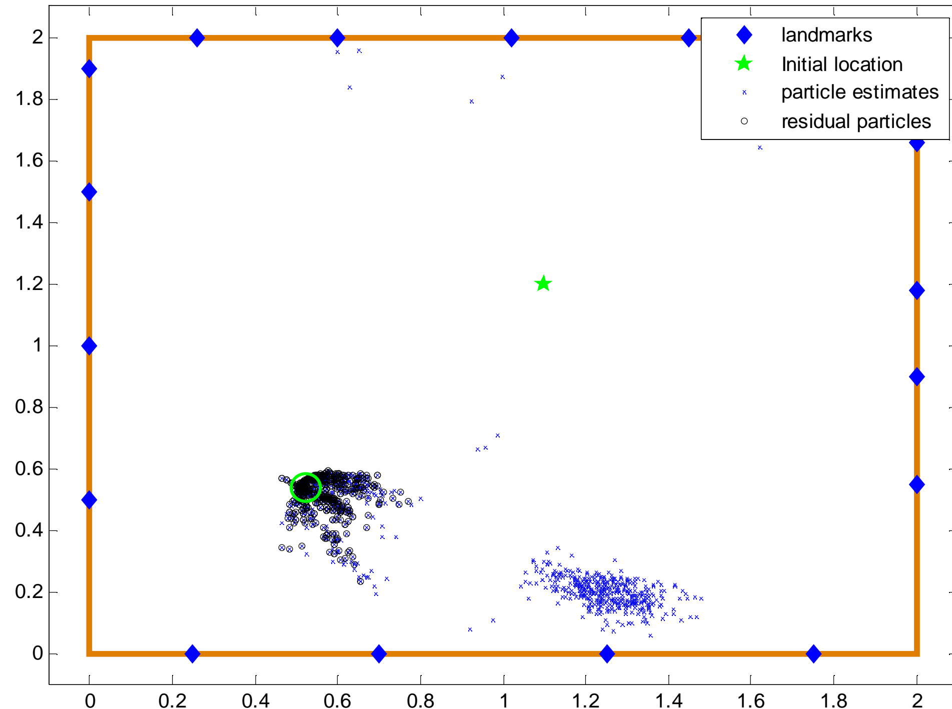

Figure 5.6 Particle pose estimate for target coordinates [0.5, 0.6]

5.3. Localization while robot moves

A target position was specified with obstacles present in the path of the robot. The robot was commanded to move to the target coordinates and to localize at every bump into an obstacle or when a move is complete. The distance to move was computed as the Euclidean distance between the robot's initial position and the target coordinates. The obstacle position is unknown to the robot and the detection is achieved only when robot collides with obstacle. The target coordinates was set as [1.3 1.2], while the robot starts pose was [0.3 0.3 120^0]. The target is reached when the difference between robots coordinates on the map and the target coordinates is less than 0.2m. The following figures show the localization results as the robots navigates a path to the target.

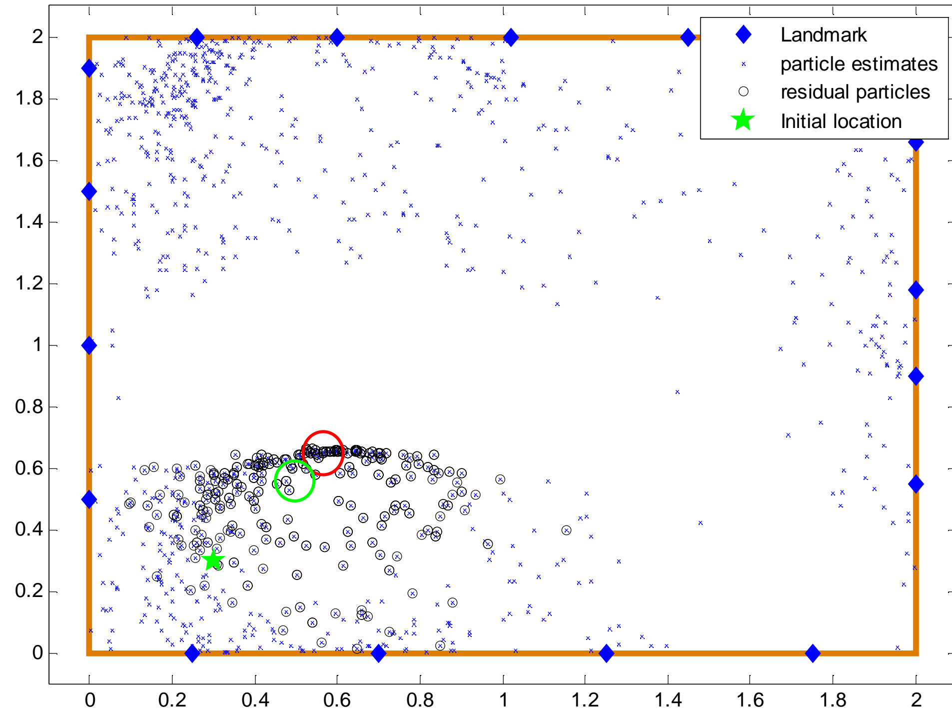

Figure 5.7 Particle pose estimate for (a) an obstacle at [0.6, 0.6].

Figure 5.7 shows the robot localization result after bumping an obstacle at point [0.6, 0.6]. The distance moved as indicated by odometry reading was 0.42m. The red ellipse shows the actual location of the robot after the collision with the obstacle. The particle filter estimates the robot's pose as **[0.55,0.55,53^0]** indicated by the green ellipse while the visual estimate of robots pose is **[0.6,0.54,48^0]**. The error in the particle filter estimated coordinates compared to the actual location of the robot is quite insignificant at [x = 0.05, y<0.05]. The difference between particle estimates of the robot's heading the true heading is 5 degrees. The large disparity in the values of the direction parameter is not surprising as residual particles were fairly dispersed, and there were extrema in the direction component of the some of the surviving particles.

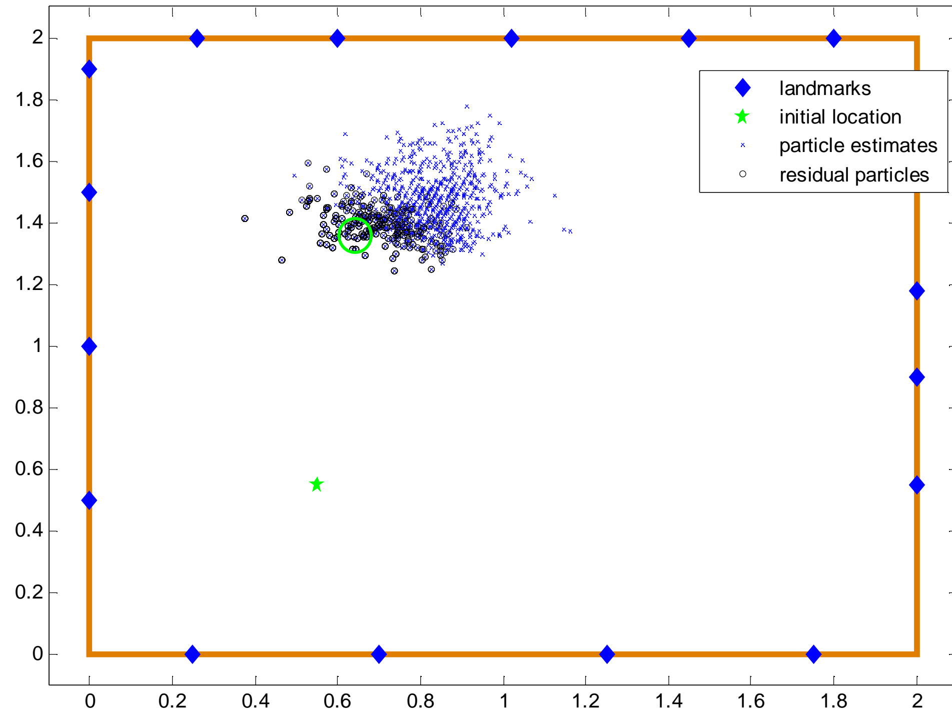

Figure 5.8. Estimate of robot position (b) obstacle at [0.6 0.6] evaded.

The initial pose for the next step was [**0.55,0.55,53⁰**], the pose at collision with the obstacle. The robot turns an angle pi/6 or –pi/6 to avoid the obstacle, then computes the distance to reach the target and moves the equivalent distance in the new direction. Figure 5.8 shows that robot moved to coordinates [0.65, 1.5]. At this position the visual localization result indicates the robot is at **[0.56,1.40 ,84⁰]**. The particle filter estimated pose is **[0.61,1.35,75⁰]**. The results for other stop point are given in Table 5.3.

Table 5.3 Results of autonomous localization with obstacles

Step	Initial pose [x, y, theta]	Visual estimate [x, y, theta]	Particles estimate [x, y, theta]
1	[0.30, 0.30, 120⁰]	[0.6 0 0.54, 48⁰]	[0.55, 0.55, 53⁰]
2	[0.55, 0.55, 53⁰]	[0.56,1.40, 84⁰]	[0.61, 1.35, 75⁰]
3	[0.61, 1.35, 75⁰]	[1.40, 1.25, 47⁰]	[1.28, 1.08, 53⁰]

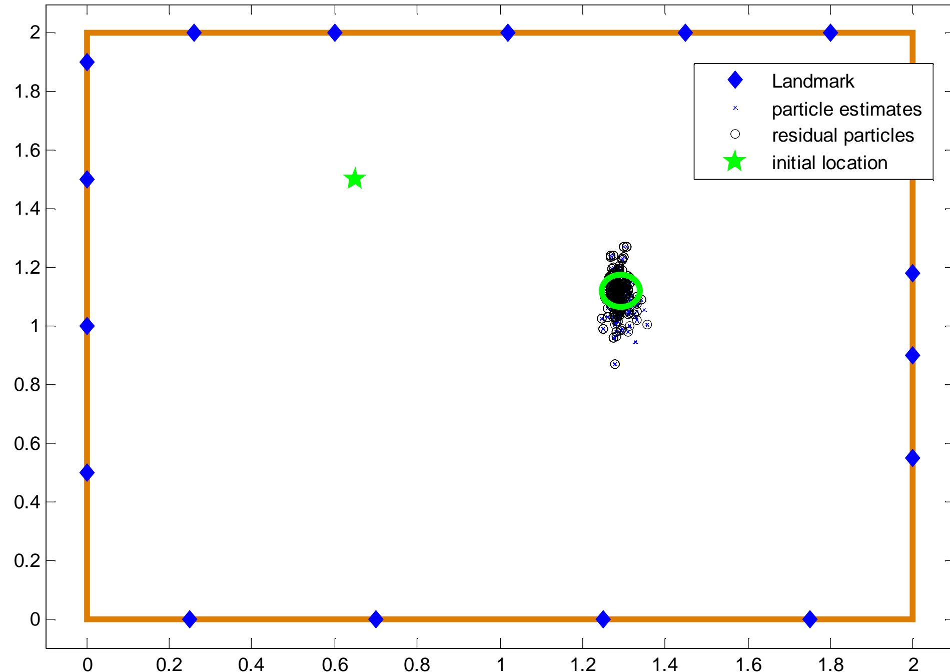

Figure 5.9. Estimate of robot position at (c) final target coordinates [1.3, 1.2]

Figure 5.9 shows particle distribution at the end of the localization task. The final estimate of the robot pose was **[1.28,1.08, 53^0]**. The visual localization gives the robot pose as **[1.40,1.25, 47^0]**. The absolute error in the final position of the robot as measured by the visual localization module [0.12, 0.05] and that from the particle filter estimate is [0.07, 0.08].

Figure 5.10 shows the estimated path the robot transverse to reach the target. The red rectangles represent obstacle placed on robots path. The green colored profile represents the path estimated by the particle filter, while the red colored profile represent the visual path estimate the robot. The profiles shown are depicted as linear between stop points although in actual sense they are fairly curved.

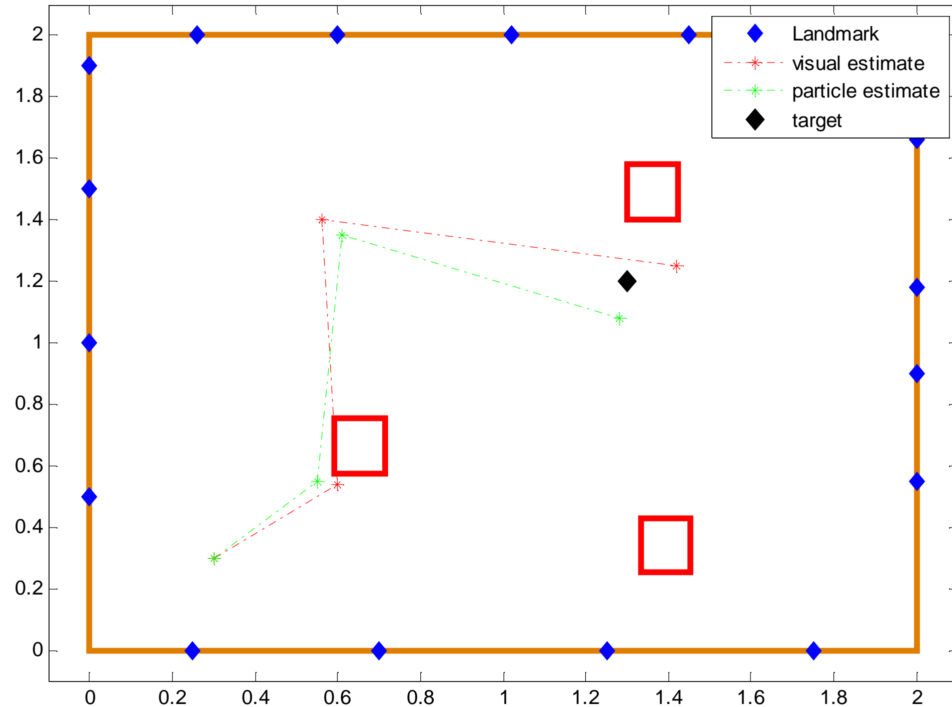

Figure 5.10 Estimate of robot path to target.

This is because the MRE sensors provide data only about the position at start of motion and the stop position. The data from the MRE sensors was used to estimate the actual distance moved at each walk and the estimate is applied for particle filter update whenever there is a collision with an obstacle on robots. This ensures the particle update tracks the actual motion of the robot.

From the above experiments, it has been established that the particle filter can provide a good estimate of the robot pose and for the localization of the robot while moving. Although the results showed only minute deviations from the target position, this performance is highly dependent on the quality and method of sensing and the visual landmark localization scheme. Combining visual landmarks and the particle filter for localization, though computationally intensive for the Nao platform, can be relied upon for indoor localization. The quality of visual estimates dictates the quality of the particle that survives during the resampling phase.

6 CONCLUSION

The work presented in this thesis focused on the problem of localization of the humanoid robot within a semi-mapped environment using particle filters. The environment of operation was assumed to be indoor and structured. The methods presented relied on the identification of known and unknown artificial landmarks, while obstacles in the scene were at unknown locations. The motion of model was discretized based on short walk epochs.

The results of the experiments performed showed that the particle filter localization scheme implemented was accurate enough to be used for path planning and navigating the robot from any point in the map to a target position and despite its computational complexity, it is a powerful scheme for robot pose estimation. A possible continuation of this work is making the robot extract information directly from the environment in combination with artificial markers and using this information to continuously localize as the robot moves. The challenges that may arise with this that real time image processing on the robot is highly resource consuming, thus, all processing and information extraction from images must be done over the network on a remote computer. Also considering the resource requirements for running a particle filter, combining image processing and the particle filter with a large number of particles might result in a low performance.

In conclusion, the thesis work provided an opportunity to explore mobile robot localization schemes and to tackle sensing and path planning tasks. It also provided an opportunity to learn the capabilities of the Nao robot concerning autonomous operation.

REFERENCES

[1] David Scott Barret, “Propulsive efficiency of a flexible hull underwater vehicle”. (Thesis Ph.D) --MIT, Dept. of Ocean Engineering, 1996. http://hdl.handle.net/1721.1/10559 Accessed: 14.05.2015

[2] Gerald Cook, “Mobile Robots: Navigation, Control and Remote Sensing, First Edition”. © 2011 Institute of Electrical and Electronics Engineers. Published 2011 by John Wiley & Sons, Inc.

[3] Borenstein, J. and Koren, Y., "Obstacle avoidance with ultrasonic sensors." IEEE Journal of Robotics and Automation, Vol. RA-4, No. 2, 1988, pp. 213-218.

[4] Borenstein, J. and Koren, Y. "Critical Analysis of Potential Field Methods for Mobile Robot Obstacle Avoidance." Submitted for publication in the IEEE Journal of Robotics and Automation, February 1990.

[5] E. Hashemi, M.G. Jadid, M. Lashgarian, M. Yaghobi, M.R.N. Shafiei “Particle filter based localization of nao biped robots”. [www] http://www.mrl-spl.ir/downloads/EhsanHashemi-etla-44-ieee-2012.pdf accessed: 14.05.2015

[6] E. Wirbel, B. Steux, S. Bonnabel, and A. de La Fortelle, “Humanoid robot navigation: from a visual slam to a visual compass,” Networking, Sensing and Control (ICNSC), 2013 10th IEEE International Conference, (2013)

[7] E. Rosten and T. Drummond. “Machine learning for high-speed corner detection”. Computer Vision – ECCV 2006 Lecture Notes in Computer Science Volume 3951, 2006, pp 430-443

[8] H. Bay, T. Tuytelaars, L. Van Gool, “SURF: Speed Up Robust Features”. Computer Vision – ECCV 2006 Lecture notes on computer science Vol 3951 2006, pp 404-417

[9] Š. Fojtu, M. Bresler, and D. Pruša, “Nao robot navigation based on a single VGA camera”, in Computer Vision Winter Workshop 2012, 2012.

[10] Nist´er, D. ”A minimal solution to the generalized 3-point pose problem.” In: CVPR 2004. pp. I: 560–I: 567 (2004)

[11] Aldebaran Nao software and robot documentation, [www] http://doc.aldebaran.com/1-14/ accessed: 06.12.2015

[12] Mojtaba Heidarysafa, ”Heuristic localization and mapping for active sensing with humanoid robot NAO”. (Thesis MSc) – Faculty of automation science and engineering. TTY. (2015)

[13] Artificial intelligence for robotics, udacity course material lesson 3. https://www.udacity.com/course/artificial-intelligence-for-robotics--cs373, accessed: 05.26.2015

[14] “Robotuna”. http://www.roboticsbusinessreview.com/article/p45 accessed: 05.06.2015

APPENDIX 1

ROBOT CLASS

```
/* Methods: connectToRobot, moverobot, setStiffness */

#ifndef ROBOT_H
#define ROBOT_H

#include <string>
#include<math.h>
#include <stdlib.h>
#include <time.h>
#include <map>

/** -----Eigen matrix headers------*/
#include <Eigen/Dense>
#include <Eigen/StdVector>

/** ------Boost C++ headers------------------*/
#include<boost/random/normal_distribution.hpp>
#include<boost/random/mersenne_twister.hpp>
#include<boost/random/variate_generator.hpp>
#include<boost/random.hpp>

/** -------- Aldebaran Nao headers-------*/
#include "alproxies/almotionproxy.h"
#include "alproxies/alrobotpostureproxy.h"
#include "alproxies/altexttospeechproxy.h"
#include "alproxies/allandmarkdetectionproxy.h"
#include "alproxies/almemoryproxy.h"
#include "alerror/alerror.h"
#include "alproxies/alnavigationproxy.h"
#include "alproxies/alsonarproxy.h"

/** ---------User defined headers---------*/
#include "datatypes.h"
#include "pose2d.h"
#include "transform.h"

using namespace std;
using  namespace Eigen;
using namespace dataobjects;

class Robot
{
private:

    int _portID;
    bool _connected;
    std::string _ip_address;
    boost::random::mt19937 rdigen;


    AL::ALMemoryProxy *_memoryProxy;
    AL::ALMotionProxy *_motionProxy;
    AL::ALNavigationProxy *_navProxy;
    AL::ALTextToSpeechProxy *_speechProxy;
    AL::ALRobotPostureProxy *_postureProxy;
```

```
    AL::ALLandMarkDetectionProxy *_flandmarkProxy;
    AL::ALSonarProxy *_sonarProxy;


public:

    Robot();
    ~Robot();

    typedef std::map<int,std::pair<float,float> > Container;
    typedef Container::const_iterator It;
    typedef Container::const_reverse_iterator rIt;

    const static float PI=3.14159265359;
    //----------------variables---------------
    std::vector<float> init_position, final_postion;
    ObsLocation ObstaclePos;
    float globalHeading; //range [0,2pi]
    float localHeading;  //range [-pi,pi]
    float targetheading;  //value in degrees.....
    MatrixXf startpose;
    MatrixXf endpose;
    int data_rows;
    MatrixXf detectedMarkers;
    MatrixXf totalDetectMarkers;

    //-------------public methods---for robot-------------------------
//
    void headstraight();
    //values dist_steps, angle_step used only in simulatons
    MatrixXf turn(float deltaphi);
    void redirectHeading(float &changeheading);
    float getDistance(MatrixXf &startpose, float &target_x, float
&target_y);
    float getHeading(MatrixXf &startpose, float &target_x, float
&target_y);
    MatrixXf move(float rwalklength, MatrixXf
&angle_turn_theading_diff,       MatrixXf &startpose); //subfunction
of moverobot
    float detectObstacles();
    MatrixXf avoidobstacle(float rev_x, float rev_y, float turn
    Container detectlandmark();
    MatrixXf localize(Container& dlandmarks, Container& globalmap);

    bool connectToRobot(string ip_address, bool _state);
    void init();
    void setStiffness(float val);
    void notify(string &message);
};
#endif // ROBOT_H

#include "robot.h"

Robot::Robot()
{
    //specifying default values for parameters.
    this->_ip_address = "";
    this->_portID = 9559;
    this->_connected = false;
    //Set the speed of the joints....

    this->startpose=MatrixXf::Zero(1,3);
```

```
    this->endpose=MatrixXf::Zero(1,3);
    this->globalHeading=0;

}
Robot::~Robot()
{
    delete this->_flandmarkProxy;
    delete this->_postureProxy;
    delete this->_motionProxy;
    delete this->_memoryProxy;
    delete this->_navProxy;
    delete this->_sonarProxy;
    delete this->_speechProxy;
}

void Robot::headstraight()
{
    AL::ALValue::array(0.3);
    this->_motionProxy->setAngles("HeadPitch",0,0.4);
}

MatrixXf Robot::move(float rwalklength, MatrixXf
&angle_turn_theading_diff,MatrixXf &startpose )
{
    MatrixXf temp(1,6);
    //returns a (1,6) matrix containing the endpose and distance, x,y.
    float distance_moved,dx,dy;
    float mindist=detectObstacles(); //checks for obstacles around
robot path.
    float safedistance=min(mindist, rwalklength);
    //------------------added---------to prevent robot from staying
stationary-------
    if(safedistance==0)
    {
        safedistance=0.15;
    }
    //--------------------------can be reomoved if it does not
optimal----------------
    bool use_sensor=true;
    init_position = this->_motionProxy->getRobotPosition(use_sensor);
    this->_motionProxy->moveTo(safedistance,0,0);
    final_postion = this->_motionProxy->getRobotPosition(use_sensor);
    cout<<"\ninit_position "<<init_position<<endl;
    cout<<"\nfinal position "<<final_postion<<endl;
    //determine how much robot moved.....using Mre sensors..
    dx=final_postion.at(0)-init_position.at(0);
    dy=final_postion.at(1)-init_position.at(1);
    distance_moved=sqrt(dx*dx +dy*dy);

    this->startpose=startpose; //to be refined later.

    float angle_diff=angle_turn_theading_diff(0,1);
    float turn_angle=angle_turn_theading_diff(0,1);
    float turn_angle2rad = turn_angle*(PI/180);

    this-
>endpose(0,0)=startpose(0,0)+distance_moved*cos(turn_angle2rad);
    this-
>endpose(0,1)=startpose(0,1)+distance_moved*sin(turn_angle2rad);

    this->endpose(0,2)=startpose(0,2) + angle_diff;
    endpose(0,2)=(endpose(0,2)>180)?(endpose(0,2)-360):endpose(0,2);
```

```
    endpose(0,2)=(endpose(0,2)<-180)?(endpose(0,2)+360):endpose(0,2);

    temp<<endpose,distance_moved,dx,dy;
    return temp;
}

MatrixXf Robot::avoidobstacle(float rev_x, float rev_y, float turn)
{
    MatrixXf temp(1,6);
    float distancemoved,dx,dy;
    bool use_sensor=true;
    init_position = this->_motionProxy->getRobotPosition(use_sensor);
    this->_motionProxy->moveTo(rev_x, rev_y,0); //reverses  (x,y, 0)
backwards......
    final_postion = this->_motionProxy->getRobotPosition(use_sensor);

    this->_motionProxy->moveTo(0,0,turn);
    dx=final_postion.at(0)-init_position.at(0);
    dy=final_postion.at(1)-init_position.at(1);
    distancemoved=sqrt(dx*dx +dy*dy);

    endpose(0,0)=endpose(0,0)+distancemoved*cos(turn);
    endpose(0,1)=endpose(0,1)+distancemoved*sin(turn);
    endpose(0,2)=(endpose(0,2)+ turn*180/PI);

    endpose(0,2)=(endpose(0,2)>180)?(endpose(0,2)-360):endpose(0,2);
    endpose(0,2)=(endpose(0,2)<-180)?(endpose(0,2)+360):endpose(0,2);

    temp<<endpose,distancemoved,dx,dy;
    return temp;
}

Robot::Container Robot::detectlandmark()
{
    Container LandmarkData;

    AL::ALValue markdata;
    std::string memvalue="LandmarkDetected", strvalue="landmarkTest";
    string jointname="HeadYaw",currCamera = "CameraTop";
    float stiffness= 1.0, time =1.0,landmarkSize=0.108; // size in
meters(diameter of landmark);
    bool isAbsolute = true;
    //use num to determine number of computations for localization
within map.

    this->_motionProxy->stiffnessInterpolation(jointname, stiffness,
time);
    AL::ALValue targetAngles = AL::ALValue::array(1.3963f,0.6981f, 0,
-0.6981f,-1.3963f);
    AL::ALValue targetTimes = AL::ALValue::array(1.5f,1.0f,1.0f,1.0f,
1.5f);

    for(int i=0; i<targetAngles.getSize(); i++)
    {
        this->_motionProxy->angleInterpolation(jointname,
targetAngles[i], targetTimes[i], isAbsolute);
        this->_flandmarkProxy->subscribe(strvalue);
        sleep(1);
        markdata=this->_memoryProxy->getData(memvalue);
        while(markdata.getSize()==0)
        {
           markdata=this->_memoryProxy->getData(memvalue);
```

```
        }
        AL::ALValue size=markdata[1];
        int num=size.getSize();
        //declares a temporary markerinfo array
        this->detectedMarkers=MatrixXf::Zero(num,3);
        //-------------landmark properties detected-------------------
------------
        float wzCamera[num], wyCamera[num], angularSize[num];
        float disCameraToLandmark[num];
        //--------array to store computation results------------
        float xtemp[num], ytemp[num];
        float x,y;
        for(int count=0; count<num; count++)
        {
            //retrieving landmark positions in radians & angular size
in radians
            wzCamera [count] = markdata[1][count][0][1];
            wyCamera [count] = markdata[1][count][0][2];
            angularSize[count] = markdata[1][count][0][3];
            //Compute distance from the robot camera to landmark.
            disCameraToLandmark[count] = landmarkSize / ( 2 * tan(
angularSize[count] / 2));

            vector<float> results=this->_motionProxy-
>getTransform(currCamera,2,true);
            ASE::Transform robotToCamera(results);
            //Compute the rotation to point towards the landmark and
the translation to point towards landmark
            ASE::Transform cameraToLandmarkRotTrans =
ASE::Transform::from3DRotation(0, wyCamera[count], wzCamera[count]);
            ASE::Transform cameraToLandmarkTranslTrans
=ASE::Transform::fromPosition(disCameraToLandmark[count], 0, 0);
            //Combine transforms: gives landmark position {R} space.
            ASE::Transform robotToLandmark = robotToCamera *
cameraToLandmarkRotTrans * cameraToLandmarkTranslTrans;
            x=robotToLandmark.r1_c4;
            y=robotToLandmark.r2_c4;

            //Rotation to align marker to default head orientation.
             xtemp[count]= x;  //x*cos(target_Angles[i]) + y*(-
sin(target_Angles[i]));
             ytemp[count]= y;//x*sin(target_Angles[i]) +
y*cos(target_Angles[i]);

            //stores the detected marker info.
            this->detectedMarkers(count,0)=markdata[1][count][1][0];
            this->detectedMarkers(count,1)=xtemp[count];
            this->detectedMarkers(count,2)=ytemp[count];
        }
        if(detectedMarkers.rows()!=0)
        {
            for(int k=0; k<detectedMarkers.rows();k++)
            {

LandmarkData[detectedMarkers(k,0)]=std::make_pair(detectedMarkers(k,1)
,detectedMarkers(k,2));
            }
        }
        this->_flandmarkProxy->unsubscribe(strvalue);
    }
    AL::ALValue targetAnglesR = AL::ALValue::array(0);
    AL::ALValue targetTimesR = AL::ALValue::array(2.0f);
```

```
    this->_motionProxy->angleInterpolation(jointname, targetAnglesR,
targetTimesR, isAbsolute);
    return LandmarkData; //returns a map of detetcted markers
}
float Robot::detectObstacles()
{
    float minvalue;
    this->_sonarProxy->subscribe("MyApplication"); //subscribes to
sonar detectors.
    float lvalue=this->_memoryProxy-
>getData("Device/SubDeviceList/US/Left/Sensor/Value");
    float rvalue=this->_memoryProxy-
>getData("Device/SubDeviceList/US/Right/Sensor/Value");
    minvalue=min(lvalue,rvalue);
    return minvalue;
}

MatrixXf Robot::localize(Container &dlandmarks, Container &globalmap)
{
    //temporary holders for detected markers in robot frame
    MatrixXf Mr1=MatrixXf::Zero(1,3);
    MatrixXf Mr2=MatrixXf::Zero(1,3);
    //temporary holders for global landmark used in localization
    MatrixXf Mg1=MatrixXf::Zero(1,3);
    MatrixXf Mg2=MatrixXf::Zero(1,3);

    MatrixXf result(1,4), finalPose(1,3);
    MatrixXf totalPose=MatrixXf::Zero(1,3);

    // returns the estimated location & oreintation of the robot.

    float x1,y1, x2,y2;
    float X1,X2, Y1,Y2;
    int counter =0;

    //The following codes associates the landmarks data in robot frame
&  landmark data global frame.
    //and filters coplanar landmarks

    for(It lm(dlandmarks.begin()); lm!=dlandmarks.end();++lm)
    {
        for(rIt lms(dlandmarks.rbegin());
lms!=dlandmarks.rend();++lms)
        {
            if(*lms==*lm)
            {
                //breaks the loop if the iterators have same address
                //prevets repeating over global landmarks
                break;
            }
            It it1=globalmap.find(lm->first);
            It it2=globalmap.find(lms->first);
            if(it1!=it2)
            {
                if((it1->second.first!=it2->second.first) && (it1-
>second.second != it2->second.second))
                {
                    Mg1(0,0)=it1->first; Mg1(0,1)=it1->second.first;
Mg1(0,2)=it1->second.second;
                    Mr1(0,0)=lm->first; Mr1(0,1)=lm->second.first;
Mr1(0,2)=lm->second.second;
```

```
                    Mg2(0,0)=it2->first; Mg2(0,1)=it2->second.first;
Mg2(0,2)=it2->second.second;
                    Mr2(0,0)=lms->first; Mr2(0,1)=lms->second.first;
Mr2(0,2)=lms->second.second;

                    X1=Mg1(0,1), Y1=Mg1(0,2);
                    x1=Mr1(0,1), y1=Mr1(0,2);

                    X2=Mg2(0,1), Y2=Mg2(0,2); //global upper case
                    x2=Mr2(0,1), y2=Mr2(0,2); //local lower case

                    float A =-(X1*(x2-x1) + X2*(x1-x2) + Y1*(y2-y1) +
Y2*(y1-y2))/((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));

                    float B =-(y1*(X1-X2) + Y1*(x2-x1) + y2*(X2-X1) +
Y2*(x1-x2))/((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));

                    float Xo =-(x1*x2*X1 - X1*x2*x2 - x1*x1*X2 +
x1*x2*X2 -X2*y2*y2 + x2*y1*Y1 + X1*y1*y2 + X2*y1*y2 - x1*y2*Y1 -
X1*y2*y2 - x2*y1*Y2 + x1*y2*Y2)/
                            ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));

                    float Yo =-(-X1*x2*y1 + x2*y1*X2 +x1*x2*Y1 -
x2*x2*Y1 + x1*y2*X1 - x1*y2*X2 + y1*y2*Y1 - Y1*y2*y2 - x1*x1*Y2 +
x1*x2*Y2 - y1*y1*Y2 +y1*y2*Y2)/
                            ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));

                    result(0,0)=Xo;
                    result(0,1)=Yo;
                    result(0,2)=acos(A)*180.0/PI;
                    result(0,3)=asin(B)*180.0/PI;

                    //cout<<Mg1<< " G||R " <<Mr1<<endl;
                    //cout<<Mg2<< " G||R " <<Mr2<<endl;
                    cout<<"Localized : "<<result<<endl;

                    if(!(isnan(result(0,2))) && !(isnan(result(0,3)))
&&
(result(0,0)<2.0)&&(result(0,0)>0)&&(result(0,1)<2)&&(result(0,1)>0))
                    {
                        totalPose(0,0)+=result(0,0);
                        totalPose(0,1)+=result(0,1);
                        totalPose(0,2)+=result(0,2);

                        counter+=1;
                    }
                }
            }
        }
    }
    finalPose= totalPose/counter;
    return finalPose;
}

MatrixXf Robot::turn(float deltaphi)
{
    MatrixXf _temp(1,2);  //[turn_angle, target_heading,
heading_difference]
    cout<<"Difference in Orientation :"<<deltaphi<<endl;
    //determine the shortest turn angle for the robot...
    if(deltaphi>180){deltaphi = deltaphi-360;}
    if(deltaphi<-180){deltaphi = deltaphi+360;}
```

```
    cout<<"Shortest_Turn to target :"<<deltaphi<<" degrees"<<endl;
    float deltaphi2rad=deltaphi*PI/180;
    this->_motionProxy->moveTo(0,0,deltaphi2rad);
    _temp(0,0)=this->targetheading; //angle to turn
    _temp(0,1)=deltaphi;    //difference in heading.
    return _temp; //for use in the particle filter computations.
}

void Robot::redirectHeading(float &changeheading)
{
    this->_motionProxy->moveTo(0,0,changeheading);
    //turns  Nao around for localizaation
}

float Robot::getDistance(MatrixXf &startpose, float &target_x, float
&target_y)
{
    float dx,dy;
    dx=target_x-startpose(0,0);
    dy=target_y-startpose(0,1);
    return sqrt(dx*dx + dy*dy);
}

float Robot::getHeading(MatrixXf &startpose, float &target_x, float
&target_y)
{
    float dx,dy ,deltaphi;
    dx=target_x-startpose(0,0);
    dy=target_y-startpose(0,1);
    float target_phi=atan2(dy,dx); //determines the quadrant by the
sign.
    this->targetheading=target_phi*180/PI; //convert to degrees
    //all values in degrees...............
    deltaphi=targetheading - startpose(0,2);  //calculate difference
    return deltaphi;
}

bool Robot::connectToRobot(string ip_address, bool _state)
{
    bool conflag;
    //Check if we want to connect or disconnect
    if (_state)
    {   //Connect to the robot
        if (!this->_connected)
        {   try
            {
                this->_postureProxy=new
AL::ALRobotPostureProxy(ip_address,this->_portID);
                sleep(0.5);
                this->_postureProxy->goToPosture("Stand",1.0f);
                this->_motionProxy=new AL::ALMotionProxy(ip_address,
this->_portID);
                sleep(0.5);
                this->_navProxy=new
AL::ALNavigationProxy(ip_address,this->_portID);
                sleep(0.5);
                this->_flandmarkProxy=new
AL::ALLandMarkDetectionProxy(ip_address,this->_portID);
                sleep(0.5);
                this->_memoryProxy=new
AL::ALMemoryProxy(ip_address,this->_portID);
                sleep(0.5);
```

```
                this->_sonarProxy=new AL::ALSonarProxy(ip_address,
this->_portID);
                sleep(0.5);
                this->_speechProxy=new
AL::ALTextToSpeechProxy(ip_address, this->_portID);
                cout<< "Connection succesful!"<<endl;
                this->_ip_address=ip_address;
                this->_connected = true;
                conflag=true;
            }
            catch (const AL::ALError &e)
            {
                //An expection was caught. Print to the console what
it is.
                std::cout << "NaoMotionController: Caught Exception: "
<< e.what() << std::endl;
            }
        }
        return conflag;
    }

    if(!_state)
    {
        if (this->_connected)
        {
            //Turn motors off
            cout<<"Going to safe posture"<<endl;
            this->_postureProxy->goToPosture("Sit",1.0f);
            cout<<"turning motors off"<<endl;
            AL::ALValue value = AL::ALValue(0);
            this->_motionProxy->setStiffnesses("Body",value);
            cout<<"Motors Off: stiffness zero"<<endl;
            conflag=false;
        }
        return conflag;
    }
}

void Robot::init()
{
    string message="Nao is ready.";
    this->_motionProxy->moveInit();
    this->notify(message);
}

void Robot::setStiffness(float val)
{   //sets the whole body stiffness.
    AL::ALValue value = AL::ALValue(val);
    AL::ALValue pitchvalue=AL::ALValue::array(0.0, -0.3);
    //float fractionalSpeed=0.5f;
    std::string joints="Body";
    if (this->_connected)
    {
        cout<<"Setting Stiffness........"<<endl;
        this->_motionProxy->setStiffnesses(joints,value);
        //tilt head forward.
        this->_motionProxy->setAngles("HeadYaw",pitchvalue[0],0.5);
        this->_motionProxy->setAngles("HeadPitch",pitchvalue[1], 0.5);
    }
}

void Robot::notify(string &message)
```

```
{
    this->_speechProxy->say(message);
}
```

PARTICLE FILTER CLASS

```
#ifndef PARTICLEFILTER_H
#define PARTICLEFILTER_H

/** -----Eigen matrix headers------*/

#include <Eigen/Dense>
#include <Eigen/StdVector>

#include "datatypes.h"
#include "pose2d.h"
#include "transform.h"

#include <cstdlib>
#include <ctime>
#include <map>

#include <boost/random/uniform_int.hpp>
#include <boost/random/normal_distribution.hpp>
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/variate_generator.hpp>
#include <boost/random.hpp>

using namespace std;
using namespace PF;
using namespace Markers;
using  namespace Eigen;
using namespace dataobjects;

class ParticleFilter
{
public:
    ParticleFilter();
    ~ParticleFilter();

    typedef boost::mt19937 ENG;                       // Mersenne
Twister
    typedef boost::normal_distribution<double> DIST; // Normal
Distribution
    typedef boost::uniform_real<double>URealDist; //Uniform real
distribution
    typedef boost::variate_generator<ENG,DIST> NGEN;    // Variate
generator
    typedef boost::variate_generator<ENG,URealDist> RGEN;// Variate
generator
    typedef boost::uniform_int<int>INTDIST;    //Uniform Integer
Distribution
    typedef boost::variate_generator<ENG,INTDIST> INTGEN;//integer
variate generator;

    //member variables
    float PI;
    float sqrt2PI;
```

```
    float safedistance;
    float globalHeading;

    MatrixXf oldparticles;
    MatrixXf res_particles;
    MatrixXf new_particles;
    MatrixXf weightsX;

    int rangex, rangey, number_particles;

    MatrixXf createParticles(boost::random::mt19937 &rng,MatrixXf&
startpose, float &mapsize, int part_num);
    MatrixXf updateParticles(boost::random::mt19937 &rng, MatrixXf
&oldparticles,float &turnangle, MatrixXf displacement_x_y,float
&mapsize);
    PF::pfoutput resampleParticles(boost::random::mt19937
&rng,MatrixXf &res1_particles, MatrixXf &measuredPose,  float
&mapsize, Vector2f &meas_uncs);

    float gen_random_float(boost::random::mt19937 &rng, float min,
float max);
    int gen_random_int(boost::random::mt19937 &rng, int min, int max);
};

#endif // PARTICLEFILTER_H


#include "particlefilter.h"

ParticleFilter::ParticleFilter()
{
    PI=3.1415;
    sqrt2PI= 2.5066282746;
}

ParticleFilter::~ParticleFilter()
{

}

MatrixXf ParticleFilter::createParticles(boost::random::mt19937
&rng,MatrixXf &startpose, float &mapsize, int part_num)
{
    //create random particles within map area using the initial robot
location
    this->number_particles=part_num;
    this->oldparticles = MatrixXf::Zero(number_particles,4);
    this->res_particles = MatrixXf::Zero(number_particles,4);
    double mean=0.0, var=0.55;
    DIST dist(mean,var);
    NGEN  ngen(rng,dist);

    for(int i=0; i<number_particles; i++)
    {   //assigns weights to each particle.
        this->oldparticles(i,0)=1;  //weights
        float randn1=ngen();

        if((startpose(0,0)+randn1) > mapsize
||(startpose(0,0)+randn1)<0)
        {   this->oldparticles(i,1)=startpose(0,0);}
        else
```

```
        {   this-
>oldparticles(i,1)=startpose(0,0)+roundf(randn1*100)/100.0;}

        float randn2=ngen();
        if((startpose(0,1)+randn2 > mapsize) ||(startpose(0,1)+
randn2<0))
        {   this->oldparticles(i,2)=startpose(0,1);}
        else
        {   this->oldparticles(i,2)=startpose(0,1)+
roundf(randn2*100)/100.0;}

        int min=-2,max= 5;
        this->oldparticles(i,3)=startpose(0,2) +
gen_random_int(rng,min, max);
        if(oldparticles(i,3)>180)
        {
            oldparticles(i,3)=oldparticles(i,3) - 360;
        }
        if(oldparticles(i,3)<-180)
        {
            oldparticles(i,3)=oldparticles(i,3) + 360;
        }
   }

    // suppose i need to generate random numbers uniformly distributed
withing the map.
     double min=0, max=mapsize;
     URealDist uniformRealDistribution(min,max);
     RGEN urgen(rng,uniformRealDistribution);
    return this->oldparticles;
}

//Prediction phase
MatrixXf ParticleFilter::updateParticles(boost::random::mt19937 &rng,
MatrixXf &oldparticles, float &turnangle, MatrixXf displacement_x_y,
float &mapsize)
{
    double mean=0.0, var=0.05;
    int min=-2, max=5;
    DIST dist(mean,var);
    NGEN  genwalk(rng,dist);

    float turn_angle2rad = turnangle*(PI/180);
    float actualdistance=displacement_x_y(0,3) +
roundf(genwalk()*100)/100.0;

    //step 2: updates Nao(particle-pose) [each particle is update as
control + process noise ]
    for(int i=0; i <number_particles; i++)
    {
        //computes position and pose for each particle after walk
        res_particles(i,0)=oldparticles(i,0);
        res_particles(i,1)=oldparticles(i,1) + actualdistance *
cos(turn_angle2rad);
        res_particles(i,2)=oldparticles(i,2) + actualdistance *
sin(turn_angle2rad);
        //perfom same correction as done within change orientation for
all particles.....
        res_particles(i,3)=oldparticles(i,3) + turnangle +
gen_random_int(rng, min,max); //Process noise added due drift in walk.

        //ensures particle position is cyclic.
```

```
        if(res_particles(i,1)>2.0)
        {
            res_particles(i,1)=res_particles(i,1)-mapsize;
        }
        else if(res_particles(i,1)<0)
        {
            res_particles(i,1)=res_particles(i,1)+mapsize;
        }
        if(res_particles(i,2)>2.0)
        {
            res_particles(i,2)=res_particles(i,2)-mapsize;
        }
        else if(res_particles(i,2)<0){
            res_particles(i,2)=res_particles(i,2) +mapsize;
        }

        if(res_particles(i,3) >180)
        {   res_particles(i,3)=res_particles(i,3)-360;}
        if(res_particles(i,3)<-180)
        {   res_particles(i,3)=res_particles(i,3)+360;}
    }
    return this->res_particles;
}

//resampling after measurement
PF::pfoutput ParticleFilter::resampleParticles(boost::random::mt19937
&rng, MatrixXf &res1_particles, MatrixXf &measuredPose, float
&mapsize, Vector2f &meas_uncs)
{
    pfoutput tempOut;
    this->res_particles=res1_particles;
/*Step 3:
     - Computing probabilities for the cloud of particles
     - assuming that the robot has knowledged of all landmark
locations, using the res1_particles
     - the chosen weighting approach is to weight each particles based
on the difference in cartesian coordinates && orientation of the robot
pose measured from sensors.
   */
    VectorXf prob(number_particles),weights(number_particles);

    float dx, dy, dtheta;
    float psi_d = meas_uncs(0); //sensor noise due to drift during
walk
    float psi_theta =meas_uncs(1);//sensor noise due to overturn;

    for(int i=0; i<number_particles; i++){
        dx=measuredPose(0,0)-res1_particles(i,1);
        dy=measuredPose(0,1)-res1_particles(i,2);
        dtheta=measuredPose(0,2)-res1_particles(i,3);

        prob(i)=((1/(sqrt2PI*psi_d))*exp(- pow(dx,2)/(2*psi_d*psi_d)))
                *((1/(sqrt2PI*psi_d))*exp(-
pow(dy,2)/(2*psi_d*psi_d))) *
                  ((1/(sqrt2PI*psi_theta))*exp(-
pow(dtheta,2)/(2*psi_theta * psi_theta)));
    }
    this->res_particles.col(0)=prob; // assigns probabilities to
particles.

     //4.0 compute weights, cumsum(probabilities/sum(probabilities))
    weights(0)=prob(0);
```

```
    for(int j=1;j<number_particles; j++)
    {
        weights(j)=prob(j-1)+ prob(j);
    }
    //compute the total weight
    float total=0;
    for(int j=0; j<number_particles;j++)
    {
        total+=weights(j);
    }
    VectorXf norm_weights(number_particles);
    norm_weights= weights/total; //normalized weights

    float checksum=0;
    for(int i=0; i<number_particles; i++)
    {
        checksum+=norm_weights(i);
    }

    // 5.0 Sample particle cloud with replacement.
    this->new_particles=MatrixXf::Zero(number_particles,4);

    int index=gen_random_int(rng,0,number_particles);
        //select a random index from the number of particles
    float beta=0.0;
    //find the maximum weight
    float maxweight=0;
    for(int i=0;i<number_particles; i++)
    {
       if(norm_weights(i)>maxweight)
       {
           maxweight=norm_weights(i);
       }
    }

    for(int i=0;i<number_particles; i++)
    {
        beta+=(gen_random_float(rng,0.0, 1.0)*2*maxweight);
        while(beta>norm_weights(index))
        {
            beta -=norm_weights(index);
            index=(index+1)% number_particles;//ensures index never
overflows
        }
        this->new_particles.row(i)=this->res_particles.row(index);
        this->new_particles(i,0)=1; // resets weight to 1

    tempOut.new_particles=this->new_particles;
    tempOut.res_particles=this->res_particles;

    cout<<"\n\n"<<checksum<<endl;
    return tempOut;
}

float ParticleFilter::gen_random_float(boost::random::mt19937 &rng,
float min, float max)
{
    boost::uniform_real<float> u(min, max);
    boost::variate_generator<boost::mt19937&,
boost::uniform_real<float> > gen(rng, u);
    float temp=gen();
    return floor(temp*1000.0)/1000.0;
```

```
}

int ParticleFilter::gen_random_int(boost::random::mt19937 &rng, int
min, int max)
{
    boost::uniform_int<int> u(min, max);
    boost::variate_generator<boost::mt19937&, boost::uniform_int<int>
> gen(rng, u);
    return gen();
}
```

MAIN

```
/*
 * Copyright (c) 2012-2014 Aldebaran Robotics. All rights reserved.
 * Use of this source code is governed by a BSD-style license that can
be
 * found in the COPYING file.
 */
#include <iostream>
#include <fstream>
#include<cmath>
#include <map>
#include <Eigen/Dense>

#include "robot.h"
#include "particlefilter.h"
#include "datatypes.h"

using namespace std;
using namespace PF;
using namespace Eigen;

/*
 * Global Methods: required to ---
 * 1. Initialize Nao's Startpose.
 * 2. Set the target, define landamrk map.
 */

void init_input();
void init_commands();
float xpos,ypos,phi, target_x, target_y, target_phi;

float squareddiffx, squareddiffy;

//User defined type (Map for global landmarks).......
typedef std::map<int,std::pair<float,float> > Container;
typedef Container::const_iterator It;
typedef Container::const_reverse_iterator rIt;

typedef boost::mt19937  ENG;    // Mersenne Twister

#define PI 3.14159265359
/*
```

```
 * The main program includes the robot class, particle filter, the
map, and the path planning module
 *
 * 1. first the robot is connected and inititalized.....
 * 2. Then the particle filter is used to estimate the robot location
at each walk
 *
 *      Scheme:
 * I.   The target is specified. (x,y)  from which we compute also
heading
 *      relative to the origin.
 * II.  The start Pose (x,y, theta) is specified, next the difference
between robot (x,y, theta)
 *      and Target (Xt,Yt, Theta_t)

int main()
{
    static ENG engine(std::time(0));  //random engine seeded.
    srand(time(NULL));
    //Map of landmarks for localization contains landmark ID, x,y
cordinate of marker on map.
    //----------------------------------------------------------------
---------------------------------------------------
    Container globalmap, landmarkdata;
    globalmap[112] =std::make_pair(0.0,0.5);
    globalmap[117]=std::make_pair(0.0,1.0);
    globalmap[138]=std::make_pair(0.0,1.5);
    globalmap[108]=std::make_pair(0.0,1.9);
    globalmap[109]=std::make_pair(2.0,1.66);
    globalmap[131]=std::make_pair(2.0,1.18);
    globalmap[130]=std::make_pair(0.7,0.0);
    globalmap[125]=std::make_pair(2.0,0.9);
    globalmap[146]=std::make_pair(0.26,2.0);
    globalmap[80]=std::make_pair(0.6,2.0);
    globalmap[143]=std::make_pair(1.02,2.0);
    globalmap[124]=std::make_pair(1.8,2.0);
    globalmap[127]=std::make_pair(1.45,2.0);
    globalmap[170]=std::make_pair(2.0,0.55);
    globalmap[171]=std::make_pair(0.25,0.0);
    globalmap[141]=std::make_pair(1.25,0.0);
    globalmap[119]=std::make_pair(1.75,0.0);
    //----------------------------------------------------------------
--------------------------------------------------
    int num_particles=1000; float mapSize =2.0;
    Vector2f meas_uncs(2);
    meas_uncs<<3,2;

    //----------------------------------------------------------------
--------------------------------------------------
    MatrixXf startpose(1,3),endpose(1,3),measuredPose,
angles_turn_diff(1,3),odometry_endpose_disp_x_y(1,6);
    //1--------Robot & Particle filter--------------------------------
-------------------------------------------------
    Robot *NaoRobot= new Robot();
    ParticleFilter *NaoParticle=new ParticleFilter();
    //----------------------------------------------------------------
--------------------------------------------------
    MatrixXf oldparticles;
    PF::pfoutput particleFilterOutput;
    //1.0a--------  Robot connection variables------------------------
--
    string ip_address;
```

```
    bool connect=true, state=false;
    while(state==false)
    {
        cout<<" Enter Robot IP Address e.g: 127.000.0.001 \n"<<endl;
        cin>>ip_address;
        state=NaoRobot->connectToRobot(ip_address,connect);
    }

    cout<<"Robot is connected : Press 1 and any key to
disconnect"<<endl;
    int proceed;
    cin>>proceed;
    if(proceed!=1)
    {
        state=NaoRobot->connectToRobot(ip_address,!state);
        return 0;
    }

    //1.0b ---------------- creating csv file for datasets------------
-------
    ofstream datafile, lmsdetected,measlocation, odometrydata;

datafile.open("/home/uwaoma/Development/particle_results.csv",ios::out
|ios::app);

lmsdetected.open("/home/uwaoma/Development/landmarks_detected.csv",
ios::out|ios::app);
    measlocation.open("/home/uwaoma/Development/measuredlocation.csv",
ios::out|ios::app);
    odometrydata.open("/home/uwaoma/Development/odometerydata.csv",
ios::out|ios::app);
    if(datafile.is_open()){ datafile<<"Title: ,Old_Part, , , ,New_Par,
,  , Residual_Part,** ,\n";}
    if(lmsdetected.is_open()){
lmsdetected<<"detected,landmarks\n"<<"ID, x-pos, y-pos\n\n";}
    if(measlocation.is_open()){measlocation<<" Saves, the robot's,
position, as measured, by, robotcamera\n\n";}
    if(odometrydata.is_open()){odometrydata<< " Odometry,
[POSES]\n\n";}

    //1.1-------program initialization.-------------------------------
-------
    //Initial Simulation inputs: Note-:- all values for orientations
are returned in degrees.
    init_input();
    startpose(0,0)=xpos,startpose(0,1)=ypos,startpose(0,2)=phi;
    //filter input
    if(startpose(0,2)>180){ startpose(0,2)-=360;}
    if(startpose(0,2)<-180){ startpose(0,2)+=360;}

    oldparticles = MatrixXf::Zero(num_particles,4);
    MatrixXf inter_particles(num_particles,4);
    oldparticles = NaoParticle->createParticles(engine,startpose,
mapSize, num_particles);
    //get target
coordinates...............................................
    init_commands();  //specifies target cordinates in the
map.............

    float distance2Target=0, distancemoved=0;
    int count=0;
```

```
    NaoRobot->headstraight(); //positions nao head for proper view of
landmarks

    while(true)
    {
        count+=1;
    // 1.0---------------------------------------------------

        landmarkdata=NaoRobot->detectlandmark();
        measuredPose=NaoRobot->localize(landmarkdata,globalmap);
        cout<<"Visual EndPose : "<<measuredPose<<endl;
        measlocation<<measuredPose<<"\n\n";
        for(It iter(landmarkdata.begin());
iter!=landmarkdata.end();++iter)
        {
            lmsdetected<<iter->first<<","<<iter-
>second.first<<","<<iter->second.second<<",\n\n";
        }

        if(measuredPose(0,0)<0 || measuredPose(0,1)<0
||measuredPose(0,0)>2 ||measuredPose(0,1)>2)
        {
            cout<<"Cannot localize, Try again"<<endl;
            float newheading=PI;
            MatrixXf dummy=MatrixXf::Zero(1,6);
            NaoRobot->redirectHeading(newheading);
            landmarkdata=NaoRobot->detectlandmark();
            measuredPose=NaoRobot->localize(landmarkdata,globalmap);
            cout<<"Visual EndPose : "<<measuredPose<<endl;
            for(It iter(landmarkdata.begin());
iter!=landmarkdata.end();++iter)
  {
                lmsdetected<<iter->first<<","<<iter-
>second.first<<","<<iter->second.second<<","<<"Innerloop"<<",\n\n";
            }
  inter_particles=NaoParticle-
>updateParticles(engine,oldparticles,newheading,dummy,mapSize);
        }

        if(measuredPose(0,0)<0 || measuredPose(0,0)>2.0 ||
measuredPose(0,1)<0 || measuredPose(0,1)>2)
        {
            cin>>measuredPose(0,1);
            cin>>measuredPose(0,0);
            cin>>measuredPose(0,2);

            startpose=measuredPose; //note need to check that
startpose & measure pose are the same.
        }
        else
        {
            startpose=measuredPose;
        }


//first we check if we are at target or within a circle radius 15cm
around target.
        distance2Target= roundf(NaoRobot-
>getDistance(startpose,target_x,target_y)*10)/10.0;
        cout<<"\nDistance to target : "<<distance2Target<<endl;

        if(distance2Target<0.20)
```

```
        {//required to check if robot is already at goal.
            break;
        }
        else
        {
            //first calculate difference in heading and make shortest
turn to meet target direction
            //then turn the robot the requred angle of turn,

            float headingdiff=NaoRobot-
>getHeading(startpose,target_x,target_y);
            angles_turn_diff=NaoRobot->turn(headingdiff);

            odometry_endpose_disp_x_y=NaoRobot-
>move(distance2Target,angles_turn_diff,startpose);

distancemoved=roundf(odometry_endpose_disp_x_y(0,3)*100)/100;
            cout<<"distancemoved  : "<<distancemoved<<endl;

            endpose=odometry_endpose_disp_x_y.block<1,3>(0,0);
            cout<<"Odometry endppose"<<endpose<<endl;//----------------
-----------1
            //particle updates
            inter_particles=NaoParticle-
>updateParticles(engine,oldparticles,angles_turn_diff(0,1),odometry_en
dpose_disp_x_y,mapSize);
            // after walk relocalize
            landmarkdata=NaoRobot->detectlandmark();
            for(It iter(landmarkdata.begin());
iter!=landmarkdata.end();++iter)
            {
                lmsdetected<<iter->first<<","<<iter-
>second.first<<","<<iter->second.second<<",\n\n";
            }
            measuredPose=NaoRobot->localize(landmarkdata,globalmap);
            /* assumming that the localization work perfectly
            if(measuredPose(0,0)<0 || measuredPose(0,1)<0
||measuredPose(0,0)>2 ||measuredPose(0,1)>2)
            {
                cout<<"Cannot localize, Try again"<<endl;
                float newheading=PI;
                NaoRobot->redirectHeading(newheading);
                landmarkdata=NaoRobot->detectlandmark();
                measuredPose=NaoRobot-
>localize(landmarkdata,globalmap);
            }
            */

            //then we resample particles and write output to file.
            particleFilterOutput=NaoParticle-
>resampleParticles(engine,inter_particles,measuredPose,mapSize,meas_un
cs);
            //stores and reassigns particles..............
            datafile <<count<<","<<"Old X,  Old Y , Old_Theta, [- | -]
, New_X , New_Y, New_Theta, [- | -] , Res_X, Res_Y, Res_Theta,\n";
            for(int i=0; i<num_particles;i++)
                {
                    datafile<<","<<
oldparticles(i,1)<<","<<oldparticles(i,2)<<","<<oldparticles(i,3)<<",
[- | -],"<<
```

```
particleFilterOutput.new_particles(i,1)<<","<<particleFilterOutput.new
_particles(i,2)<<","<<particleFilterOutput.new_particles(i,3)<<
                              ",  [- | -
],"<<particleFilterOutput.res_particles(i,1)<<","<<particleFilterOutpu
t.res_particles(i,2)<<","<<particleFilterOutput.res_particles(i,3)<<"\
n";
                   }
              datafile<<"\n, nextIteration \n\n";
              oldparticles=particleFilterOutput.new_particles;

   //-2.0 -------------------------------------------------------
         startpose=measuredPose; //localization assumed correct......
         cout<<"Localized @ : [ "<<measuredPose<<" ]"<<endl;

odometrydata<<endpose(0,0)<<","<<endpose(0,1)<<","<<endpose(0,2)<<"\n\
n";
     }

     string message="Robot is at target!";
     NaoRobot->notify(message);
     //filestream close operations
     datafile.close();
     lmsdetected.close();
     measlocation.close();
     odometrydata.close();

     int endrun=0;
     cout<<"Task is complete : Enter 1 to stop robot: "<<endl;
     cin>>endrun;
     if(endrun!=0)
     {
         //set robot in safe posture and disconnects
         NaoRobot->connectToRobot(ip_address,!state);
     }
   return 0;
}


void init_input()
{    //Takes the start pose
     cout<<"\nEnter the initial pose (range : 0 - 200): x-cm, y-cm &
theta-deg. (-180<=theta<=180)\n"<<endl;
     int x,y, angle;
     cin>>x>>y>>angle;
     xpos=fabs(x%200);
     ypos=fabs(y%200);
     xpos=xpos/100;
     ypos=ypos/100;

     phi=angle%360;
     cout<<"Startpose: ["<<xpos<< ","<<ypos<<","<<phi<<"]"<<endl;
}


void init_commands()
{
     cout<<"Specify Target: Position [x ,y ] Range [0 < x,y < 200cm
]"<<endl;
     int t_x,t_y;
     cin>>t_x>>t_y;
     target_x=fabs(t_x%200);
```

```
    target_y=fabs(t_y%200);
    target_x=target_x/100; //convert to meters.
    target_y=target_y/100;
    //computes target direction;
    target_phi=atan2(target_y,target_x) *180/PI;
    cout<<"\nTarget Specified:[ "<<target_x<<" "<<target_y<<"
"<<target_phi<<"]"<<endl;
}
```