Robot Navigation Using Velocity Potential Fields and Particle Filters for Obstacle Avoidance

Jin Bai

A thesis submitted to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of
MASTER OF APPLIED SCIENCE
in Mechanical Engineering

uOttawa

Ottawa-Carleton Institute for Mechanical and Aerospace Engineering
University of Ottawa
Ottawa, Canada
July, 2015

Abstract

In this thesis, robot navigation using the Particle Filter based FastSLAM approach for obstacle avoidance derived from a modified Velocity Potential Field method was investigated. A switching controller was developed to deal with robot's efficient turning direction when close to obstacles. The determination of the efficient turning direction is based on the local map robot derived from its on board local sensing. The estimation of local map and robot path was implemented using the FastSLAM approach. A particle filter was utilized to obtain estimated robot path and obstacles (local map). When robot sensed only obstacles, the estimated robot positions was regarding to obstacles based the measurement of the distance between the robot and obstacles. When the robot detected the goal, estimation of robot path will switch to estimation with regard to the goal in order to obtain better estimated robot positions. Both simulation and experimental results illustrated that estimation with regard to the goal performs better than estimation regarding only to obstacles, because when robot travelled close to the goal, the residual error between estimated robot path and the ideal robot path becomes monotonously decreasing. When robot reached the goal, the estimated robot position and the ideal robot position converge. We investigated our proposed approach in two typical robot navigation scenarios. Simulations were accomplished using MATLAB, and experiments were conducted with the help of both MATLAB and LabVIEW. In simulations and experiments, the robot successfully chose efficiently turning direction to avoid obstacles and finally reached the goal.

Acknowledgements

I would like to express my sincerest gratitude to my supervisor Dr. Dan Necsulescu and my co-supervisor Dr. Jurek Sasiadek. You gave me encourage and support when I met troubles and had no clues about what I was doing. During the journey to my destination, I was lost for some time and it was you who woke me up and instructed me a vivid and profound lesson. I cannot complete my thesis without your guidance and patience.

Also I would like to thank my colleagues and my friends for their inspiration, encouragement and advice, some of which really matters to me. Thank you Guangqi, Qingfeng and Yu.

Finally, I would like to thank my family, especially my parents, for their love, understanding and supports, and to Miss Duan, for her love, support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background Information

Mobile robots are used in industry, military and field applications. They are designed and developed to search and explore areas, to perform certain tasks. Their application areas are various such as: inspection, mining, intelligent transportation, and military applications [1].

Based on the environment in which they travel, mobile robots are classified as aerial robots, ground robots, underwater robots, etc. Depending on the device they use to move, ground robots are legged robots or wheeled robots [1]. Autonomous robot used for our experiments is presented in detail. As can be seen, from the name "autonomous", are robots with a high level of autonomy and are capable of intelligent motion and action without requiring either a wireless guide to or teleoperator control [2].

1.1.1 Autonomous Mobile Robot

Among autonomous mobile robots, ground robots play an important role. Ground robots, which are also called Unmanned Ground Vehicles (UGVs) were first developed in 1930s. USSR developed Teletanks, a machine gun-armed tank remotely controlled by radio from another tank, but the combination of cost, low speed, reliance on a cable for control, and poor protection against weapons resulted that it was not considered a success [3]. Nowadays, Unmanned Ground Vehicles (UGVs), including teleoperated and autonomous vehicles (mobile robots), have been used in military and civil applications [4], and can be found in toys stores, exhibitions, even in the school labs. Research and development of UGVs has been dominated by DARPA (Defense Advanced Research Projects Agency) and NASA (National Aeronautics and Space Administration). The DARPA initiative started with the development of the first mobile robot, Shakey, and also includes the Autonomous Land Vehicle and the

DARPA Demo I1 Program. NASA sponsors the development of unmanned vehicles for planetary surface exploration, from the Jet Propulsion Laboratory Mars Rover to the most recent Mars Pathfinder. Recent UGV design and development has been enhanced to build UGVs capable of operating in Intelligent Vehicle Highway Systems [5].

Engineers and researchers design autonomous mobile robot systems based on various levels of autonomy and how they interact with the human operator. The decision on the level of autonomy depends on the scope of the work and the budget of the project [46]. Fully autonomous systems are more complex and have expensive solutions. In general there are four levels of autonomy for unmanned vehicle systems [46]:

- Remote control and teleoperation.
- The semi-autonomous mobile robot.
- Platform-centric autonomous.
- Network-centric autonomous.

In the first level of remote control and teleoperation, humans manipulate robots from a distance. The operator can visualize the location and movement of the platform through imbedding all the cognitive processes and onboard sensors into the platform. The onboard effectors enable the human to react on the data it provides.

The second level is about the semi-autonomous mobile robot. In this case, the robot is more complex. The system with advanced navigation, obstacle avoidance and data fusion capabilities would minimize operator interaction with the platform.

The third level is called platform-centric autonomous. Robots at this level are fully autonomous so that they can engage in complex missions, and react directly from a controller without any guidance from an operator.

The last level, network-centric autonomous, will make robots receive information from a network and incorporate it online in the mission processing. They must respond to appropriate information requests and action commands received from the network, including resolution of conflicts when request or commands, interference with each other or with the original mission assigned to the UGV [46].

1.1.2 Motion Planning

In automatic motion planning lie certain challenges about autonomous mobile robots. Motion planning algorithms are derived from control theory, computational and differential geometry and computer science [46]. Generally, there is no single solution to solve motion planning problems, and controllers are usually derived from combinations of different methods from different fields of engineering and computer science.

As shown in [61], motion planning is composed of four main components: navigation, coverage, localization and mapping. Firstly, navigation is focusing on the process of controlling robot systems from one configuration to another. Secondly, coverage is the problem of passing a sensor or tool over all points in space and determining if all points are available to the robot. Thirdly, localization is the problem of determining where the robot is in the whole environment. And finally, mapping problem is about building a map of unknown environment based on the data collected from sensors.

Considering the third component—localization— and the fourth component—mapping — together and simultaneously they form an algorithm called SLAM (Simultaneous Localization and Mapping). Our research is mainly to advance in this field, which will be elucidated later.

Properties of the robot make an important effect on motion planner. For instance, once understanding the robot configuration space, we should consider whether robot can move in any direction instantaneously in this space. An omnidirectional mobile robot can travel in any direction in its configuration space; however, in practice, it is not convenient to use an omnidirectional mobile robot and it may cost too much. Wheeled mobile robots with nonholonomic constraints prevail in the mobile robots applications since that are easy to handle and control. They are constrained in sideway translation movements in their configuration space. Moreover, wheeled mobile robots can be modeled using kinematic and dynamic equations with velocity vectors or force vectors as controls.

The basic strategy in developing motion planner to regulate the motion of mobile robots [46]:

- Sense
- Map
- Plan

- Act

In sensing step, information regarding to the local environment is obtained by sensors and the robot pose is driven by controllers. Then in the step of mapping, mapping algorithms take in sensor data and develop a two-dimensional map of the terrain determining the location of obstacles and walls. The third step is planning, or path planning. Path-planning algorithms will give a robot path, velocity profile along with the path based on the map building, sensor data and the nonholonomic constraints of the robot. Finally, robot converts velocity data obtained in last step to the commands to the robot motors of the differential drive mobile robot.

1.1.3 Simultaneous Localization and Mapping

In robotics, Simultaneous Localization And Mapping (SLAM) is the computational approach of constructing or updating a map of an unknown environment while simultaneously keeping track of robot location within it. This problem initially appears to be a "chicken-and-egg problem". There are several algorithms known for solving it among which efficient solution methods include the particle filter and extended Kalman filter. [62]

SLAM approaches are employed in self-driving cars, unmanned aerial vehicles, autonomous underwater vehicles, planetary rovers, newly emerging domestic robots and even inside the human body [63]. In other words, the problem of building a map of an unknown environment from a series of landmark measurements obtained from a moving robot is addressed in SLAM [34]. As mentioned in Section 1.1.1, for a truly autonomous robot, a key prerequisite is considered by many the ability to simultaneously localize a robot and accurately map its surrounding environment [64].

The dominant approach to the SLAM problem is the use of the extended Kalman filter and the particle filter. In [18] was proposed an approach using the extended Kalman filter for estimating the posterior distribution about robot pose along with the positions of the landmarks. This approach prevails for the last decade. Recent research has changed it to scaling to much larger environments, with more than a few hundred landmarks [26, 27, 32]. Some papers focused on modifying this approach to algorithms for handling data association problems [33].

Computational complexity limits certain approaches such as the EKF-based SLAM approaches mentioned here. Sensor updates in this approach require time is quadratic in the

number of landmarks K, and this computational complexity is derived from the covariance matrix maintained by the Kalman filters has $O(K^2)$ elements [34]. Hence, if the natural environment contains millions of features, this approach would be hard to deal with it. This issue has been mentioned in [26, 27, 48].

In order to minimize the computational complexity of EKF-based approaches and obtain easy handling results, the FastSLAM approach was developed as a factored solution to the SLAM problem by M. Montemerlo, S. Thrun from Carnegie Mellon University and D. Koller, B. Wegbreit from Stanford University. The FastSLAM is approached from a Bayesian point of view.

This factored solution is decomposing the SLAM problem into a robot localization problem, and a bunch of landmark estimation problems conditioned on the robot pose estimation. Reference [49] indicates that this factored representation is accurate because of the natural conditional independences in the SLAM problem. For robot path estimation, FastSLAM utilizes a particle filter to obtain the posterior distribution over robot paths, whereas for landmark location estimation, it is based on K Kalman filters of each particle in order to estimate the K landmark locations conditioned on estimated robot path.

1.2 Research Objective and Approach

Robot navigation draws plenty of attention in the mobile robotics literature. In our research, robot navigation algorithms were investigated. We assumed that the robot has a goal to travel towards and that in between its path towards the goal lie some obstacles with distinct dimensions. The robot may detect obstacles when its onboard sensors find them. At this time, robot will build its surrounding environment using data obtained by its bearing sensors. When robot goes too close to obstacles, it has to find a way avoiding them. The problem is that the robot can choose to turn left or right in order to avoid obstacles in front of it. The question is then in which direction robot is supposed to choose? The local map robot built can deal with this issue. Assuming obstacle in left front of the robot has a larger dimension (for example, a massive rock), whereas obstacle in right front of the robot is a bit smaller (such as a small wood block), then robot will choose to turn right since in that direction robot will avoid obstacles with efficiency and saving time.

For robot navigation with obstacles avoidance, a switching controller is proposed. The algorithm is a modified velocity potential field approach. In this approach, the robot will behave as if an attractive velocity with regard to the goal and a repulsive velocity corresponding to obstacle are applied. The robot will be attracted to the goal while turning away based on a switching controller when going too close to obstacles.

There are however noises in robot navigation process. When the robot sensed obstacle, the positions of obstacles can be obtained based on robot positions. But the noise effect has to be accounted for. For instance, if given a command for moving the robot one meter ahead, the robot might just go to a point further away in practice. This noise is called system noise. Likewise, when robot's bearing sensor detected obstacles, the measurements we obtained are also not accurate due to the measurement noise.

Because of the noise, we need to obtain optimal estimations from these noisy variables. This technique is called the optimal state estimation. There are various methods to get state estimation. In our research, we found the particle filter method suitable and accessible to our issues. Particularly, in robot navigation process with obstacle avoidance, estimations with regard to robot path and locations of obstacles are needed to be obtained. This approach, regarding to robot path estimation and obstacle location estimation, was developed is called SLAM (Simultaneously Localization and Mapping).

The problem of SLAM can be solved several ways. The most popular solutions are EKF (the extended Kalman filter) SLAM, UKF (the unscented Kalman filter) SLAM and the FastSLAM. In this thesis, the FastSLAM (version 1.0) approach with the help of particle filter method is used to obtain estimation wrt robot path and obstacles. The reason we choose the FastSLAM approach other than EKF SLAM and UKF SLAM is that the FastSLAM performs better in a highly nonlinear system and it works with not only white noise but colored noise. We did simulations about two scenarios using the FastSLAM approach with obstacles avoidance.

In our research, we will focus on the mobile ground robots for the experiments, specifically the wheeled mobile robot. Our robot is equipped with three wheels, in which the back one is a castor wheel that is able to spin in 360 degrees and the front two are based on differential drive steering approach. Both front wheels are controlled independently and can be coordinated by two distinct speeds drivers so that the robot can perform various tasks such as spinning in place,

moving in a straight line and on a circle. However, our robot cannot move sideways due to non-holonomic kinematic constrains, which means we will not give it tasks like moving in a direction perpendicular to the front wheels. Thus, when the robot meet obstacles, it will not move sideways. Instead it is spinning and moving on a smooth path around the obstacles.

1.3 Thesis Outline

This thesis consists of eight chapters that consist in the application of the FastSLAM approach in robot navigation with obstacle avoidance combined with Velocity Potential Field approach.

Chapter 2 provides a literature review regarding various obstacle avoidance techniques and solutions regarding to the problem of SLAM.

Chapter 3 presents robot navigation using a switching controller with obstacle avoidance based on a modified velocity potential field method.

Chapter 4 details the FastSLAM approach applied in robot navigation with obstacle avoidance.

Chapter 5 presents the experimental setup for our research and functions of all experimental hardware components.

Chapter 6 illustrates all of our simulation results implemented in MATLAB with two distinct scenarios using our proposed algorithm.

Chapter 7 includes experimental results corresponding to all our simulations presented by using both LabVIEW and MATLAB.

Finally, Chapter 8 ends with a conclusion and presents our research contributions, and discusses future work that might be done.

Chapter 2
Literature Review

2.1 Obstacle Avoidance Techniques

In the past, extensive work has been carried out for mobile robots trajectory planning for both holonomic and non-holonomic vehicles. Virtual Force Field and Artificial Potential Field approaches were used for planning a trajectory which avoids collisions with known obstacles [6, 7, 8]. These approaches were developed from the formulation of potential energy for elastic materials. Vision based obstacle avoidance techniques were addressed since vision is one of the most powerful sensing methods used for autonomous navigation of robots. Compared with other on-board sensing techniques, vision based approaches to navigation continue to demand a lot of attention from the mobile robot research community [9].

There were certain researches with regard to Non-Rigid Obstacle Avoidance for Mobile Robots, in which a classification of obstacles was considered. Most researches about obstacle avoidance for mobile robots are regarding rigid obstacle, either static or moving. In [9], non-rigid obstacle was considered which has been less studied before.

In an unknown clustered environment, reactive navigation joint operation between sensory data and commands could be defined. Providing that such an operation can come from a family of predefined functions like potential fields methods or it can be built using 'universal' approximation such as fuzzy logic [10].

2.1.1 Virtual Force Field Approach

This method used a force-field approach in which obstacles have an applied virtual repulsive forces to the robot, while the target has an applied a virtual attractive force [11]. Force-field-based obstacle avoidance methods had been developed by Khatib (1985), Krogh (1984) and Krogh and Thorpe (1986). These approaches are difficult to use for real-time

navigation. Brooks (1986) and Arkin (1989) applied force-field methods on experimental mobile robots equipped with ultrasonic sensors, but the results are prone to sensor errors.

Methods for constructing a kind of *Safe Zone* or *Free Zone* to avoid a robot from collisions with obstacles and other robots have been investigated by researchers [12]. Masoud proposed a repulsive field that is surrounding the robot to protect it from collision, in which this repulsive field is generated as the gradient flow of a spherically symmetric potential field [13]. An approach for real-time manipulator collision avoidance was investigated by Seraji and Bon [14]. In this method, a *Safe Zone* is defined for each obstacle and, when a manipulator enters this safe zone, it will suffer from virtual repulsive force, indicated as a virtual spring-damper model. The closer the manipulator is form the obstacle, the larger the resulting repulsive force.

2.1.2 Artificial Potential Field Approach

Artificial Potential Field (APF) method provides effective and simple motion planners for practical purpose and has become a very popular and basic method for obstacle avoidance. Khatib was the first to develop this method for obstacle avoidance [16]. The APF approaches consist basically in a gradient descent search method which is directed toward minimizing the potential function [15]. There are two fields surround both obstacles and the goal point. Repulsive potential fields are around obstacles which have to be avoided and attractive potential fields surround the goal point. The attractive potential is generally a bowl shaped energy well which drives a robot to its center if the environment is free of obstacles. In an obstructed condition, a repulsive potential energy hill to repel the robot is added to the attractive potential field at the locations of the obstacles [15]. The robot experiences the force that equals to the negative gradient of the potential. This force drives the robot downhill until the robot reaches the minimum energy point.

2.1.3 Vision Based Obstacle Avoidance

The problem of controlling a vehicle based on the image data has been considered before in various contexts. Horswill (1993) proposed a simple but effective vision-based behavior which could be used to guide a robot through an obstacle filled environment safely [19]. Dickmanns et al investigated the problem of controlling a motor vehicle depended on the

information obtained from conventional cameras mounted onboard [20, 21].

In [17], the authors addressed the development of intelligent robot systems by composing simple building blocks in a *bottom-up approach*. The building blocks consist of controllers and estimators, and the framework for composition allows for tightly coupled perception-action loops. Eventually, they describe a framework for cooperative control of a group of nonholonomic mobile robots that allows us to build complex systems from simple controllers and estimators. In this approach, an omnidirectional camera is used as a single type of sensor for all our controllers. They describe estimators that abstract the sensory information at different levels, enabling both decentralized and centralized cooperative control.

2.1.4 Non-Rigid Obstacle Avoidance

Researchers have developed a variety of obstacle avoidance algorithms, and these algorithms correspond to different kinds of obstacles. The types of obstacles were classified in [9]. Obstacles are static or moving. Most strategies focus on generating immediate reaction to environments. In [9], they proposed a distinct classification of obstacles from a different standpoint, which is shown in Table 2.1, and introduced a concept of *non-rigid obstacles*.

Classification of obstacles	Shape of obstacle	Position of obstacle
Non-rigid obstacle	Changing	——
Moving rigid obstacle	Constant	Changing
Static rigid obstacle	Constant	Constant

Table 2.1 Classification of obstacles.

Non-rigid obstacles are defined as an abstract object whose boundary is changeable. To deal with these obstacles, the agent's visibility and a possible observer play a role for smart navigation. Avoidance algorithms for non-rigid obstacle can be classified into four categories: reactive method, grid-based method, randomization method, and path-velocity decomposed (PVD) method [9]. Combining these four strategies could deal with non-rigid obstacle

avoidance effectively.

2.1.5 Fuzzy Logic Control for Obstacle Avoidance

This approach is usually applied in case of an unknown dynamic environment. The fuzzy logic control does not need a mathematical model of the controlled process. This will permit controlling vehicles with various structures [22]. In some cases, two fuzzy logic controllers, which use distinct membership functions, are applied to mobile robot navigation in an unknown environment. One cab a Tracking Fuzzy Logic Controller (TFLC), which is employed to navigate the mobile robot to its goal, and the other is an Obstacles Avoiding Fuzzy Logic Controller (OAFLC), which is proposed to perform the obstacle avoidance behavior.

In [23], a fuzzy controller is used to control an obstacle avoidance mobile robot. This controller uses three sub-controllers. The outputs are summed to produce a concerted effort to control the motors, steering the robot away from obstacles. As the robot moves, it is continuously monitoring for obstacles on its left, front and right side. There are two sensors on each side of the robot. When the robot moves, the data from its sensors change due to the unknown dynamic environment. With such multitude of input data, a hierarchical structured controller is designed.

2.2 Simultaneous Localization and Mapping Techniques

The problem of SLAM (Simultaneous Localization and Mapping) has attracted significant attention in the mobile robotics literature. SLAM is mainly regarding the problem of building a map of an environment from a sequence of landmark measurements obtained from a moving robot. The name SLAM is derived from the idea that the mapping problem necessarily induces a robot localization problem since robot motion is subject to error. The ability to simultaneously localize a robot and accurately map its environment has been considered by many researchers to be a key prerequisite of truly autonomous robots. [34]

In [18], the dominant approach to the SLAM problem was introduced. This paper proposed using the Extended Kalman Filter (EKF) for incrementally estimating the posterior distribution over robot pose along with the positions of landmarks. This approach has been widely accepted

in the field robotics in the last decade.

EKF SLAM is the algorithms using the extended Kalman filter to solve SLAM. Typically the algorithms are based on the maximum likelihood algorithm for data association. For the past decade, the EKF SLAM has been the popular method for SLAM, until the introduction of the FastSLAM approach. M. Montemerlo et. all proposed the FastSLAM approach firstly in proceedings of the AAAI National Conference on Artificial Intelligence. [24]

2.2.1 EKF SLAM Techniques

In [25], the SLAM problem refers to the possibility for an autonomous vehicle to start in an unknown location in an unknown environment and then to incrementally build a map of this environment while simultaneously using this map to compute absolute vehicle location. This paper proves that a solution to the SLAM problem is indeed possible. The underlying structure of the SLAM problem is first elucidated. A proof that the estimated map converges monotonically to a relative map with zero uncertainty is then developed. It is then shown that the absolute accuracy of the map and the vehicle location reach a lower bound defined only by the initial uncertain position of the vehicle. These results show that it is possible for an autonomous vehicle to start in an unknown location and in an unknown environment and, using relative observations only, incrementally build a map of the world and simultaneously to compute a bounded estimate of vehicle location.

Paper [26] addresses real time implementation of the Simultaneous Localization and Map Building (SLAM) algorithm. It presents optimal algorithms that consider the special form of the matrices and a new compressed filter that can significantly reduce the computation requirements when working in local areas or with high frequency external sensors. It is shown that by extending the standard Kalman filter models the information gained in a local area can be maintained with a cost~O(N_a^2), where N_a is the number of landmarks in the local area, and then transferred to the overall map in only one iteration at full SLAM computational cost. Additional simplifications are also presented that are very close to optimal when an appropriate map representation is used. Finally the algorithms are validated with experimental results obtained with a standard vehicle running in a completely unstructured outdoor environment.

In [27], decoupled stochastic mapping (DSM) as a computationally efficient approach to

large-scale concurrent mapping and localization is proposed. DSM reduces the computational burden of conventional stochastic mapping by dividing the environment into multiple overlapping submap regions, each with its own stochastic map. Two new approximation techniques are utilized for transferring vehicle state information from one submap to another, yielding a constant-time algorithm whose memory requirements scale linearly with the size of the operating area. The performance of two different variations of the algorithm is demonstrated through simulations of environments with 110 and 1200 features. Experimental results are presented for an environment with 93 features using sonar data obtained in a 3 by 9 by 1 meter testing tank.

The paper [32] studied the problem of consistent registration of multiple frames of measurements (range scans), together with the related issues of representation and manipulation of spatial uncertainties. The approach maintains all the local frames of data as well as the relative spatial relationships between local frames. These spatial relationships are modeled as random variables and are derived from matching pairwise scans or from odometry. Then it formulates a procedure based on the maximum likelihood criterion to optimally combine all the spatial relations. Consistency is achieved by using all the spatial relations as constraints to solve for the data frame poses simultaneously.

Paper [33] addresses the problem of building large-scale geometric maps of indoor environments with mobile robots. It poses the map building problem as a constrained, probabilistic maximum-likelihood estimation problem. It then devises a practical algorithm for generating the most likely map from data, along with the most likely path taken by the robot. Experimental results in cyclic environments of size up to 80 by 25 meter illustrate the appropriateness of the approach.

In [48], the structure of the SLAM problem is elucidated and this is achieved by the analysis of a conventional and well known SLAM algorithm using global coordinates called, in this case, the Absolute Map Filter or AMF. Using this algorithm, three convergence theorems central to the SLAM problem are proved for the first time. They prove that the uncertainty in the estimated map decreases monotonically and achieves a defined lower bound. Furthermore, in the limit, as the number of landmark observations increases, the relationship between landmarks becomes perfectly known. These proofs constitute the second theoretical contribution [25]. The third principal theoretical contribution is the development of a novel

SLAM solution capable of solving the SLAM problem in real time. This algorithm is called the Geometric Projection Filter or GPF. Rather than estimate the location of landmarks in global coordinates it estimates the relationships between individual landmarks. The convergence properties of this algorithm are derived and compared with those of the conventional AMF algorithm.

An implementation of the GPF and the AMF is also provided for a custom built subsea vehicle. The performance of the two filters are compared and shown to have the properties predicted by the preceding theoretical analysis. This implementation constitutes the fourth principal contribution. It shows that the GPF can be used as the basis of a substantive real time deployment of a mobile robot in an initially unknown environment. [48]

2.2.2 SLAM Techniques Based on Particle Filtering Algorithm

In [49] is considered the problem of learning a grid-based map using a robot with noisy sensors and actuators. Two approaches are compared: online EM, where the map is treated as a fixed parameter, and Bayesian inference, where the map is a (matrix-valued) random variable. It shows that even on a very simple example, online EM can get stuck in local minima, which causes the robot to get "lost" and the resulting map to be useless. By contrast, the Bayesian approach, by maintaining multiple hypotheses, is much more robust. Then a method for approximating the Bayesian solution, called Rao-Blackwellised particle filtering is introduced. This approximation is shown fast and accurate when coupled with an active learning strategy.

In [50] is presented an overview of methods for sequential simulation from posterior distributions. These methods are of particular interest in Bayesian filtering for discrete time dynamic models that are typically nonlinear and non-Gaussian. A general importance sampling framework is developed that unifies many of the methods which have been proposed over the last few decades in several different scientific disciplines. Novel extensions to the existing methods are also proposed. Is shown in particular how to incorporate local linearization methods similar to those which have previously been employed in the deterministic filtering literature; these lead to very effective importance distributions. Furthermore it describes a method which uses Rao-Blackwellisation in order to take advantage of the analytic structure present in some important classes of state-space models. In a final section algorithms for

prediction, smoothing and evaluation of the likelihood in dynamic models are developed.

The paper [51] presents a new algorithm for mobile robot localization, called Monte Carlo Localization (MCL). MCL is a version of Markov localization, a family of probabilistic approaches that have recently been applied with great practical success. Previous approaches were either computationally cumbersome (such as grid-based approaches that represent the state space by high-resolution 3D grids), or had to resort to extremely coarse-grained resolutions. The approach proposed is computationally efficient while retaining the ability to represent (almost) arbitrary distributions. MCL applies sampling-based methods for approximating probability distributions, in a way that places computation "where needed". The number of samples is adapted on-line, thereby invoking large sample sets only when necessary. Empirical results illustrate that MCL yields improved accuracy while requiring an order of magnitude less computation when compared to previous approaches. It is also much easier to implement.

Paper [52] introduced the definition of localization as the problem of determining the position of a mobile robot from sensor data. Most existing localization approaches are passive, i.e., they do not exploit the opportunity to control the robot's effectors during localization. This paper proposes an active localization approach. The approach is based on Markov localization and provides rational criteria for (1) setting the robot's motion direction (exploration), and (2) determining the pointing direction of the sensors so as to most efficiently localize the robot. Furthermore, it is able to deal with noisy sensors and approximate world models. The appropriateness of the approach is demonstrated empirically using a mobile robot in a structured office environment.

In [53], Hidden Markov models (HMMs) have proven to be one of the most successfully used tools for probabilistic models for time series data. In an HMM, information about the past is conveyed through a single discrete variable—the hidden state. It is discussed that a generalization of HMMs, in which this state is factored into multiple state variables and is therefore represented in a distributed manner. It shows that it is an exact algorithm for inferring the posterior probabilities of the hidden state variables given the observations, and relates it to the forward-backward algorithm for HMMs and to algorithms for more general graphical models. Due to the combinatorial nature of the hidden state representation, this exact algorithm is intractable. As in other intractable systems, approximate inference can be carried out using

Gibbs sampling or variational methods. Within the variational framework, it is presented that a structured approximation in which the state variables are decoupled, yielding a tractable algorithm for learning the parameters of the model. Empirical comparisons suggest that these approximations are efficient and provide accurate alternatives to the exact methods. Finally, the structured approximation to model Bach's chorales is used and factorial HMMs can capture statistical structure in this data set which an unconstrained HMM cannot is shown.

In [54], navigation methods for office delivery robots need to take various sources of uncertainty into account in order to get robust performance. In previous work, it was developed a reliable navigation technique that uses partially observable Markov models to represent metric, actuator, and sensor uncertainties. The paper describes an algorithm that adjusts the probabilities of the initial Markov model by passively observing the robot's interactions with its environment. The learned probabilities more accurately reflect the actual uncertainties in the environment, which ultimately leads to improved navigation performance. The algorithm, an extension of the Baum-Welch algorithm, learns without a teacher and addresses the issues of limited memory and the cost of collecting training data. Empirical results show that the algorithm learns good Markov models with a small amount of training data.

In [55], a general framework for using Monte Carlo methods in dynamic systems is provided and its wide applications are discussed. Under this framework, several currently available techniques are studies and generalized to accommodate more complex features. All these methods are partial combinations of three ingredients: importance sampling and resampling, rejection sampling, and Markov chain iterations. Guidelines on how they should be used and under what circumstance each method is most suitable is provided. Through the analysis of differences and connections, these methods are consolidated into a generic algorithm by combining desirable features. In addition, a general use of Rao-Blackwellization to improve performance is proposed. Examples from econometrics and engineering are presented to demonstrate the importance of Rao-Blackwellization and to compare different Monte Carlo procedures.

In [56] and [57], particle filters (PFs) proved to be powerful sampling based inference/learning algorithms for dynamic Bayesian networks (DBNs). They allow us to treat, in a principled way, any type of probability distribution, nonlinearity and non-stationary. They have appeared in several fields under such names as "condensation", "sequential Monte Carlo"

and “survival of the fittest”. This paper shows how can be exploited the structure of the DBN to increase the efficiency of particle filtering, using a technique known as Rao-Blackwellisation. Essentially, this samples some of the variables, and marginalizes out the rest exactly, using the Kalman filter, HMM filter, junction tree algorithm, or any other finite dimensional optimal filter. RaoBlackwellised particle filters (RBPFs) lead to more accurate estimates than standard PFs. RBPFs is demonstrated on two problems, namely non-stationary online regression with radial basis function networks and robot localization and map building. Other potential application areas are discussed and references are provided to some finite dimensional optimal filters.

In [58] is shown that to navigate reliably in indoor environments, a mobile robot must know where it is. Thus, reliable position estimation is a key problem in mobile robotics. Probabilistic approaches are among the most promising candidates to providing a comprehensive and real-time solution to the robot localization problem. However, current methods still face considerable hurdles. In particular, the problems encountered are closely related to the type of representation used to represent probability densities over the robot’s state space. Recent work on Bayesian filtering with particle-based density representations opens up a new approach for mobile robot localization, based on these principles. Monte Carlo Localization method, where the probability density is represented, maintains a set of samples that are randomly drawn from it. By using a sampling-based representation a localization method that can represent arbitrary distributions is obtained. Experimentally is shown that the resulting method is able to efficiently localize a mobile robot without knowledge of its starting location. It is faster, more accurate and less memory-intensive than earlier grid-based methods.

In [59], Monte Carlo methods are shown to revolutionize the on-line analysis of data in fields as diverse as financial modelling, target tracking and computer vision. These methods, appearing under the names of bootstrap filters, condensation, optimal Monte Carlo filters, particle filters and survival of the fittest, have made it possible to solve numerically many complex, non-standard problems that were previously intractable. This reference presents the first comprehensive treatment of these techniques, including convergence results and applications to tracking, guidance, automated target recognition, aircraft navigation, robot navigation, econometrics, financial modelling, neural networks, optimal control, optimal filtering, communications, reinforcement learning, signal enhancement, model averaging and

selection, computer vision, semiconductor design, population biology, dynamic Bayesian networks, and time series analysis.

In [60] it is shown that for many application areas becomes important to include elements of nonlinearity and non-Gaussianity in order to model accurately the underlying dynamics of a physical system. Moreover, it is typically crucial to process data on-line as it arrives, both from the point of view of storage costs as well as for rapid adaptation to changing signal characteristics. In this paper both optimal and suboptimal Bayesian algorithms for nonlinear/non-Gaussian tracking problems are used, with a focus on particle filters. Particle filters are sequential Monte Carlo methods based on point mass (or "particle") representations of probability densities, which can be applied to any state-space model and which generalize the traditional Kalman filtering methods. Several variants of the particle filter such as SIR, ASIR, and RPF are introduced within a generic framework of the sequential importance sampling (SIS) algorithm. These are discussed and compared with the standard EKF through an illustrative example.

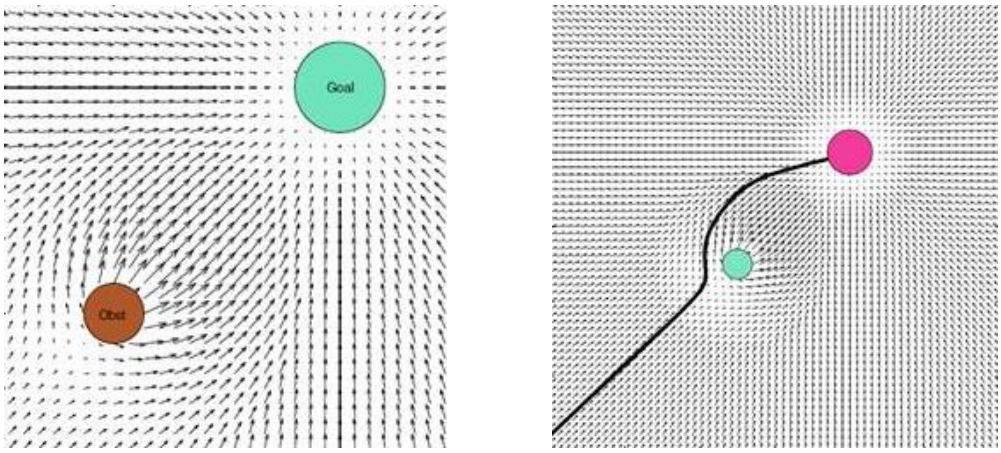
Chapter 3
Robot Navigation with Obstacle Avoidance

As shown in Chapter 2, various techniques are applied to mobile robot obstacle avoidance. In this thesis, we choose Artificial Velocity Potential Field derived from the earlier Artificial Potential Field Approach. In this case the robot builds its local map based on data from on-board bearing sensors, and then the controller will choose proper direction in which to avoid previously unknown obstacles. After determining the turning direction from the local map, the robot controller will apply Artificial Velocity Potential Field method to bypass obstacles.

3.1 Artificial Potential Field Method

In this thesis, the robot is assumed to travel towards a known goal position while avoiding obstacles in between on its path. Generally, Artificial Potential Field (APF) requires a certain field built around the obstacles so that when the robot enters this field, it will be subject to a repulsive force to avoid arriving too close. Moreover, in the whole environment, there is a field towards the goal that attracts the robot.

Figure 3.1 shows the artificial potential field and a possible robot path. In Figure 3.1 (a), we can see in the whole environment, where numerous arrows point towards the goal which means when the robot is located into this environment, it will directed towards the goal. The circle field around the obstacle contains arrows pointing away from the obstacle. When the robot moves into this circle, it will be subject to a repulsive force away from the obstacle. As a result, when close to the goal the robot will be subject the addition of both attractive force from the goal and the repulsive force from the obstacle. On the contrary, when is far from the goal, the effect becomes smaller, even getting to zero (local minima which will be analyzed later), in order to guide the robot to avoid the obstacle.



(a) (b)

Figure 3.1 (a) Artificial potential field; (b) A possible path planning in artificial potential field.

Consequently, in this scenario the robot will travel towards the goal while not hitting the obstacle in front of it, as shown in Figure 3.1 (b). The red circle in Figure 3.1 (b) is the goal and the green circle is an obstacle. The path planning is the black line from the left bottom corner to the goal.

3.1.1 The Principle of APF

Artificial Potential Field (APF) is a reactive path planning method. It is a reactive approach since the trajectories are not planned in advance but are obtained while executing actions by differentiating a function, called *potential function* and its negative gradient is the net-force acting on the robot [28].

The output of a potential function could be considered as the energy and APF method seeks a function can generate a force on the robot so that the energy of the system is minimized and reach its minimum value at the goal position. Therefore, moving a robot to the goal position could be considered as moving it from a *high-value* state to a *low-value* state by following a "downhill" path. The robot stops moving at the point where the gradient vanishes, which is $\nabla U\left(q^*\right)=0$ where q^* is called the critical value of U.

We will introduce APF using a classic attractive-repulsive potential function here. The intuition behind the attractive/repulsive potential functions is to attract the robot to the goal position while repelling it from the obstacles by superposing these two effects into one

resultant force. This potential function can be written as

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{3.1}$$

where $U_{att}(q)$ is the attractive potential, $U_{rep}(q)$ is the repulsive potential and q is the Cartesian coordinate of the robot.

The attractive potential monotonically increases with the current distance $d(q, q_{goal})$ of the robot to the goal, q_{goal}. Two common $U_{att}(q)$ functions are conical forms and quadratic form [28],

$$U_{att}(q) = \begin{cases} \frac{1}{2}\zeta d^2, & d \le d^*_{goal} \\ \zeta d^*_{goal} d - \frac{1}{2}(d^*_{goal})^2, & d > d^*_{goal} \end{cases} \tag{3.2}$$

where d is the distance of the robot to the goal, i.e. $d(q, q_{goal})$, ζ is the attraction gain, and d^*_{goal} is the threshold distance from the robot to the goal after which quadratic function changes to the conical form.

The movement is realized by following the negative gradient of the potentials, i.e. the force

$$F(q) = -\nabla U(q) \tag{3.3}$$

Substituting equation (3.2) in, we obtain

$$F_{att}(q) = -\nabla U_{att}(q) = \begin{cases} \zeta(q_{goal} - q), & d \le d^*_{goal} \\ -\zeta d^*_{goal}, & d > d^*_{goal} \end{cases} \tag{3.4}$$

The meanings of all parameters are the same as the definition of equation (3.2).

The term $U_{rep}(q)$ is defined as [45]

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{D(q)} - \frac{1}{Q^*}\right)^2, & D(q) \le Q^* \\ 0, & D(q) > Q^* \end{cases} \tag{3.5}$$

where Q^* is the distance threshold for an obstacle to create a repulsion effect on the robot, $D(q)$ is the distance from the robot to the closest obstacle and η is the repulsive gain.

Based on equation (3.3), the gradient of the repulsive potential function, i.e. the repulsive force becomes [28]

$$F_{rep}(q) = -\nabla U_{rep}(q) = \begin{cases} \eta\left(\frac{1}{D(q)} - \frac{1}{Q^*}\right)\frac{1}{D^2(q)}\nabla D(q), & D(q) \le Q^* \\ 0, & D(q) > Q^* \end{cases} \quad (3.6)$$

The meanings of all parameters are the same as the definition of equation (3.5).

For both the attractive potential function and the repulsive potential function, the gain parameters (i.e. η and ζ) and the threshold parameters (i.e. Q^* and d^*_{goal}) are set empirically.

There is usually one goal for the robot to follow, but there may be several obstacles to avoid in the scenario. Thus the total potential field can be obtained by summing up the repulsive potentials caused by all of the obstacles and the attractive potentials produced by the goal,

$$U(q) = U_{att}(q) + U_{rep}(q) = U_{att}(q) + \sum_{k=1}^{N} U_{repi}(q) \quad (3.7)$$

Figure 3.1 shows how equation (3.7) works with only one obstacle to avoid. The final movement of the robot is realized by following the negative gradient (i.e. the force) of the sum of attractive/repulsive potentials,

$$F(q) = F_{att}(q) + F_{rep}(q) = -\nabla U_{att}(q) - \nabla U_{rep}(q) \quad (3.8)$$

3.2 Artificial Velocity Potential Field Method

A classic APF method has been discussed above, in which force potentials are used. However, in this thesis is used a Velocity Potential Field (VPF) inspired by hydrodynamics [29, 30]. Similar to APF, VPF uses the velocity potential but the force potential to represent the energy of the system. When the velocity potential functions are set up, the following equation will be used to obtain the velocity,

$$v(q) = -\nabla U(q) \quad (3.9)$$

where $v(q)$ is the corresponding velocity of the robot and $U(q)$ represents the velocity potential energy of the system.

Normally, there would be two velocity terms applied on the robot: one is the attractive velocity term that attracts the robot moving to the goal, the other is the repulsive velocity term to avoid the robot collision with the obstacles. Since the local minima may occur, i.e. the singularity of the robot when it travels to an obstacle, we add one more velocity term (u_t)

normal to the direction from the robot to the closest obstacle shown in Figure 3.2 [37]. The sensor circle around the robot is the safety range of the robot, which means it will have to turn when meeting obstacles in that area. We will talk about the issue of local minima in details later. Thus, the final velocity of the robot based on our assumptions would be

$$\mathbf{v}_{tot}(q) = \mathbf{v}_a(q) + \mathbf{v}_n(q) + \mathbf{v}_t(q) = -\nabla U_a(q) - \nabla U_n(q) - \nabla U_t(q) \qquad (3.10)$$

The addition of three velocity terms is vector addition as all three terms are vectors.

Now we need find the attractive potential function and two repulsive potential functions respectively. The attractive velocity potential in VPF is chosen as

$$U_a(q) = \begin{cases} \frac{1}{2}\zeta\left(d - d^*_{goal}\right)^2, & d \le d^*_{goal} \\ 0, & d > d^*_{goal} \end{cases} \qquad (3.11)$$

where d is the distance of the robot to the goal, i.e. $d\left(q, q_{goal}\right)$, ζ is the attraction gain, and d^*_{goal} is the threshold distance to the goal less than which the robot will suffer from the potential field.

Based on equation (3.9), we obtain

$$\mathbf{v}_a(q) = -\nabla U_a(q) = \begin{cases} \zeta\left(d^*_{goal} + q_{goal} - q\right), & d \le d^*_{goal} \\ 0, & d > d^*_{goal} \end{cases} \qquad (3.12)$$

The meanings of all parameters are the same as the definition of equation (3.11).

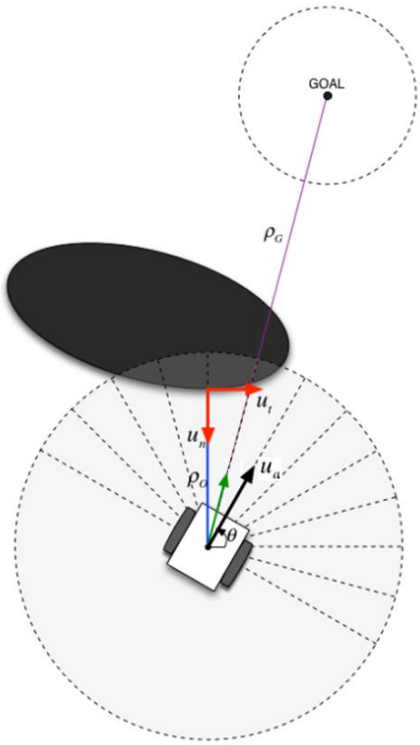


Figure 3.2 The attractive and repulsive velocity vectors.

For the normal repulsive velocity potential function, we use the repulsive force potential corresponding to the normal Velocity Potential U_n, which defined in the direction of the position variable from the robot to the closest obstacle (i.e. $D(q)$).

$$U_n(q)=\begin{cases}\frac{1}{2}\eta\left(\frac{1}{D(q)}-\frac{1}{Q^*}\right)^2, & D(q)\le Q^* \\ 0, & D(q)>Q^*\end{cases} \tag{3.13}$$

where Q^* is the distance threshold for an obstacle to create a repulsion effect on the robot, $D(q)$ is the distance from the robot to the closest obstacle and η is the repulsive gain.

The negative gradient gives the normal repulsive velocity vector

$$\mathbf{v}_n(q)=-\nabla U_n(q)=\begin{cases}\eta\left(\frac{1}{D(q)}-\frac{1}{Q^*}\right)\frac{1}{D^2(q)}\nabla D(q), & D(q)\le Q^* \\ 0, & D(q)>Q^*\end{cases} \tag{3.14}$$

The last term is determined by the tangential repulsive velocity function, and this term will determine the robot to move away from the local minimum point. We use the same potential function [37]

$$U_t(q)=\begin{cases}\frac{1}{2}\eta\left(\frac{1}{D(q)}-\frac{1}{Q^*}\right)^2, & D(q)\le Q^* \\ 0, & D(q)>Q^*\end{cases} \tag{3.15}$$

Following the same procedure we get

$$\mathbf{v}_t(q)=-\nabla U_t(q)=\begin{cases}\eta\left(\frac{1}{D(q)}-\frac{1}{Q^*}\right)\frac{1}{D^2(q)}\nabla D(q), & D(q)\le Q^* \\ 0, & D(q)>Q^*\end{cases} \tag{3.16}$$

The meanings of all parameters in equations (3.14) to (3.16) are the same as the definition of equation (3.13).

3.3 The Local Minima Problem

The local minima problem is a singularity phenomenon in the robot navigation. The basic principle is that the net force or velocity of the robot is zero when it reaches certain point which is called the local minimum. As shown in Figure 3.3 [37], the robot is posed by an attractive force from the goal behind the obstacle; at the same time, it is subject to a repulsive force due to the obstacle right opposite of the direction of that attractive force. Therefore, the net force of the robot becomes zero which stops the robot in a local minimum point.

In the simulation, the robot may travel to the goal while going toward an obstacle whose shape is concave like in Figure 3.4. Without the tangential force applied to the robot, the robot will be stuck in a point where the repulsive force or velocity caused by the concave obstacle is equal to the attractive force or velocity produced by the goal, and the directions are opposite.

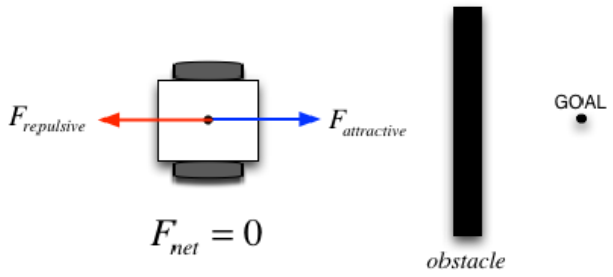


Figure 3.3 The local minima problem.

In Figure 3.4, the red diamond and its round outside circle represents the goal, and the arrow with its outside circle represents the robot. In the beginning, the robot is attracted by the goal moving directly toward the goal, but when it goes deep inside the concave obstacle, it will also suffer from the repulsive force produced by the obstacle, which causes the robot to be stuck there.

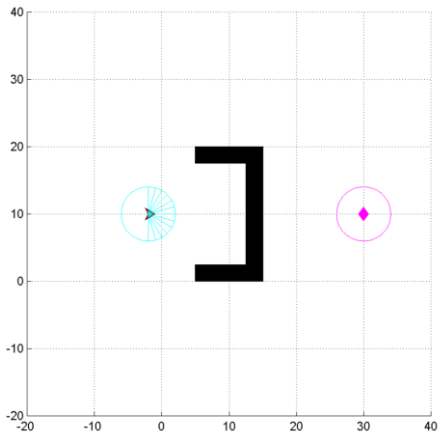


Figure 3.4 The local minima inside the concave obstacle.

In later simulations, when the robot goes into the area of the obstacle, it will not be affected by the attractive force but only applied by the repulsive force so that the robot will move backwards. When it gets back, it goes out of the area of the obstacle which means it will only suffer from the attractive force again. Therefore, the robot will move toward the goal again along, then it goes into that area once again and it will move back the second time. The robot will move only forward and back, forward and back, due to a singularity in the local minimum

point, the addition of a tangential term will solve the local minimum problem successfully.

3.4 A Modified Velocity Potential Field Method

Before conducting the simulation on MATLAB, we will introduce a modified velocity potential field method [37]. First, we will define an exponential function which will be used widely in our simulation:

$$f(d) = e^{-d} \tag{3.17}$$

where d is per unit distance. In our case, we use

$$f\left(\rho_G, \rho_{GR}\right) = e^{-\left(\rho_G / \rho_{GR}\right)} \tag{3.18}$$

where ρ_G is a distance of a robot to the goal and ρ_{GR} is a critical radius around the goal inside which the robot decrease the speed and stop at the goal position. This exponential equation (3.18), when $\rho_G >> \rho_{GR}$, $f\left(\rho_G, \rho_{GR}\right)$ is getting close to zero.

Similarly, we define

$$f\left(\rho_O, \rho_{OR}\right) = e^{-\left(\rho_O / \rho_{OR}\right)} \tag{3.19}$$

where ρ_O is a distance of a robot to an obstacle and ρ_{OR} is a threshold radius for the obstacle within which the robot will be subject to a repulsive velocity to avoid it hitting into the obstacle.

Based on VPF we mentioned before, we need find the attractive velocity produced by the goal and two repulsive velocities yielded by the obstacle.

For the attractive velocity, we use [46]

$$\mathbf{v}_a(q) = \begin{cases} k_a u_a \left(1 - f\left(\rho_G, \rho_{GR}\right)\right), & \rho_G \le d^*_{goal} \\ k_a u_a \left(1 - f\left(\rho_G, \rho_{GR}\right)\right), & \rho_G > d^*_{goal} \end{cases} \tag{3.20}$$

where k_a is the attractive gain, u_a is set to the maximum cruising velocity of the robot and d^*_{goal} is the threshold distance to the goal less than which the robot will suffer from the potential field. As we can see, when $\rho_G > d^*_{goal}$, we do not set the attractive velocity to zero like equation (3.12), that is because we want the robot to be attracted the whole time when we put it into the environment wherever it is.

So equation (3.20) can be written as

$$\mathbf{v}_a(q) = k_a u_a \left(1 - f\left(\rho_G, \rho_{GR}\right)\right) \tag{3.21}$$

From this equation, we can see that when the robot is far away from the goal (i.e. $\rho_G >> \rho_{GR}$), $f(\rho_G, \rho_{GR})$ is getting close to zero and is very small. So the attractive velocity $\mathbf{v}_a$ is getting to $k_a u_a$. When the robot moves closer to the goal, $f(\rho_G, \rho_{GR})$ is getting much closer to 1. So the attractive velocity $\mathbf{v}_a$ decreases and becomes smaller and smaller than $k_a u_a$. Finally, when the robot reaches the goal (i.e. $f(\rho_G, \rho_{GR})$=1), the attractive velocity $\mathbf{v}_a$ gets to 0, which make the robot stop on the goal point.

Let's move on to the two repulsive velocities afterwards. Similar to the attractive velocity, we also use an exponential function to form the repulsive velocities [46].

$$\mathbf{v}_n(q) = \begin{cases} k_n u_n f(\rho_O, \rho_{OR}), & \rho_O \le \rho_{OR} \\ 0, & \rho_O > \rho_{OR} \end{cases} \tag{3.22}$$

Equation (3.22) will be used to represent the normal repulsive velocity, where k_n is the normal repulsive gain, u_n is the magnitude of the normal velocity and is set to a function of the distance to the obstacle $1/\rho_O$, ρ_O and ρ_{OR} have the same meanings as the parameters definition of equation (3.19).

As a result of equation (3.22), when the robot enters into the threshold area of the obstacle (i.e. $\rho_O \le \rho_{OR}$), it will suffer from a repulsive normal velocity. The closer it is from the obstacle, the bigger u_n is. Similarly, $f(\rho_O, \rho_{OR})$ gets bigger when the robot is getting closer to the obstacle. u_n and $f(\rho_O, \rho_{OR})$ are like two weighted factors combining together so that they can perform better to avoid obstacles. On the other hand, if the robot is not that close to the obstacle (i.e. $\rho_O > \rho_{OR}$), it will not be affected by the obstacle.

The last term needed to be considered is the tangential repulsive velocity which, in our case, will be applied by the same model as the normal velocity.

$$\mathbf{v}_t(q) = \begin{cases} k_t u_t f(\rho_O, \rho_{OR}), & \rho_O \le \rho_{OR} \\ 0, & \rho_O > \rho_{OR} \end{cases} \tag{3.23}$$

where k_t is the tangential repulsive gain, u_t is the magnitude of the tangential velocity and is set to a function of the distance to the obstacle $1/\rho_O$, ρ_O and ρ_{OR} have the same meanings as the parameters definition of equation (3.19).

Accordingly, when the robot travels into the threshold area of the obstacle (i.e. $\rho_O \leq \rho_{OR}$), it will be affected by a repulsive tangential velocity. As in the case of the normal repulsive velocity function, this tangential velocity will increase if the robot gets close to the obstacle. If the robot is far from the obstacle (i.e. $\rho_O > \rho_{OR}$), there is no tangential velocity.

3.5 Turning Direction Strategy

In our simulation in MATLAB, we use holonomic robot which means the robot can spin in certain point to turn its direction. The question is in which direction does the robot turn so that it can avoid the obstacle and reach the goal efficiently (i.e. travelling through the shorter path and saving time).

The robot and its surrounding sensing environments are shown in Figure 3.5. In this figure, the yellow range and green range comprise the left and right sensor ranges. The outermost circle is the searching range which means, the robot can detect objects when they are into this area. The red circle, close to the robot, is safety range meaning that robot controller will use the obstacle avoidance algorithm when obstacles are entering in this range.

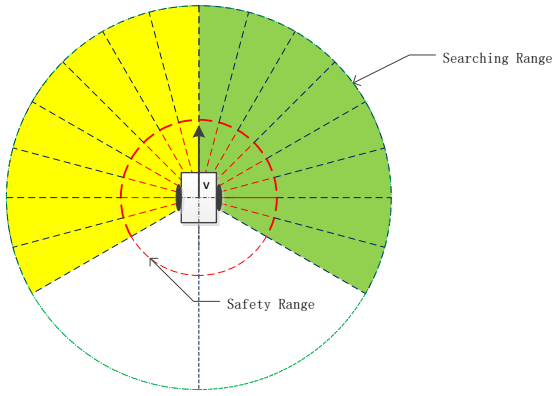


Figure 3.5 The robot with its sensors in the simulation.

We divide the whole searching range into three equal sections (i.e. each section consist of a 120 degree fan-shaped sector). The section located behind the robot moving forward is not allocated to sensors since in our experiment we do not need the robot move backwards. The front two sections are set up sensors in order to obtain obstacle data in front of the robot. Based

on the data from sensors, the robot can build up a local map, and based on this map, it can determine its turning direction.

3.5.1 Building Local Map

Assuming there are obstacles entering into the searching range of the robot shown in Figure 3.6, the robot will build its local map simultaneously as it travels to the goal.

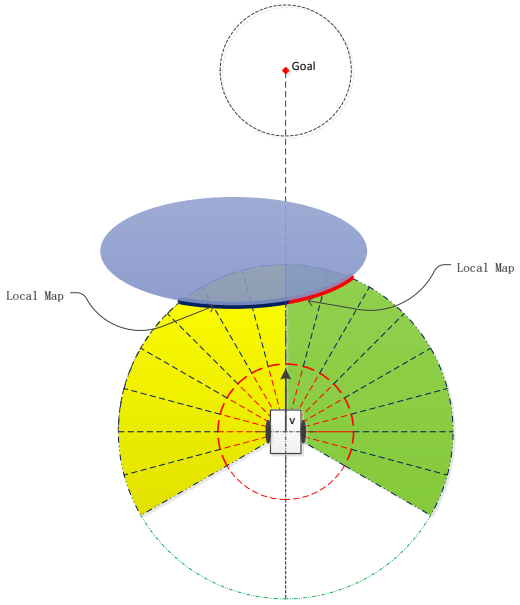


Figure 3.6 The local map robot built when meeting obstacles.

As we can see in Figure 3.6, the red and black solid lines comprise the local map. Since the black line in yellow section has larger dimensions than red line in green section, the robot will choose to turn right in order to save time and energy to avoid obstacles and reach the goal. In fact, the robot will not turn until the obstacle travels into the safety radius. However, the map building begins only when the obstacle enters into the searching range.

There are some other more complicated cases than the scenarios above, which include two obstacles of different dimension and multiple obstacles which correspond to a concave shaped obstacle and so on. These scenarios are also considered in next chapter.

3.6 Controller Design

The aim of the controller design is to calculate the new target position of the robot based on the old coordinates and current sensor data. Actually, the controller design for the robot navigation without obstacle collision was explained in Section 3.4. The modified VPF method is applied here to the robot controller design.

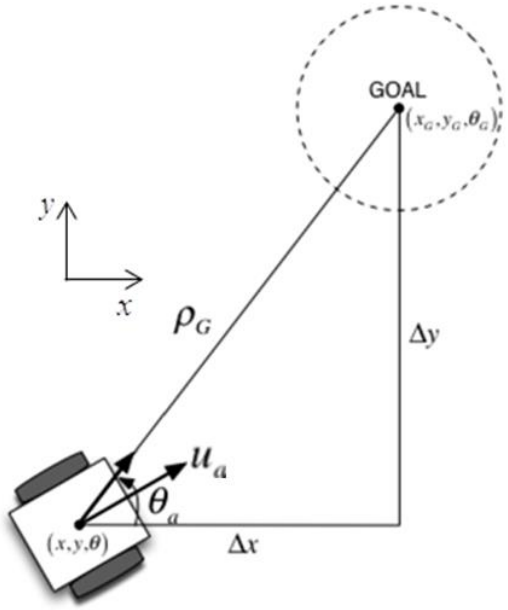


Figure 3.7 Robot navigation to the goal without obstacles.

The working environment for the robot navigation is in the Cartesian coordinate system. In Figure 3.7, the goal has the coordinate (x_G, y_G, θ_G), and the robot has the current coordinate (x, y, θ), so the distance between these two points [46]

$$\rho_G = \sqrt{(x_G - x)^2 + (y_G - y)^2} \tag{3.24}$$

Setting

$$\Delta x = x_G - x \tag{3.25}$$

$$\Delta y = y_G - y \tag{3.26}$$

The angle attracted by the goal is

$$\theta_a = \arctan\left(\frac{\Delta y}{\Delta x}\right) \tag{3.27}$$

The error between θ_a and θ is

$$\Delta\theta = \theta_a - \theta \tag{3.28}$$

Based on equation (3.21), we can get the attractive velocity u_a, applied to the traveling speed when there is no obstacle. In the beginning, u_a is not pointing toward the goal and the error between θ_a and θ is calculated. Afterwards, the robot will turn based on certain criteria which let the robot turn efficiently to the desired direction. The obstacle-free state of the robot is that the robot travels towards the goal and finally reaches the goal. When the robot is turning, it will travel on a curve. We need calculate the Cartesian coordinates of the robot to obtain its trajectory.

When the robot meets the obstacle, as shown in Figure 3.2 [37], based on the modified VPF above, it will suffer from two repulsive velocities which force it to turn. The turning strategy has been presented in Section 3.5. The direction of tangential repulsive velocity depends on how the robot turns. If the robot turns right, the tangential velocity points to the right, vice versa. Additionally, when the robot detects an obstacle in its safety range, the attractive velocity u_a is not acting, because otherwise it may cause the contradictory command. For example, in a certain case, if the robot meets the obstacle, based on the direction of the attractive velocity, the robot is supposed to turn left. However, relied on the turning strategy proposed before, it should turn right. So if we counted it both, the robot will go straight instead of turning and will hit the obstacle, which is not what we want. The solution to this problem is switching different strategies for different cases.

Strategy 1: The robot will navigate to the goal with the velocity defined in equation (3.21).

Strategy 2: The robot will ignore the goal and avoid the obstacle with the velocity summation of two velocities defined respectively in equation (3.22) and (3.23).

The scenarios	No obstacles detected	Obstacles detected
Solutions	*Strategy 1*	*Strategy 2*

Table 3.1 Solutions for different scenarios.

Table 3.1 illustrates the solutions for different scenarios. Firstly, when the robot navigates

to the goal without obstacles detected *Strategy 1* is used. When an obstacle is detected by the robot, *Strategy 2* is applied. Finally, when the robot successfully passes beyond the obstacle and no obstacles are detected again, it switches back to *Strategy 1* until reaching the goal.

3.7 Simulation Algorithm

Our simulations are built on MATLAB which is a high-level programming software used by engineers to analyze data, develop algorithms, create models and so on. In this section, flowcharts will be used to demonstrate the methodology. A complete flowchart is composed of a number of simple flowchart parts. So fundamental flowchart symbols need to be presented.

Certain fundamental flowchart symbols are listed in Figure 3.8.

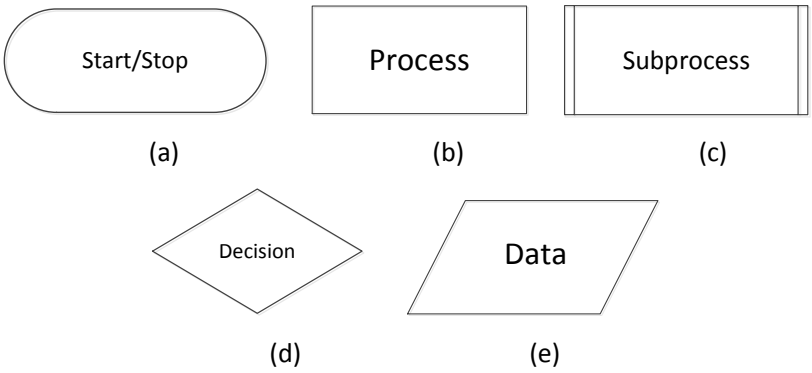
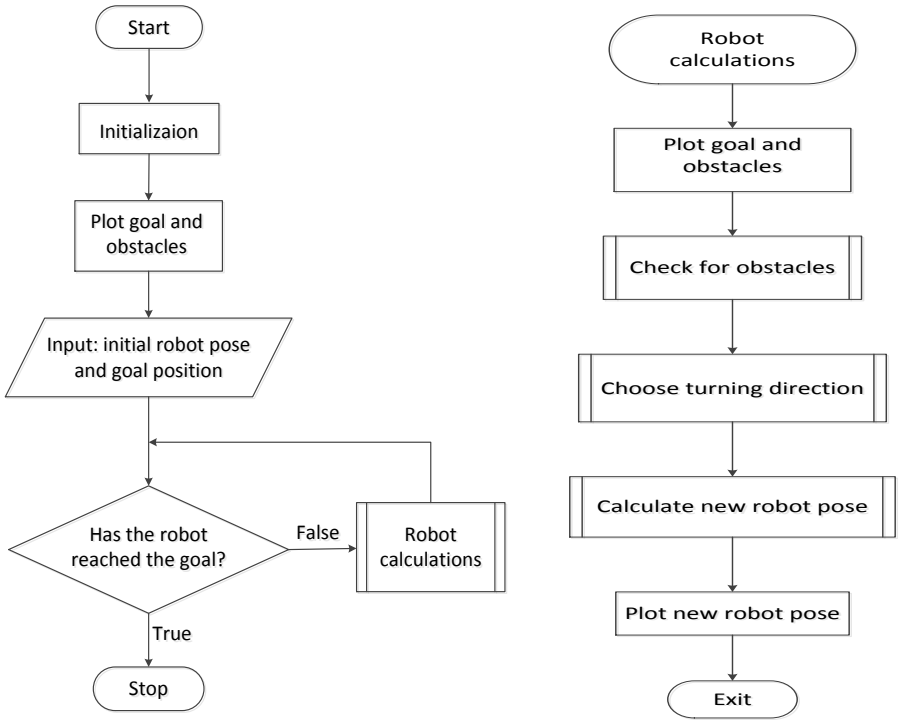


Figure 3.8 Basic flowchart symbols.

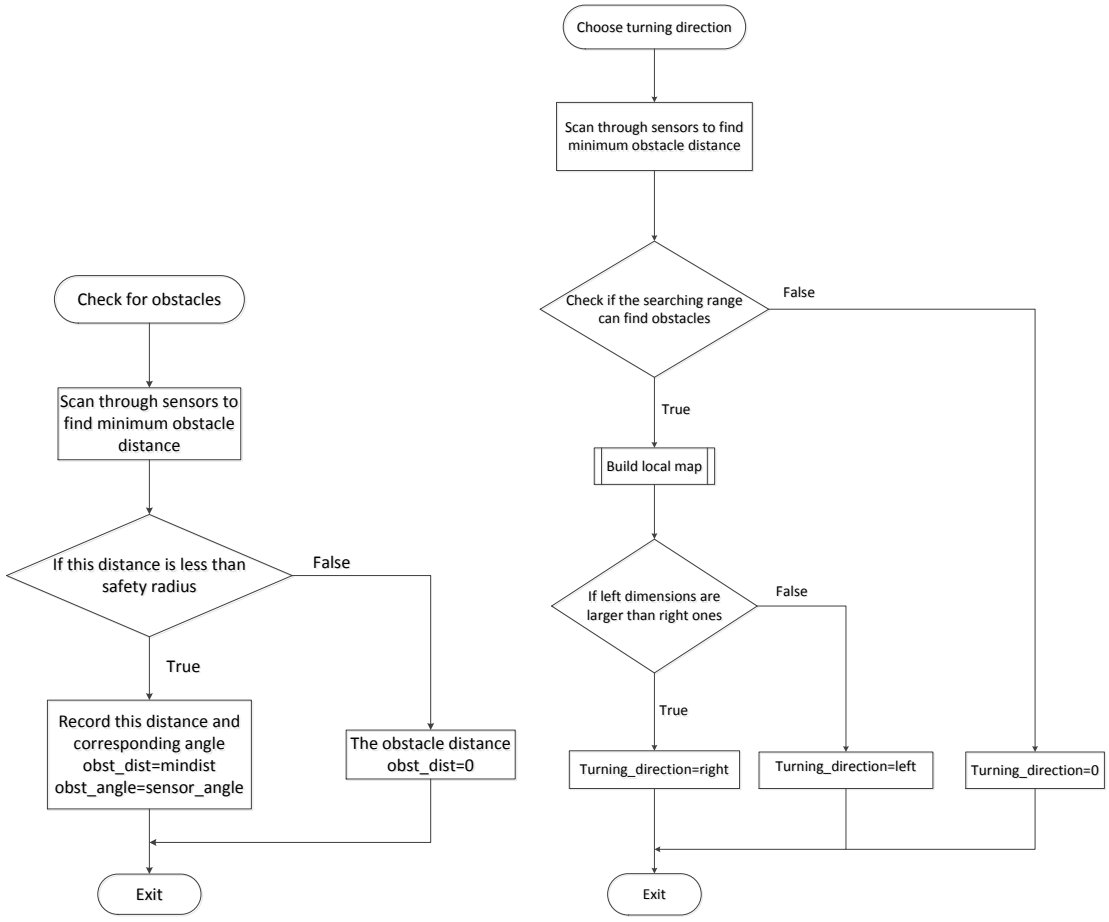


(a) (b)

Figure 3.9 (a) main function; (b) robot calculations function.

In Figure 3.8, (a) refers to the beginning and end of a procedure or program; (b) represents a processing program; (c) shows a sub process which normally is defined in another flowchart; (d) indicates that a decision point in the program where the logic flow will follow one of the two paths depending on a given situation; the last one (e) means certain input and output processes are given.

Based on corresponding proposed approach regarding obstacles avoidance for robot, we can draw a flowchart of main function in Figure 3.9 (a), where the main function needs to initialize certain parameters such as maximum velocity and angular velocity, sensor radius and goal radius, time constants and map dimensions.



(a) (b)

Figure 3.10 (a) Check for obstacles function; (b) Choose turning direction function.

In Figure 3.9 (b), sub function *Robot calculations* refers to the major calculations of the whole program which comprises a major loop. In the beginning, the robot will search for obstacles. When obstacles invade into searching range of the robot, it will build up a local map regarding to surrounding environment. Afterwards, the robot will choose turning direction based on the local map it has built. We can see sub function *Build local map* is a sub function of function *Choose turning direction*.

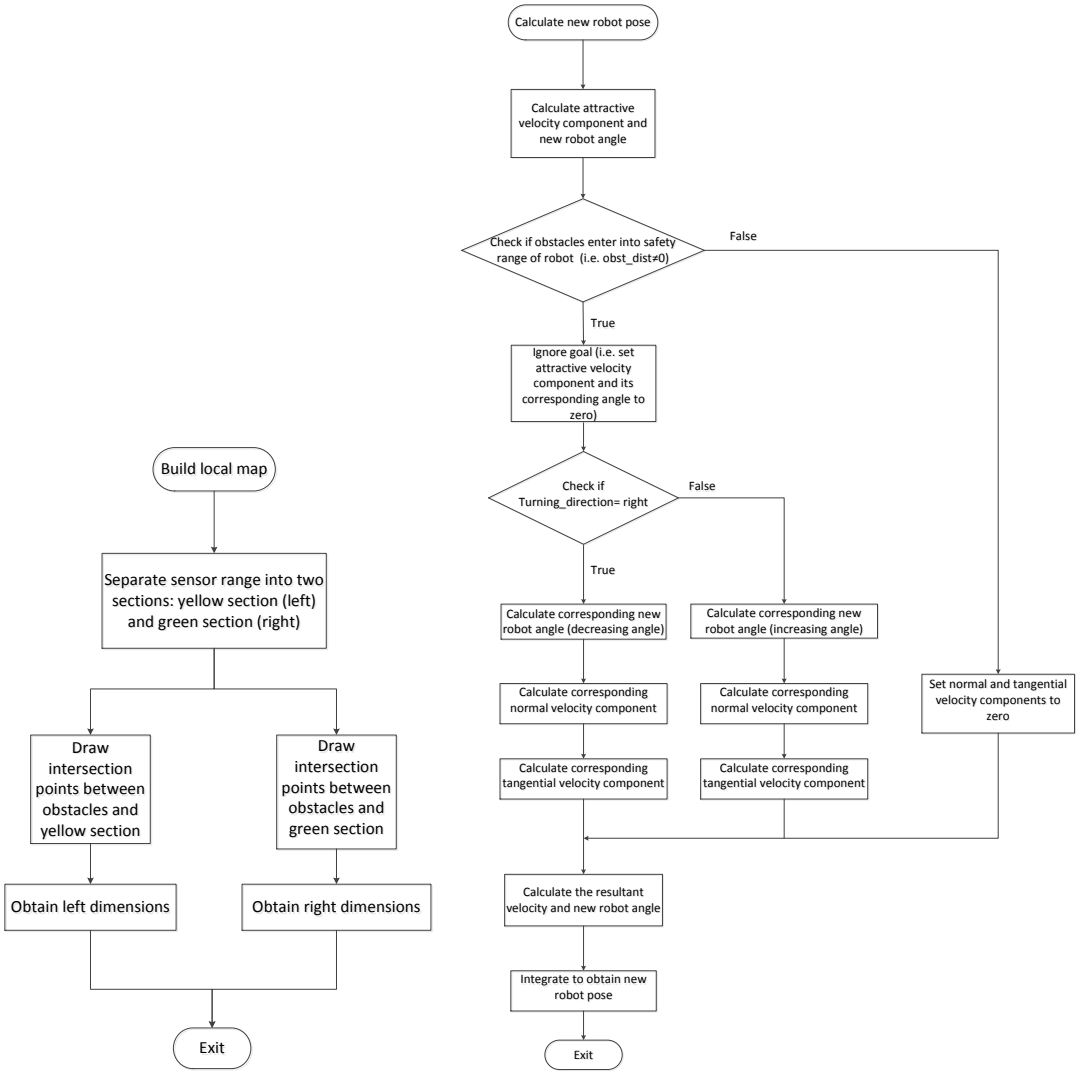


Figure 3.11 (a) Build local map function; (b) Calculate new robot pose function.

Even though the robot has determined the turning direction, it will not execute this turning command until it is close enough to obstacles (i.e. the distance from the robot to the closed obstacle is less than safety radius). Robot will move towards the goal and turn away when safety radius is invaded. Then we can calculate new robot pose based on the controller we designed until the robot reaches the goal.

Check for obstacles function in Figure 3.10 (a) returns two parameters to the main function, which are minimum distance from obstacles to the robot and corresponding angle. This returning distance must be less than safety radius in order to be applied to obstacle avoidance algorithm.

Choose turning direction function in Figure 3.10 (b) returns turning direction to the main function. There is a *Build local map* sub function in it since the determination of turning direction is relied on how the local map of robot would be. In sub function *Build local map* in Figure 3.11 (a), we can see the robot sensor range is divided into two sections, one is left front section, and the other is right front section. Local map could be obtained by these two sensor sections intersect with obstacles, which can be seen in Figure 3.6. In Figure 3.6, we can see left dimensions indicated with black bold solid line are larger than right ones indicated with red bold solid line. Hence in function *Choose turning direction*, robot will turn right in order to avoid obstacles efficiently and reach the goal quickly.

After determining turning direction, all we left to do is to calculate the new robot pose in function *Calculate new robot pose* in Figure 3.11 (b). Attention is required that robot will execute the turning direction only when the distance from the robot to the closest obstacle is less than safety radius. This means that even though local map is obtained when the robot meets obstacles in searching range, the robot will not execute the resulting turning direction until it meets obstacles in safety range. The detailed algorithm is demonstrated below. When the robot meets obstacles in safety range, it will ignore influence about the goal which is setting attractive velocity and relevant angle to zero. The robot will be allocated corresponding normal velocity and tangential velocity and turning angle based on the determination of turning direction. Then we can find the resultant velocity and new robot angle, through integrating which new robot pose could be obtained.

Simulations based on the algorithm we have explained above were carried out for two different obstacles scenarios. One is for the robot navigation regarding to two distinct sized obstacles; the other is robot navigation about a concave obstacle with dissymmetry composed of five small obstacles. The simulation results will be presented in Chap. 5 along with the FastSLAM approach.

Chapter 4

Robot Navigation Using the FastSLAM Approach

In this chapter, FastSLAM approach is applied to robot tracking when navigating to a goal without collision with obstacles, and then, based on the estimations of robot positions, and estimations of relative distances to obstacles a local map of robot surroundings is built. Section 4.1 introduces the principles of the FastSLAM approach and of the particle filter method outlining the strengths and weakness of particle filter method compared with other filters such as the Kalman filter, the extended Kalman filter, etc. Section 4.2 looks at robot navigation using the FastSLAM approach and demonstrates the algorithm in simulations. Section 4.3 combine section 4.2 with obstacles avoidance approach illustrated in Chap.3 and proposes robot navigation using the FastSLAM approach with obstacles avoidance using velocity potential field approach.

4.1 Principles of the FastSLAM Approach

The FastSLAM approach is a factored solution to the SLAM (Simultaneous Localization and Mapping) problem. We have briefly introduced the SLAM problem already, which represents a process of localizing a robot and accurately mapping its surroundings simultaneously. FastSLAM algorithm obtains estimations of full posterior distribution over robot pose and landmark locations recursively. This process is decomposed as an exact factorization of the posterior into a product of a distribution of robot paths and conditional landmark distributions.

In FastSLAM approach, robot path estimation is based on a particle filter, and landmarks location estimations on Kalman filters which are attached on individual robot pose particles. Our work is mainly focused on robot path estimation using particle filter method. Landmark

location estimations using Kalman filters assume that we have multi measurements for the same landmark. While in our research, we did only one measurement for one landmark, thus we do not apply Kalman filters into our landmark location estimations.

4.1.1 Particle Filter Method: Strengths and Weaknesses

There are plenty of optimal state estimation methods in statistics such as the Kalman filter method, the extended Kalman filter (EKF), the unscented Kalman filter (UKF) and the particle filter method. Each method has its own advantages and disadvantages and has its application field, respectively. For instance, the last three filtering methods are applied to nonlinear systems whereas the first one, the Kalman filter method, is applied only to linear systems.

The Kalman filter is a fundamental and widespread filtering method in every area of engineering. Mostly, it is used to get the optimal state estimation when the noise is Gaussian. However, the Kalman filter is an optimal estimator even when the noise is not Gaussian and is the optimal linear estimator.

The extended Kalman filter enlarges the application range to nonlinear systems because we see that linear systems do not really exist. Many systems are considered as linear systems as they are close enough to linear that linear estimation approaches give satisfactory results [47, Chapter 13]. When we deal with a system which does not behave linearly even over a small range of process, nonlinear estimators have to be developed.

If a system is getting more severe nonlinearities, the EKF is going to be difficult to tune and to obtain reliable estimates, which induced to the advent of the unscented Kalman filter (UKF). The EKF relied on linearization to propagate the mean and covariance of the state, while the UKF would reduce the linearization errors of the EKF [47, Chapter 14]. The use of the UKF can be an improved version of the EKF.

The particle filter is developed from the foundation of Bayesian state estimation, and could get good results of a nonlinear system at the price of computational effort. It has certain similarities with the UKF. Both the particle filter and the UKF need to choose a set of points to estimate the mean and covariance of the state. However, the points are chosen randomly in the particle filter, whereas in the UKF the points are chosen on the basis of a specific algorithm [47, Chapter 15]. Therefore, in a particle filter the number of points (particles) has to be much larger than in a UKF in order to get accurate results. There is another difference between the

two filters which is the estimation error in a UKF does not converge to zero in all cases, but in a particle filter does when the number of particles approaches infinity.

For a nonlinear system, the Kalman filter can be applied for state estimation. However, the optimal state estimator is the particle filter along with big computational effort. This also applies to a system with non-Gaussian noise. Although the particle filter and the UKF usually are used in nonlinear systems, they could also be applied in linear Gaussian systems. There is always a dilemma in between estimation accuracy and computational effort. One more interesting thing is The UKF plays a balance between the low computational effort of the KF and the high performance of the PF in both cases [47].

4.1.2 Principles of Particle Filter Method

Suppose we have a nonlinear system

$$\begin{cases} x_{k+1} = f_k\left(x_k, \omega_k\right) \\ y_k = h_k\left(x_k, v_k\right) \end{cases} \tag{4.1}$$

where the first equation in equation (4.1) indicates the system equation, which is the time propagation of the system, k is the time index, x_k is the state, ω_k is the process noise, y_k is the measurement, and v_k is the measurement noise. The functions $f_k(.)$ and $h_k(.)$ are time-varying nonlinear system and measurement equations, respectively. The noise sequences $\{\omega_k\}$ and $\{v_k\}$ are assumed as independent white noise processes with known probability density functions (pdf's).

We use x_0 to represent the initial state of the system before any measurements are taken. If the pdf of the initial state $p(x_0)$ is known, N initial particles can be randomly generated based on the pdf $p(x_0)$. Otherwise, we generate them randomly on all possibilities. These particles are denoted as $x_{0,i}^{+}$ (i=1,…,N) , i.e. $x_{k,i}^{+}$ for k=0.

For particles $x_{k,i}^{+}$ (i=1,…,N), the "+" superscript denotes that these particles are obtained when we took all of the measurements up to and including time k (i.e. these particles are called *a posteriori* particles). In this case, given that we take measurements the first time at k=1, it

means that no measurements are available for k=0, when we produce N particles $x_{0,i}^{+}$ (i=1,…,N). For particles $x_{k,i}^{-}$ (i=1,…,N), the "-"superscript indicates that these particles are obtained when we took all of the measurement before (but not including) time k, and these particles are called *a priori* particles.

When k=1, we take measurements the first time. We need propagate particles at time k=0 to ones at time k=1. Based on equation (4.1) and some declarations above, we get propagation equation for particles [47]

$$x_{k,i}^{-} = f_{k-1}\left(x_{k-1,i}^{+}, \omega_{k-1}^{i}\right) \qquad \left(i = 1,...N\right) \tag{4.2}$$

where each ω_{k-1}^{i} noise is randomly generated on the basis of the known pdf of ω_{k-1}. Substituting k for 1, we obtain

$$x_{1,i}^{-} = f_{0}\left(x_{0,i}^{+}, \omega_{0}^{i}\right) \qquad \left(i = 1,...N\right) \tag{4.3}$$

Based on the declaration above, particles $x_{1,i}^{-}$ (i=1,…,N) are obtained before any measurements are taken. For particles $x_{0,i}^{+}$ (i=1,…,N), we perform the time propagation step (equation (4.3)) to get *a priori* particles $x_{1,i}^{-}$ (i=1,…,N) based on the known process equation and the known pdf of the process noise.

Then, we need to compute the relative likelihood q_i of each particle $x_{k,i}^{-}$ conditioned on the measurement y_k. That is realized on evaluating the pdf $p\left(y_k \middle| x_{k,i}^{-}\right)$. Firstly, let us remind some fundamental concepts. One is likelihood function, and the other is relative likelihood function. Likelihood is like the reverse of probability, which is a function of how likely an event is. In a rigorous definition the likelihood function is function of a set of parameter values given some observed outcomes and is equal to the probability of those observed outcomes given those parameter values [35].

$$L\left(\theta \middle| x\right) = P\left(x \middle| \theta\right) \tag{4.4}$$

Suppose that maximum likelihood estimation for θ is $\hat{\theta}$, which means when $\theta = \hat{\theta}$, $L\left(\theta \middle| x\right)$ gets maximum values so that the system gets more plausible outcome. Relative plausibilities of other θ values can be obtained by comparing the likelihood of those other values with the likelihood of $\hat{\theta}$. The relative likelihood of θ is defined as

$$q = \frac{L\left(\theta \middle| x\right)}{L\left(\hat{\theta} \middle| x\right)} = \frac{P\left(x \middle| \theta\right)}{P\left(x \middle| \hat{\theta}\right)} \tag{4.5}$$

Assuming an m-dimensional measurement equation is given as [47]

$$\begin{aligned} y_k &= h\left(x_k\right) + v_k \\ v_k &\sim N\left(0, R\right) \end{aligned} \tag{4.6}$$

then a relative likelihood q_i that the measurement is equal to a specific measurement y^*, given that x_k equals to the particle $x_{k,i}^-$ can be computed as

$$\begin{aligned} q_i &= \frac{L\left(x_{k,i}^- \middle| y^*\right)}{L\left(\hat{x}_{k,i}^- \middle| y^*\right)} = \frac{P\left(y^* \middle| x_{k,i}^-\right)}{P\left(y^* \middle| \hat{x}_{k,i}^-\right)} \\ &\sim \frac{1}{\left(2\pi\right)^{m/2}\left|R\right|^{1/2}} \exp\left(\frac{-\left[h\left(x_{k,i}^-\right) - y^*\right]^T R^{-1}\left[h\left(x_{k,i}^-\right) - y^*\right]}{2} \right) \end{aligned} \tag{4.7}$$

The symbol $\sim$ in the equation above means that the probability is not really given by the expression on the right side, but is directly proportional to the right side. We can obtain the relative likelihoods for all the particles $x_{k,i}^-$ (i=1,…,N) [47].

The relative likelihood q_i for each particle is actually the importance weight W_k^i of each particle, which can be found in many references about particle filters. Now we normalize the weights as

$$w_k^i = \frac{W_k^i}{\sum_{j=1}^{N} W_k^j} \tag{4.8}$$

We can obtain the normalized importance weights w_k^i for all the particles $x_{k,i}^-$ (i=1,…,N) through equation (4.8), which ensures that the sum of all the normalized importance weights w_k^i is equal to one.

We combine all the particles $x_{k,i}^-$ with their weights w_k^i to form a set of particles $\left\{x_{k,i}^-, w_k^i\right\}\left(i = 1,...,N\right)$. Next step is resampling this set of particles to a brand new set of particles $\left\{x_{k,i}^+, w\right\}\left(i = 1,...,N\right)$. Usually after resampling, we will give each of new particles $x_{k,i}^+$ the same weight w for next iterations. For example, if we have N particles, we will allocate

$1/N$ to the weight w of each particle $x_{k,i}^{+}$. Therefore, there is no need to distinguish the weight w for each particle after resampling.

Resampling from $\left\{x_{k,i}^{-}, w_k^i\right\}(i=1,...,N)$ to $\left\{x_{k,i}^{+}, w\right\}(i=1,...,N)$ can be done several different ways. There are many efficient resampling methods which can be implemented such as order statistics, stratified sampling and residual sampling, and systematic resampling. The core of resampling method is to remove particles with small weights and make them substituted by particles with large weights. We can treat the process of resampling as the process of propagation offspring. Particles with large weights will have more offspring in next generation. The more the weight of a particle is, the more offsprings it will have. Their offsprings will have the same position as their parents, which means there will be much more particles overlapping each other. Particles $\left\{x_{k,i}^{+}, w\right\}(i=1,...,N)$ after resampling will have the same weights $1/N$ for all of them.

Now we have a set of *a posteriori* particles with the same weights $\left\{x_{k,i}^{+}, w\right\}(i=1,...,N)$. Next step is to do time propagation for new particles $x_{k,i}^{+}$ based on equation (4.2) to get $x_{k+1,i}^{-}$. Then we will do the same procedures like what we have talked about above to calculate the weights for each particle and do resample etc. Just like this we iterate and iterate to do the same process until the end. Figure 4.1 shows the flowchart of the algorithm of the particle filter method.

4.1.3 The FastSLAM Approach Using the Particle Filter Method

FastSLAM is a factored solution to the simultaneous localization and mapping problem (SLAM). It estimates the posterior distribution over robot path and landmark locations recursively. The algorithm is based on an exact factorization of the posterior into a product of a distribution over robot path and conditional landmark distributions over that distribution. FastSLAM approach is derived on the Bayesian point of view. Figure 4.2 [34] illuminates a generic probabilistic model (dynamic Bayes network) of SLAM, in which $s_1, s_2, ..., s_t$ denote robot poses at different time step, $u_1, u_2, ..., u_t$ are functions of robot controls, $z_1, z_2, ..., z_t$ denoted as the landmark measurements are a function of the position θ_k of the landmark measured and of the robot position at the time that the measurement was obtained. We assume that without

loss of generality landmarks are observed one at a time.

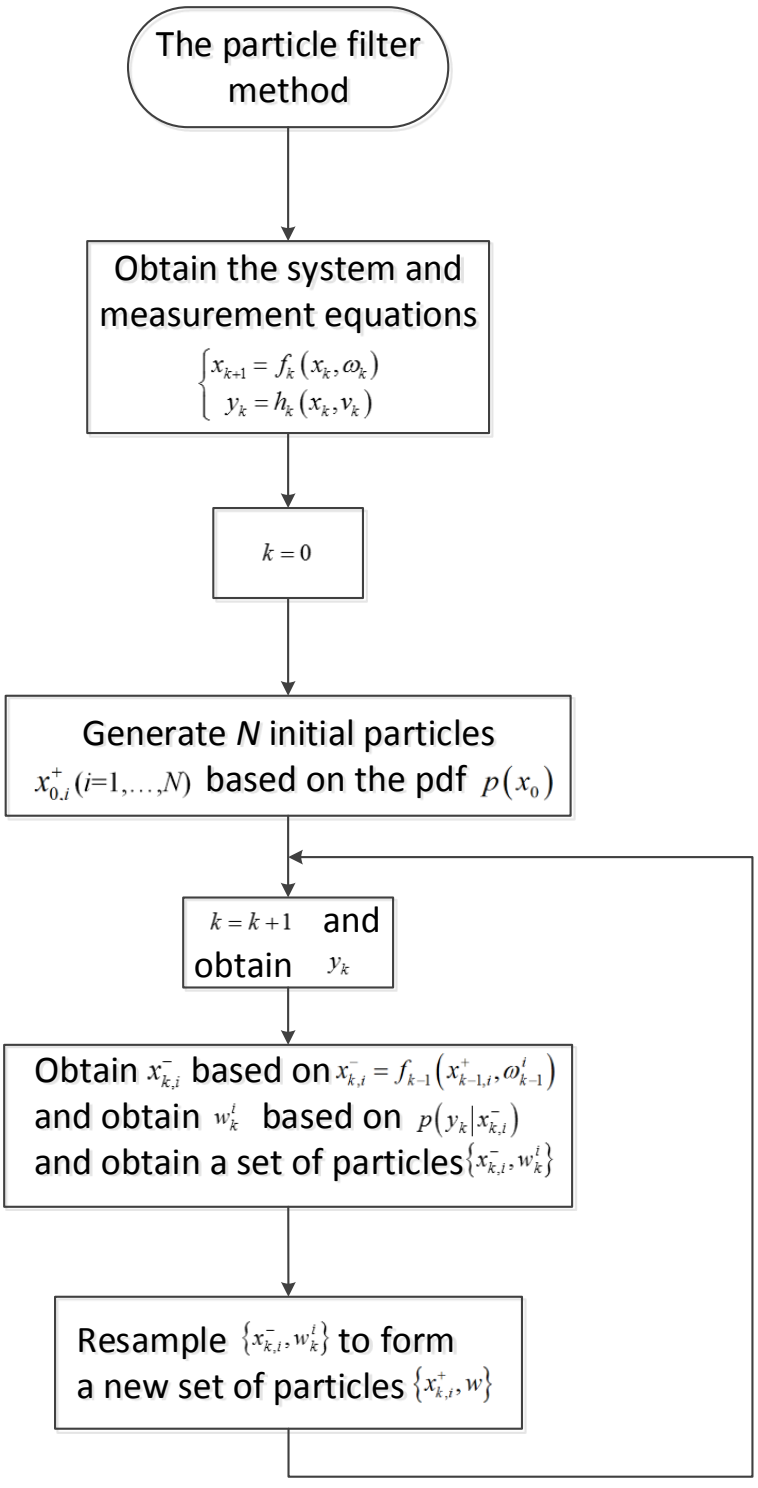


Figure 4.1 The algorithm of the particle filter method.

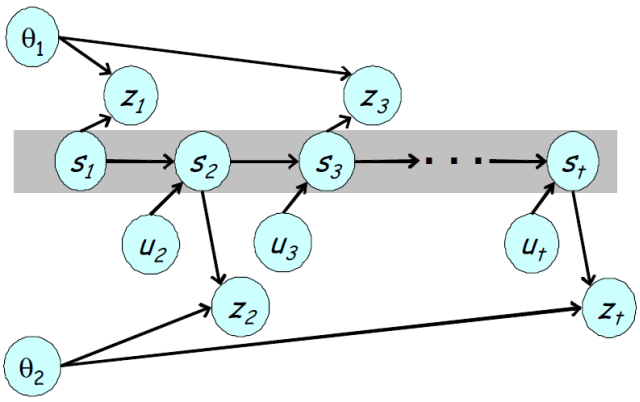


Figure 4.2 The SLAM problem description.

In our research, we consider scenarios with regard to robot navigation with obstacles avoidance, where we treat obstacles as a bunch of landmarks with unknown positions in the global system. At each time step, robot estimates its path based on the observation of one landmark even though it can sense more than one landmark at a time. It is obtained by evaluated the path posterior $p(S_k \mid Y_k, U_k, N_k)$, where Y_k is the measurements $y_1, y_2, ..., y_k$ from robot sensors we will defined in next section, N_k is a set of landmarks $n_l, n_l, ..., n_l$ $(l \in \{1, ...K\})$ perceived from time 1 to time k (where K is the number of landmarks observed), S_k and U_k denote the robot's path (poses) and a set of controls $u_1, u_2, ..., u_k$ respectively.

On the basis of estimation of robot path, we will proceed estimation of landmark location. This process is obtained by evaluating the landmark locations posterior $p(\theta_l \mid S_k, Y_k, U_k, N_k)$. After obtaining these two posteriors, we can use factored representation in order to solve SLAM problem.

$$\begin{aligned} & p(S_k, \theta \mid Y_k, U_k, N_k) \\ & = p(S_k \mid Y_k, U_k, N_k) \prod_{l=1}^{K} p(\theta_l \mid S_k, Y_k, U_k, N_k) \end{aligned} \tag{4.9}$$

where θ consists of all landmarks $\theta_1, \theta_2, ..., \theta_K$. From Equation (4.9) [34] we can see that this problem can be decomposed into K+1 estimation problems, which are one problem of estimating robot paths S_k, and K problems of estimating posteriors over the locations of the K landmarks conditioned on the robot path estimate. Like what we have talked about, we will focus on the particle filter method in order to obtain this.

4.2 Robot Navigation Using the FastSLAM Approach

4.2.1 Obtaining the System Model

Section 4.1 introduces the generic issues regarding to the FastSLAM approach and the particle filter method. In this section, we propose robot tracking with particle filter method. When robot detects the obstacles, a map of local environment can be built based on data received from bearing sensors of robot shown in Figure 4.3. The robot can find the minimum distance to obstacles from one specific beam. The coordinate of the intersection point of that beam with obstacles could be obtained. This intersection point can be considered as a landmark. Since, for knowing the position of the landmark, two measurements with regard to that intersection point are enough to determine the position of the robot properly, in which, one is the distance $d_{\min}$ from the robot to the intersection point shown in Figure 4.3, and the other is the angle α about the chosen beam with the positive direction of x axis also illustrated in Figure 4.3.

We choose these two measurements as the measurement states

$$y_k = \left[d_{\min,k}, \alpha_k \right]^T \tag{4.10}$$

where k indicates the measurements are taken by time k. The obstacle robot detected is treated as a bunch of landmarks in order to get measurements.

We use the coordinate (at time k) of the robot as the system states

$$x_k = \left[x_k, y_k \right]^T \tag{4.11}$$

Based on equation (4.1), we obtain a hidden Markov model (HMM) or general state space model (SSM) as

$$\begin{cases} x_{k+1} = f_k\left(x_k, u_k, \omega_k\right) \\ y_k = h_k\left(x_k, u_k, v_k\right) \end{cases} \tag{4.12}$$

where u_k is the control vectors of the system. We indicate the coordinate of the landmark of the chosen beam and the obstacle as

$$p_k = \left[x_k^*, y_k^* \right]^T \tag{4.13}$$

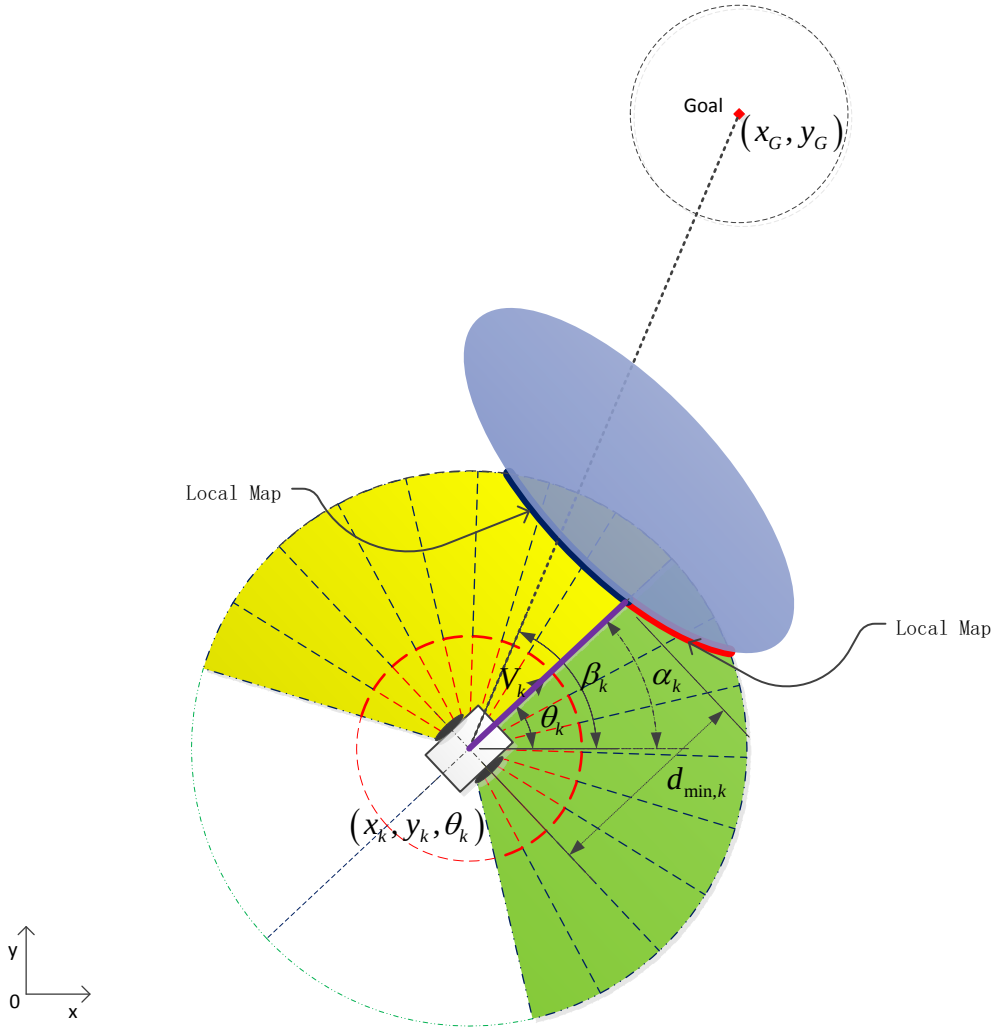


Figure 4.3 The local map robot built when sensing obstacles.

Based on the algorithm of obstacle avoidance using velocity potential field approach, we can expand equation (4.12) as [46]

$$\begin{cases} x_{k+1} = x_k + T * \left[\begin{array}{l} k_a v_{\max} \left(1 - e^{-\frac{\sqrt{(x_G - x_k)^2 + (y_G - y_k)^2}}{R_{goal}}} \right) \cos(\theta_k \pm \omega_{\max} * T) + k_n \frac{1}{d_{\min}} * e^{-\frac{d_{\min}}{R_{safe}}} * \cos(\alpha_k + \pi) + k_t \frac{1}{d_{\min}} * e^{-\frac{d_{\min}}{R_{safe}}} * \cos\left(\alpha_k + \pi \pm \frac{\pi}{2}\right) \\ k_a v_{\max} \left(1 - e^{-\frac{\sqrt{(x_G - x_k)^2 + (y_G - y_k)^2}}{R_{goal}}} \right) \sin(\theta_k \pm \omega_{\max} * T) + k_n \frac{1}{d_{\min}} * e^{-\frac{d_{\min}}{R_{safe}}} * \sin(\alpha_k + \pi) + k_t \frac{1}{d_{\min}} * e^{-\frac{d_{\min}}{R_{safe}}} * \sin\left(\alpha_k + \pi \pm \frac{\pi}{2}\right) \end{array} \right] + \omega_k \\ y_k = \left[\begin{array}{c} \sqrt{(x_k - x_k^*)^2 + (y_k - y_k^*)^2} \\ \arctan\left(\frac{y_k^* - y_k}{x_k^* - x_k} \right) \end{array} \right] + v_k \end{cases} \quad (4.14)$$

where the first "$\pm$" before $\omega_{\max}$ is determined by

$$\begin{cases} +, & \left(\left(\beta_k > \theta_k\right) \text{and} \left|\beta_k - \theta_k\right| < \pi\right) \text{or} \left(\left(\beta_k < \theta_k\right) \text{and} \left|\beta_k - \theta_k\right| \geq \pi\right) \\ -, & \left(\left(\beta_k > \theta_k\right) \text{and} \left|\beta_k - \theta_k\right| \geq \pi\right) \text{or} \left(\left(\beta_k < \theta_k\right) \text{and} \left|\beta_k - \theta_k\right| < \pi\right) \end{cases} \tag{4.15}$$

and the second "$\pm$" before $\frac{\pi}{2}$ is determined by

$$\begin{cases} +, & \text{left obstacles dimensions} \geq \text{right obstacles dimensions} \\ -, & \text{left obstacles dimensions} < \text{right obstacles dimensions} \end{cases} \tag{4.16}$$

In equation (4.14), T is the time step for robot control. k_a, k_n, k_t are gains for attractive, normal repulsive, tangential repulsive velocity, respectively. R_{goal} is the radius for goal, when robot enters into that region it will decrease speed. R_{safe} is the radius for obstacles, when the distance from robot to obstacles is less than that radius, robot will be applied to obstacles avoidance algorithm. $v_{\max}$ and $\omega_{\max}$ are the maximum velocity and angular velocity of the robot. $d_{\min}$ is the minimum distance robot detected from it to obstacles. Equation (4.14) is applied for two situations. When $d_{\min}$ does not exist or is larger than R_{safe}, equation (4.14) becomes

$$\begin{cases} x_{k+1} = x_k + T * \begin{bmatrix} k_a v_{\max} \left(1 - e^{-\frac{\sqrt{\left(x_G - x_k\right)^2 + \left(y_G - y_k\right)^2}}{R_{goal}}}\right) \cos\left(\theta_k \pm \omega_{\max} * T\right) \\ k_a v_{\max} \left(1 - e^{-\frac{\sqrt{\left(x_G - x_k\right)^2 + \left(y_G - y_k\right)^2}}{R_{goal}}}\right) \sin\left(\theta_k \pm \omega_{\max} * T\right) \end{bmatrix} + \omega_k \\ y_k = \begin{bmatrix} \sqrt{\left(x_k - x_k^*\right)^2 + \left(y_k - y_k^*\right)^2} \\ \arctan\left(\frac{y_k^* - y_k}{x_k^* - x_k}\right) \end{bmatrix} + v_k \end{cases} \tag{4.17}$$

When $d_{\min}$ is less than or equals to R_{safe}, equation (4.14) becomes

$$\begin{cases} x_{k+1} = x_k + T * \begin{bmatrix} k_n \dfrac{1}{d_{\min}} * e^{-\frac{d_{\min}}{R_{safe}}} * \cos(\alpha_k + \pi) + k_t \dfrac{1}{d_{\min}} * e^{-\frac{d_{\min}}{R_{safe}}} * \cos\left(\alpha_k + \pi \pm \dfrac{\pi}{2}\right) \\ k_n \dfrac{1}{d_{\min}} * e^{-\frac{d_{\min}}{R_{safe}}} * \sin(\alpha_k + \pi) + k_t \dfrac{1}{d_{\min}} * e^{-\frac{d_{\min}}{R_{safe}}} * \sin\left(\alpha_k + \pi \pm \dfrac{\pi}{2}\right) \end{bmatrix} + \omega_k \\ y_k = \begin{bmatrix} \sqrt{\left(x_k - x_k^*\right)^2 + \left(y_k - y_k^*\right)^2} \\ \arctan\left(\dfrac{y_k^* - y_k}{x_k^* - x_k}\right) \end{bmatrix} + v_k \end{cases} \quad (4.18)$$

Equation (4.17) shows that the robot has applied only the attractive velocity, and equation (4.18) indicates that robot has applied only the repulsive velocity. Equation (4.17) gives the trajectories of robot to the goal not considering obstacles, whereas equation (4.18) demonstrates a better path for robot to avoid obstacles. When the visible part of the left obstacles dimensions are larger than right ones, the robot will turn right in order to save energy and time to avoid obstacles, and vice versa. From equations (4.17) and (4.18) we can see $f_k(.)$ is explicit function of time *k* so that this nonlinear system is time-varying.

4.2.2 Estimation of Robot Path and Landmark Positions

After modeling the system, we will employ a particle filter to estimate robot path based on evaluating $p(S_k \mid Y_k, U_k, N_k)$. The robot can detect more than one landmark at each time step depending of the dimension of obstacles. However, we will just consider only one landmark for the estimation of robot path problem at each time step. The initial pose of the robot and the position of goal are considered known, the controls U_k are also known based on the local map robot detected. So we have the robot path theoretically, however, we need to calculate an estimation due to the system noises ω_k.

Here is the particle filter estimation process. The first step is to produce *N* initial particles $x_{0,i}^+$ (i=1,…,N) based on the pdf of $p(x_0)$. Since we already know the initial pose of the robot, we will produce particles right in the initial position of the robot.

Then we make time propagation for all particles based on the first equation of equation set (4.14)

$$x_{k,i}^{-} = f_{k-1}\left(x_{k-1,i}^{+}, u_{k-1}^{i}, \omega_{k-1}^{i}\right) \qquad \left(i = 1,...N\right) \tag{4.19}$$

to obtain a set of *a priori* particles.

Then we compare $h_k\left(x_{k,i}^{-}, u_k^i, v_k^i\right)$ with y_k (i.e. evaluate $p\left(y_k \middle| x_{k,i}^{-}\right)$), and obtain corresponding W_k^i. We mention that since we have two measurements, the weight W_k^i is composed of the product of two other weights as follows

$$W_k^i = W_{k,d}^i * W_{k,\alpha}^i \tag{4.20}$$

where $W_{k,d}^i$ is the weight obtained based on $p\left(y_k(1) \middle| x_{k,i}^{-}\right)$; i.e. we use the pdf of normal distribution to evaluate the weights comparing $h_k\left(x_{k,i}^{-}, u_k^i, v_k^i\right)(1)$ with the measured distance $d_{\min,k}\left(i.e.\ y_k(1)\right)$. The more $h_k\left(x_{k,i}^{-}, u_k^i, v_k^i\right)(1)$ is close to $d_{\min,k}$, the bigger the weight of that particle is. Likewise, $W_{k,\alpha}^i$ is the weight got relied on $p\left(y_k(2) \middle| x_{k,i}^{-}\right)$. $h_k\left(x_{k,i}^{-}, u_k^i, v_k^i\right)(2)$ close to α_k will be allocated by higher weight based on the pdf of normal distribution. We prefer particles whose both $h_k\left(x_{k,i}^{-}, u_k^i, v_k^i\right)(1)$ and $h_k\left(x_{k,i}^{-}, u_k^i, v_k^i\right)(2)$ are close to $d_{\min,k}$ and α_k, respectively, which result in larger W_k^i because of larger $W_{k,d}^i$ product larger $W_{k,\alpha}^i$. Only one larger $W_{k,d}^i$ or one larger $W_{k,\alpha}^i$ cannot produce larger W_k^i, which corresponds to our proposal since accurate tracking of the robot need the combination of two parameters (the distance and the angle) together.

After obtaining the overall weights W_k^i for all particles $x_{k,i}^{-}$ (i=1,…,N), we have to normalize them based on equation (4.8) in order to obtain a set of normalized weights $w_k^i\ (i = 1,..., N)$. Until now we formed a set of particles $\left\{x_{k,i}^{-}, w_k^i\right\}(i = 1,..., N)$ preparing for the next step of resampling.

In section 4.1.2, numerous resampling methods were introduced. Here we use the resampling method from [36]. We have N weights w_k^i (i=1,…,N), and firstly we obtain the cumulative sum of the weights. Secondly, we randomly produce N numbers from 0 to 1. Thirdly, we put cumulative sum of the weights behind N random numbers, and get corresponding indices when sorting all the numbers in ascending order. Fourthly, find corresponding indices for all indices we got last step are less than N from all 2N indices. Fifthly,

subtract each integral from 0 to (*N*-1) from the indices we obtained last step. Finally, the indices we got in last step are what we need to keep propagating to the next generation. We can find the corresponding particles based on the indices. These particles $x_{k,i}^{+}$ are *a posteriori* particles after resampling with the same weight *w*. Then these particles are going to be done time propagation based on equation (4.19) to enter next iteration. The same process will be applied to the particles in next iteration.

The particle filtering for robot path estimation works as we have talked about until there are no measurements, since without measurements we cannot get the weight for each particle. Likewise, if robot cannot detect obstacle in the beginning, the particle filter is still not able to work because of no measurements. There is another possible case that robot can find obstacles in the beginning, whereas during the navigation process to the goal there may be some time robot lose obstacles in between, at that time particle filter is not going to work until robot can find obstacles again when particle filter resume work once again.

After obtaining new set of *a posteriori* particles though resampling, we can compute any desired statistical measure of this set of particles. We typically are most interested in evaluating the mean and the covariance of all these particles.

Since we already have the estimation of robot path S_k, we will do the estimation of landmark positions conditioned on the estimated robot path based on the measurements, which is evaluating $p(\theta_l \mid S_k, Y_k, U_k, N_k)$. For each robot estimated position, we apply the measurement without noise, then we can obtain the estimation of landmark we got the corresponding measurement about. Since each measurement consisted of the true measurement and the noise, we use the true measurement without noise to better estimate the position of landmark. If one landmark was taken several measurements, Kalman filter has to be applied to get optimal estimation about that landmark based on all measurements about it. Whereas in our research, we assume each landmark is only taken one measurement in the whole process, and we do not apply Kalman filter to estimation of landmark.

The overall algorithm regarding to robot navigation using the FastSLAM approach is shown in Figure 4.4.

4.3 Robot Navigation with Obstacles Avoidance Using the FastSLAM Approach

In last section, robot navigation with the FastSLAM approach is implemented with the help of particle filter. Recalling in Chap. 3, robot navigation with obstacle avoidance is proposed using velocity potential field method, which will take effect on the control vectors of the system u_k and thus make an influence on robot path.

Now we combine these two algorithms together in order to form a robot navigation algorithm with obstacles avoidance using the FastSLAM approach. Figure 4.5 demonstrates the flowchart of the main function of robot navigation algorithm with obstacles avoidance using the FastSLAM approach. The algorithms regarding to robot navigation using the FastSLAM approach without obstacles collision using velocity potential fields method are illustrated in Figure 4.6, in which sub function *Robot calculations* is shown in Fig. 3.9 (b) and sub function *Calculate new robot pose* is defined in Fig. 3.11 (b).

We do not use sub function *Robot navigation using the FastSLAM approach* in new sub function *Robot navigation with obstacles avoidance using the FastSLAM approach* because we just need one execution not a recursive loop which is applied in sub function *Robot navigation using the FastSLAM approach*. The recursive loop is built in the main function. Therefore, we slightly change the scheme of robot navigation using the FastSLAM approach part to produce a whole structure of the algorithms.

The overall process is the following. In the beginning, we do some initialization, plot the position of goal and obstacles, since we have known the initial position of robot, we will produce all *N* initial particles just in the initial position of robot. Certain inputs about the initial robot pose and goal position are provided. Afterwards, we check whether the robot reaches the goal. If yes, exit the main function and stop; otherwise, proceed to the main calculation of the algorithm.

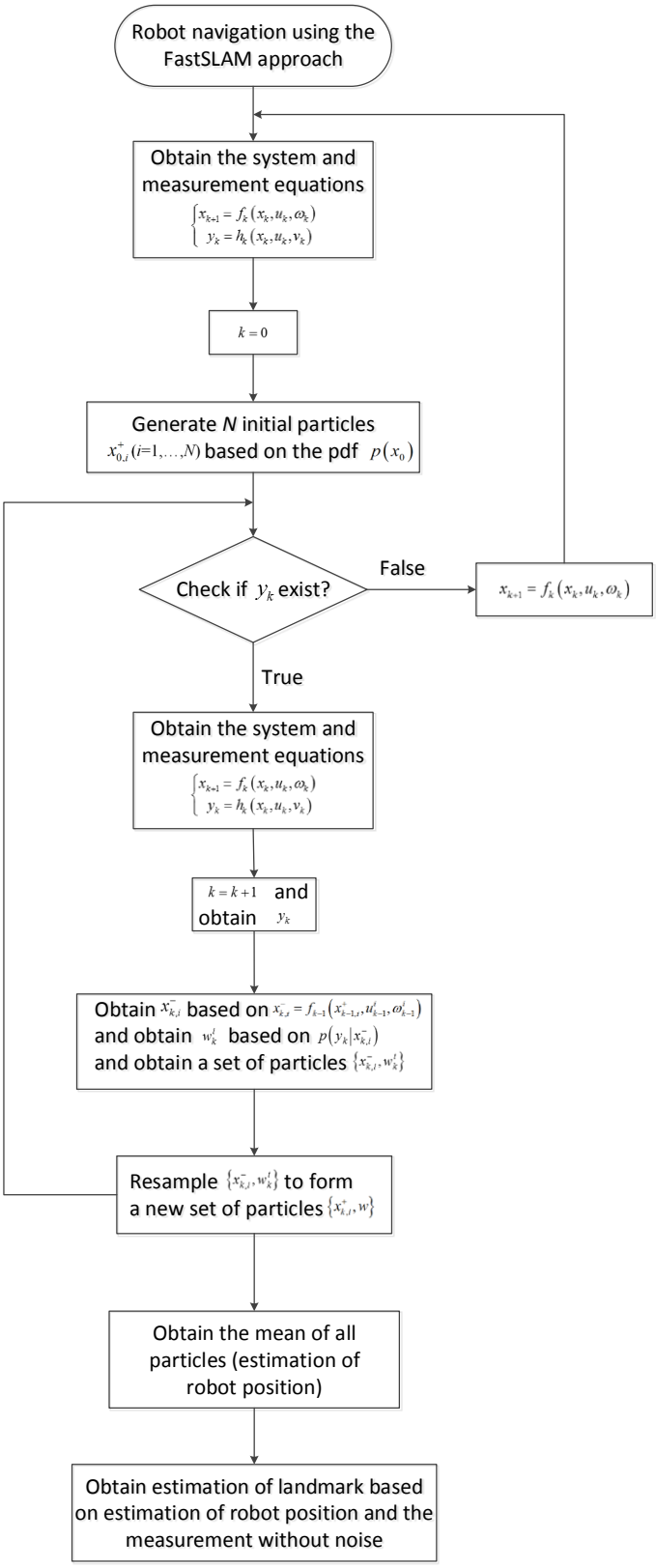


Figure 4.4 The algorithm of robot navigation using the FastSLAM approach.

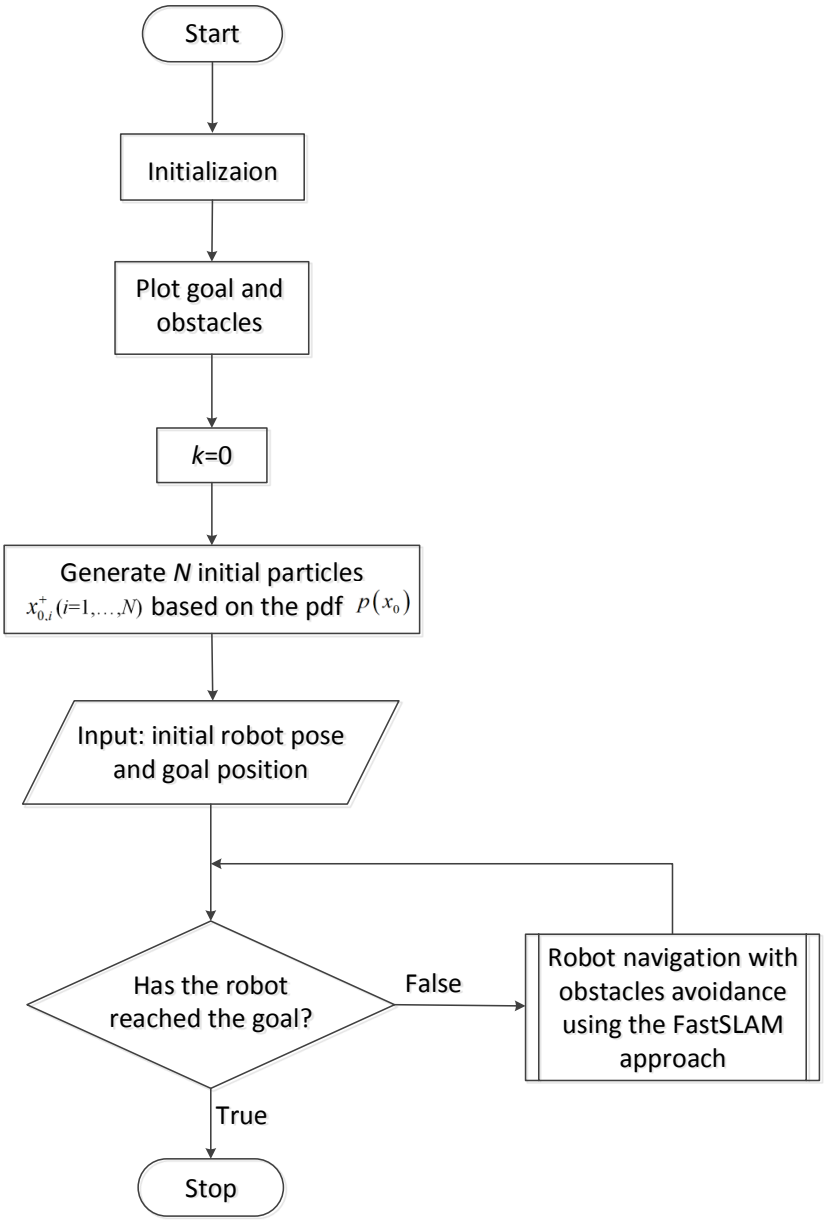


Figure 4.5 Main function of robot navigation algorithm with obstacles avoidance using the FastSLAM approach.

In the sub function *Robot navigation with obstacles avoidance using the FastSLAM approach*, we need obtain goal and obstacles again in order to do some calculations later. Then execute a decision making process of checking if robot detects obstacles. If the answer is no, sub function *Calculate new robot pose* is applied since there are no obstacles detected so that obstacle avoidance approach is not going to be implemented along with the FastSLAM algorithm as of no measurements.

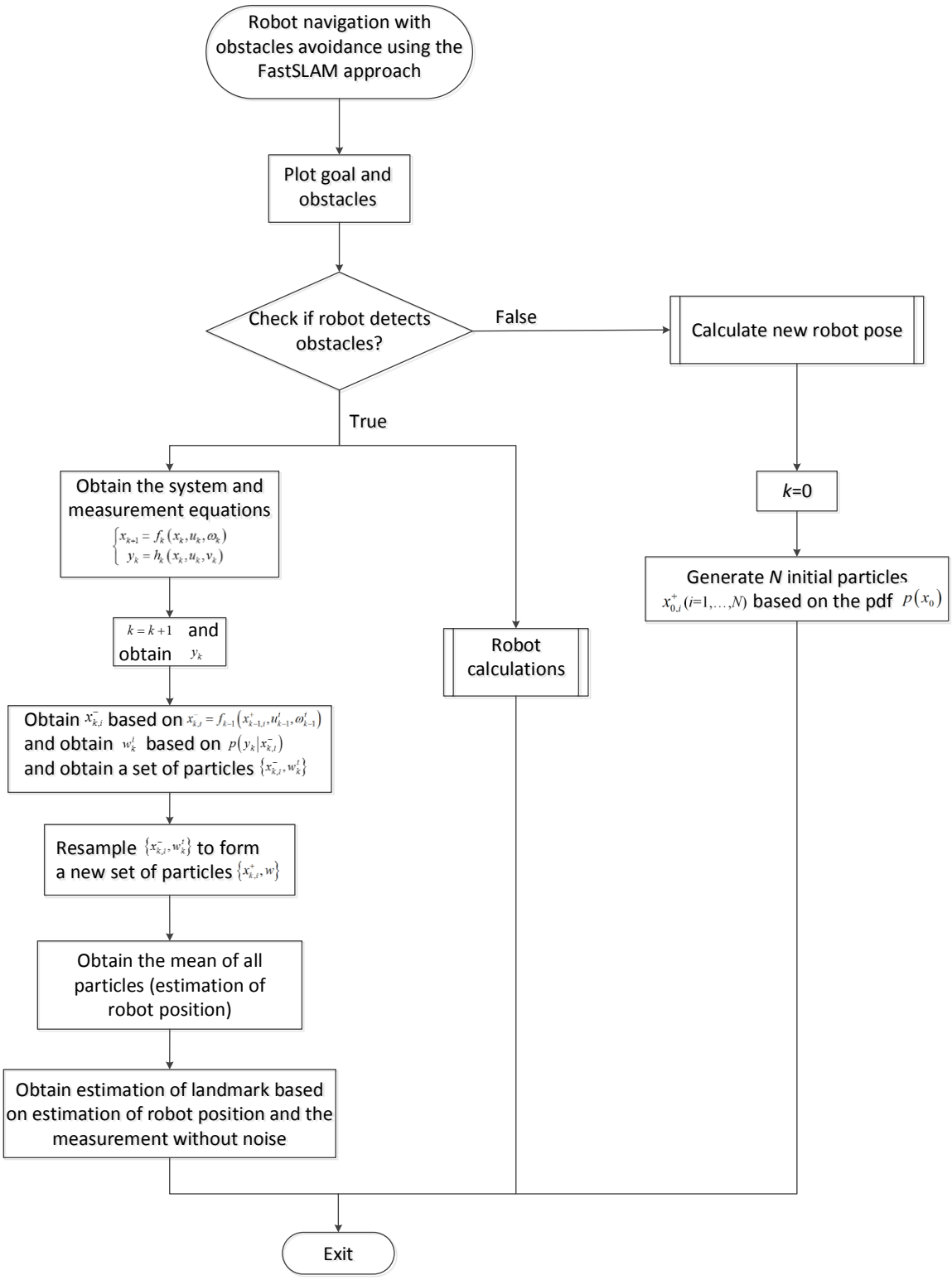


Figure 4.6 Robot navigation with obstacles avoidance using the FastSLAM approach function.

The robot just travels to the goal based on attractive velocity components. We need to reproduce N particles after that because in some cases, the robot detected the obstacles in the beginning, then lost obstacles and finally caught obstacles again. During this process, the particle filter will resume work after a break. After obtaining estimations of robot position and landmark, we exit the sub function and return the main function to check if the robot reaches the goal. If not reaching the goal, enter the sub function again until robot reaching the goal.

In the sub function *Robot navigation with obstacles avoidance using the FastSLAM approach*, if robot finds obstacles, obstacle avoidance algorithm and the FastSLAM algorithm will start working in the same time. Attention is required when the robot detects obstacles, because obstacle avoidance may not be implemented for the robot only if the distance from the obstacles and the robot is close enough (less than the safety radius). However, the FastSLAM part of the algorithm starts working anyway. We have mentioned that the system equation of our model is time-varying and actually it is determined by the controller of obstacles avoidance algorithm. Hence, for each time step, the system equation may change based on the local map robot produced when finding obstacles. That is why the system equation is obtained in every inner loop. When finishing both algorithms, we exit the sub function returning to the main function checking if the robot reaches the goal.

Chapter 5
Description of Experimental Setup

LEGO MINDSTORMS NXT is a programmable robotics kit released by LEGO in late July 2006, which is the flagship product of The Lego Group, a privately held company based in Billund, Denmark [39]. LEGO MINDSTORMS NXT 2.0 is a new version of set released on August 1, 2009, which is what we used in our research featuring some updated capabilities. This new set contains 619 pieces, including sensors, motors, and other bricks.

An advantage of LEGO MINDSTORMS NXT 2.0 turns to be the flexible mechanical designs of the robot, one can build its distinct mechanical structure based on what he/she needs. This merit of due to the classic LEGO BRICK design. In the meantime, one can imbed various sensors into the robot from the requirement of one's experiments, this can decrease the redundant work of loading the drivers for all sensors if we do not need so many other kind of sensors.

LEGO MINDSTORMS NXT 2.0 can be manipulated by many means, it provides a friendly programming environment for researchers, students, and any other robot lovers. We use LabVIEW which is a command box programming environment to program the robot. However, LEGO MINDSTORMS NXT 2.0 also support code programming such as using Visual Basic, C#.

This chapter mainly demonstrates how we set up our experiment. Section 5.1 introduces the CPU of a LEGO MINDSTORMS robot, the NXT intelligent brick. Section 5.2 gives a brief introduction about the sensors we chose and utilized in our experiments. Section 5.3 describes the actuators (the servo motors) mounted in the robot, and how these motors are set on the robot. In section 5.4, we briefly introduce the communication setup between the NXT robot and a PC. Section 5.5 shows the mechanical configuration of the NXT robot we used.

5.1 The NXT Intelligent Brick

The brain of a LEGO MINDSTORMS robot——the NXT intelligent brick is a brick shaped CPU which plays a significant role in controlling the robot. Figure 5.1 and 5.2 [40] give a detailed illustration about the NXT intelligent brick. The NXT intelligent brick has

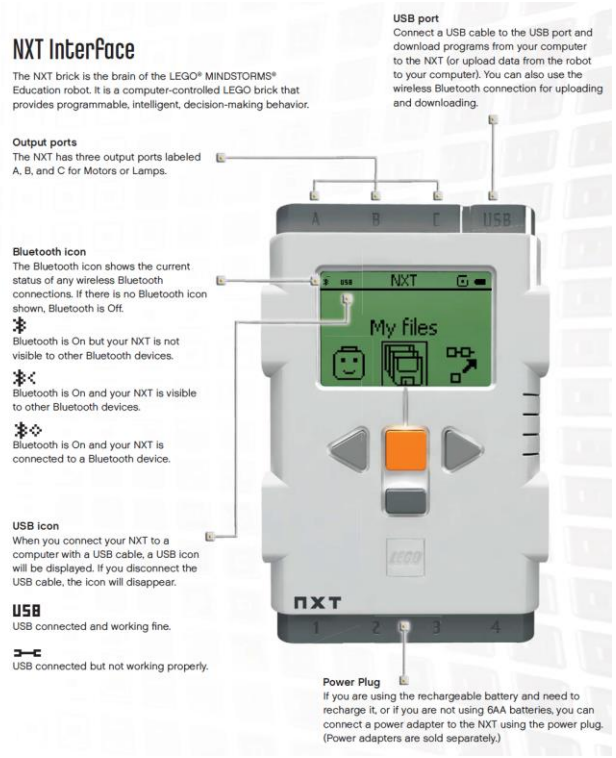


Figure 5.1 Illustration of the NXT intelligent brick 1.

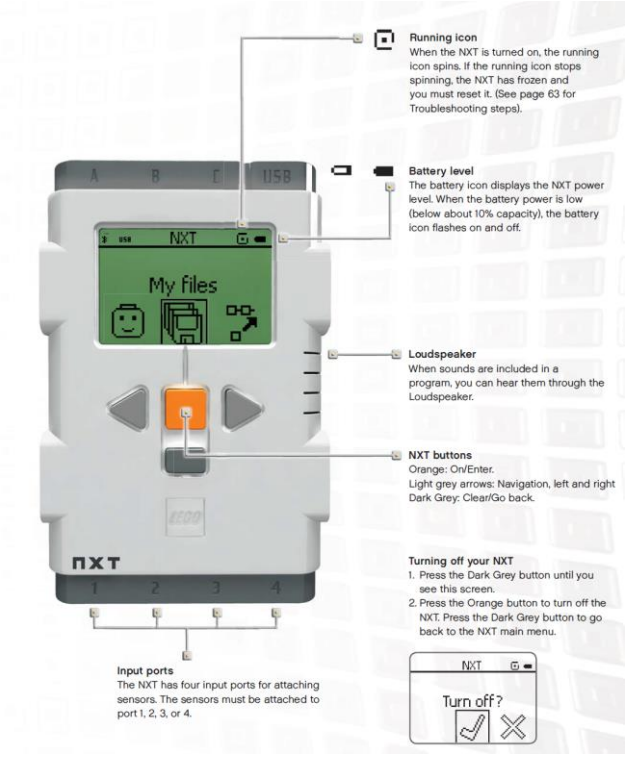


Figure 5.2 Illustration of the NXT intelligent brick 2.

four input ports connected to sensors and three output ports connected with actuators. In our experiment, we need sensors can detect obstacles and measure the distance and bearing to it. The range from robot to obstacle can be obtained by an ultrasonic sensor, and the corresponding bearing can be obtained by a rotation sensor imbedded in a servo motor. We have introduced our robot configuration and its sensing range configuration in Ch. 3. Likewise, we will use similar configuration to our NXT robot. We connect one ultrasonic sensor to the input port, and three servo motors to output ports in which one motor is applied to control the turning ultrasonic sensor. Detailed explanation will be taken in following sections.

5.2 Sensors

Sensors are devices which can sense unknown environment and provide feedback to the output. In our experiment, sensors provide data from outside environment to the NXT intelligent brick, then based on the given controller, the brain of the robot will give commands to output ports (three servo motors).

There are numerous sensors for LEGO NXT robot. The original kit only consist of touch sensor, color sensor and ultrasonic sensor. The LEGO Company then officially offers sound sensor, light sensor, compass sensor, accelerometer sensor etc. Certain third-party companies also produce unofficial sensors for LEGO NXT robot. For example, the HiTechnic Company produces a series of NXT sensors including angle sensor, acceleration/tilt sensor, force sensor, gyro sensor, magnetic sensor, PIR sensor and so on. Another company named The Dexter Industries Company offers thermal infrared sensor, thermometer sensor, pressure sensor, and flex sensor, etc.

In our experiments, 4 sensors are utilized to achieve our objects. Table 5.1 lists details of sensors we use, in which 3 rotation sensors are built in 3 servo motors.

Sensor Name	Quantity Used	Function
Ultrasonic Sensor	1	Measuring Range
Rotation Sensor	3	Measuring Bearing. Built In Servo Motors

Table 5.1 Sensor using condition.

5.2.1 The Ultrasonic Sensor

The ultrasonic sensor is a range sensor which measures the distance between an object in front of the sensor and the sensor itself. The working principle is by sending short beeps and measuring how much time it takes for the beep to bounce back shown in Figure 5.3 [42].

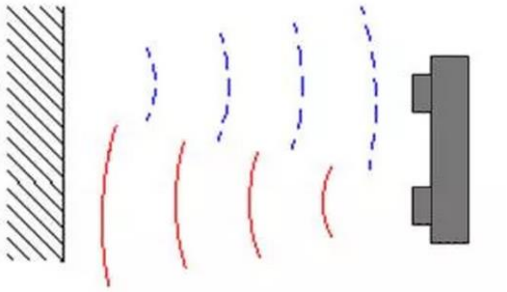


Figure 5.3 The working principle of a ultrasonic sensor.

Since the speed of ultrasonic wave is known in the air medium, which can be obtained by [41]

$$v = 331.5 + 0.6T \tag{5.1}$$

where T denotes the air temperature in degrees Celsius. The distance between the obstacle bounced from and the ultrasonic transmitter d can be calculated obviously by

$$d = \frac{v * t}{2} \tag{5.2}$$

The LEGO ultrasonic sensor is shown in Figure 5.4, in which we can see there is two "eyes" in the front of the sensor. One is the transmitter emitting a sound at a defined frequency (typically around 40 kHz) and the other is the receiver which collects the sound wave reflected back by obstacles.

Figure 5.4 LEGO ultrasonic sensor.

A sound beep does not travel in a straight line like a laser beam does. Instead it spreads out just like the light beam of a light bulb. As a result the sensor not only detects objects that are exactly in front of the sensor, it also detects objects that are somewhat to the left or right [43]. Generally, the LEGO ultrasonic sensor can sense objects that are within an angle 15 degrees to the left or right. Therefore, an ultrasonic sensor is not as accurate as a laser sensor and an infrared sensor, its directivity is around a cone of approximate 30 degrees. Whereas an infrared sensor has a cone of approximate 5 degrees directivity, and the most directional sensor is a laser sensor whose directivity is around one or half a degree. Figure 5.5 [42] shows the ultrasonic sensor intensity distribution versus sensing angle.

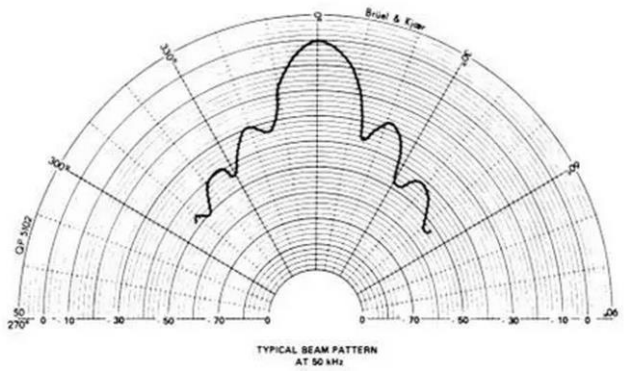


Figure 5.5 Ultrasonic sensor intensity distribution.

The LEGO ultrasonic sensor in Figure 5.4 is a digital sensor, and it returns the distance as a byte value expressed in centimeter (cm). Its detection range is from 0 to 255 cm with a precision of +/-3 cm. The value of 255 means there is no object within measuring range of the robot. Moreover, although theoretically the minimum range of the ultrasonic sensor is 0 cm, in reality the minimum range is about 6 cm. The maximum range depends on the object to be detected, large and hard objects can be detected over a longer range than small and soft objects [43].

The resolution of the ultrasonic sensor is 1 cm, which is not a high accuracy. Mostly the sensor measurement is a few cm of the true distance. If accurate measurements are needed, one has to find out how large the error of the particular sensor is and compensate it [43].

The sensor has five modes of operation, which include off, continuous, ping, capture and reset mode. Only the continuous and ping modes work properly [43]. Default working mode of LEGO NXT robot is continuous mode, in which the sensor does continuous measurements at an interval of approximate 35 msec. So, in our experiment, we cannot rotate the ultrasonic sensor too fast in order to obtain more accurate measurements.

In our experiment, the sensing range is 240 degrees (we have introduced our robot sensing configuration in Chap. 3), which is composed of two 120 degree sensing sections. One is located at the front left part of the robot and the other is mounted on the front right part of the robot. Based on our introduction of directivity of an ultrasonic sensor, one stationary sensor is not enough than our assumption. Also there was only one ultrasonic sensor available in our laboratory.

In order to deal with this issue, we designed a mechanism to provide the ultrasonic sensor fixture with a servo motor built with a rotation sensor in it. This way, the ultrasonic sensor is turned by a servo motor, and we can manipulate the motor with turning angles so that control the direction of the ultrasonic sensor. The mechanical design of the scanning sensor is shown in Figure 5.6, in which we can see that by giving the rotation angle of the rotation sensor, we could change the detecting direction of the ultrasonic sensor.

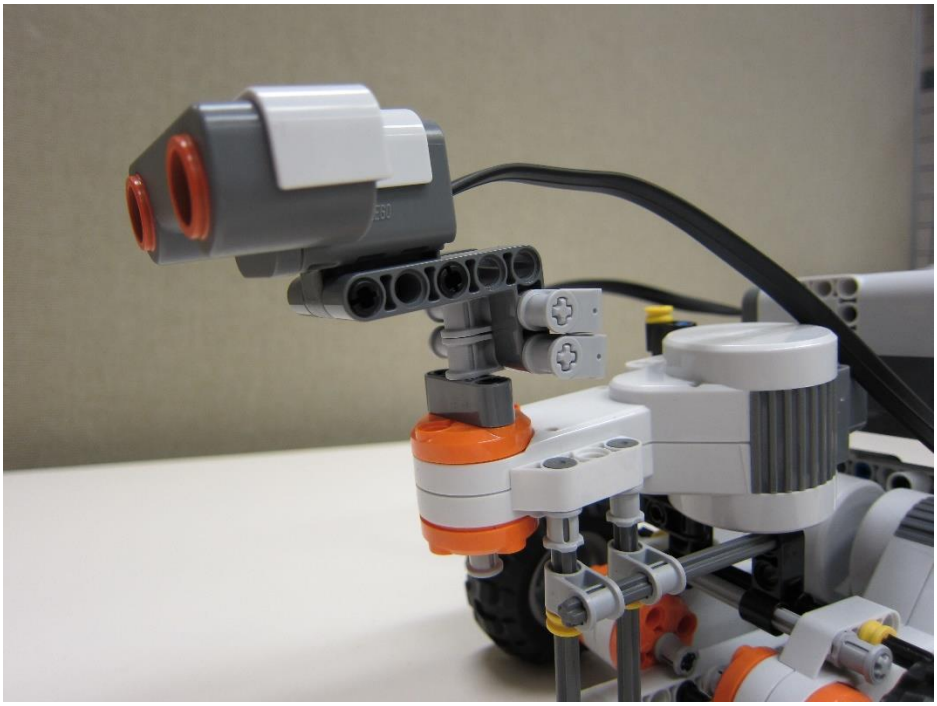


Figure 5.6 Mechanical design of the scanning ultrasonic sensor.

5.2.2 The Rotation Sensor

The rotation sensor is applied to detect the number of turns of an object, rotation angle etc. which is imbedded in the interactive LEGO servo motor. Figure 5.7 illustrates the perspective of NXT servo motor as well as the built-in rotation sensor.

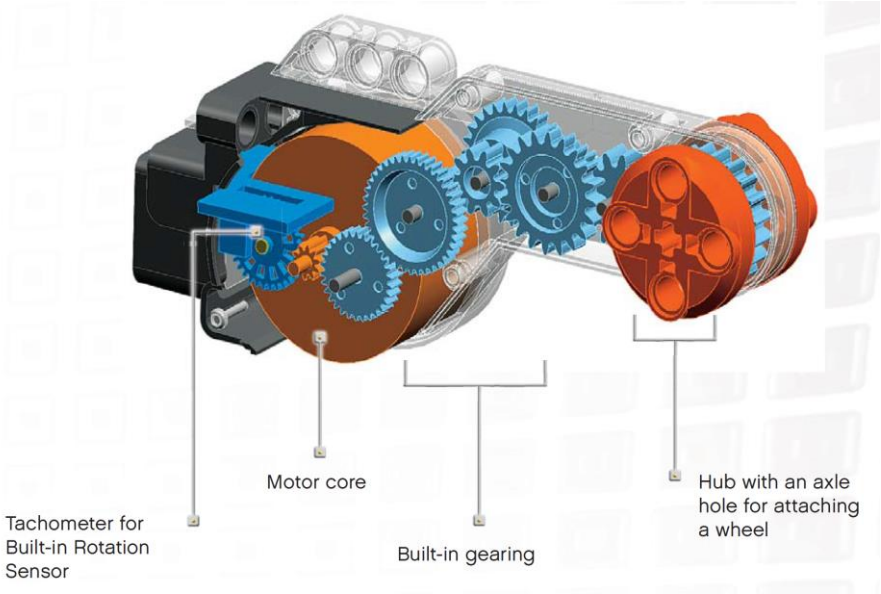
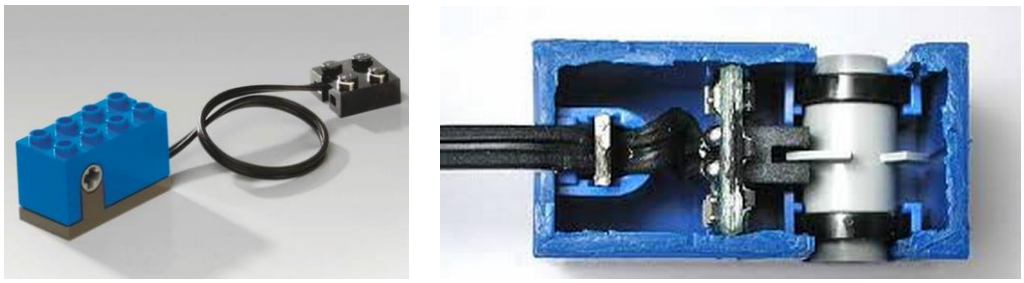


Figure 5.7 The perspective of NXT servo motor.

In Figure 5.7, the left blue small block is the built-in rotation sensor in servo motor, we can dissect it from the motor shown in Figure 5.8. Each servo motor has a built-in rotation sensor. The rotational feedback allows the NXT to control movements very precisely [40]. The embedded in rotation sensor measures the motor rotations in degrees with an accuracy of +/- one degree or in full rotations. Hence the hub will make half a turn if we set the motor to turn 180 degrees.



(a) (b)

Figure 5.8 (a) The built-in rotation sensor; (b) dissection of the rotation sensor.

5.3 Actuators

LEGO NXT robot provides 3 interactive servo motors which can move the robot. The rotation sensor we introduced is contained in the servo motor. A LEGO interactive servo motor,

shown in Figure 5.9. The perspective figure is shown in Figure 5.7, in which the motor is located in the section called “motor core”. A rotation sensor is connected to it by gears in order to count the turning numbers of the motor or record the turning angle of the shaft of the motor. Another set of gears are also connected to the hub of the motor, and the hub with an axle hole for attaching a wheel which can be seen as an orange circle hub in Figure 5.9.

Figure 5.9 LEGO interactive servo motor.

In our experiments, we plug a stick into the hub hole of a servo motor and attach a wheel to the stick. This way, the servo motor can bring power to the wheel in order to turn the wheel. The servo motor is controlled by given a power not a speed, therefore, if the robot needs more traction force to be moved, the speed of the robot will be lower since the power is a product of the pulling force the motor provides and the speed of the motor. We use all the three servo motors in our experiment, two are for providing the power to move the robot and the third one is used for turning the ultrasonic sensor.

Our robot in experiments is manipulated using a differential drive steering model, i.e. by giving different speed of the two driving wheels we control the movement of the robot. Both wheels are controlled independently through two servo motors. The differential drive steering model has certain advantages. For example, the robot with this model can perform various maneuvers such as: moving in a straight line, or in a circular path, spinning in ground and follow prescribed trajectories.

Figure 5.10 illustrates the configuration of all three servo motors. In Figure 5.10, we can see that two parallel servo motors are connected with two wheels, respectively, showing the

differential drive steering model. Based on our mechanical design of our robot, this robot is characterized by nonholonomic kinematic constraints which prevent it from moving sideways, perpendicular to the wheel movement during its motion. Nonholonomic constraints influence the mobility of mobile robots in their workspace; if meeting obstacles in front, they have to move around them in a smooth path with an appropriate velocity since sideway motion is prohibited.

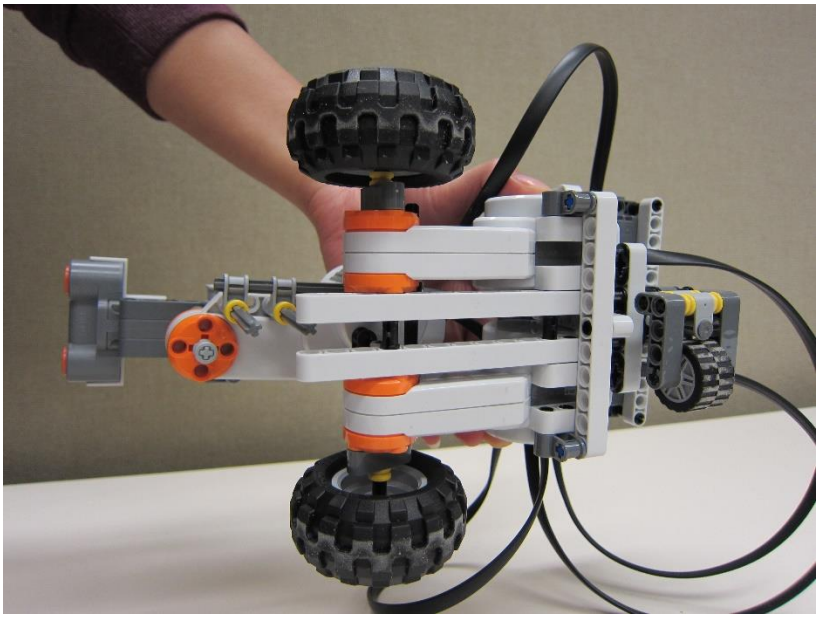


Figure 5.10 The configuration of all three servo motors.

5.4 Networking Setup

LEGO NXT can be connected by a bunch of electronic devices such as cellphone, tablet, and a personal computer (PC) through wired or wireless connections. In our experiment, we control our NXT robot through a PC. Wired connection to a PC can be achieved by a USB cable and Wireless connection can be performed using Bluetooth, which are shown in Figure 5.11.

(a) (b)

Figure 5.11 (a) Wired connection using a USB cable; (b) Wireless connection using Bluetooth.

In Figure 5.11 (a), we can see that in NXT intelligent brick there is a USB port which can be plugged in by a USB cable. If first connecting the NXT robot by a PC, one have to connect it by a USB cable in order to give the firm version of the NXT hardware to a PC. Otherwise, directly connecting the robot to a PC through Bluetooth the first time using it may cause error connecting.

In our experiments, robot will move in an open ground so there is difficult to plug the cable in a PC all the time the experiment is conducted. Thus, we used Bluetooth connection to a PC. Bluetooth is a wireless communication technology which is utilized to send and receive data without using wires. Once the Bluetooth connection is set up, you can use it for these features [40]:

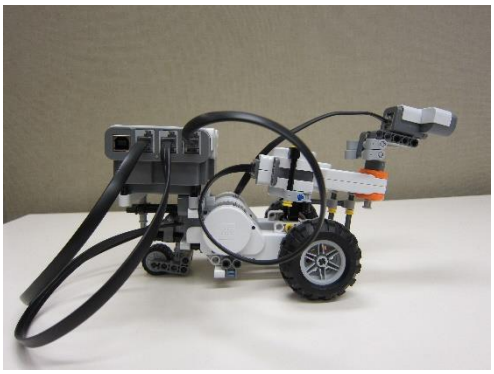
- Downloading programs from your computer without using a USB cable.
- Sending programs from devices other than your computer, including your own NXT.
- Sending programs to various NXT units either individually or in groups. A group can contain up to three NXT devices.

Connecting with Bluetooth can be seen on the LCD screen of the NXT intelligent brick shown in Figure 5.1. Details about how to connect a PC to the NXT with Bluetooth can be found in [40].

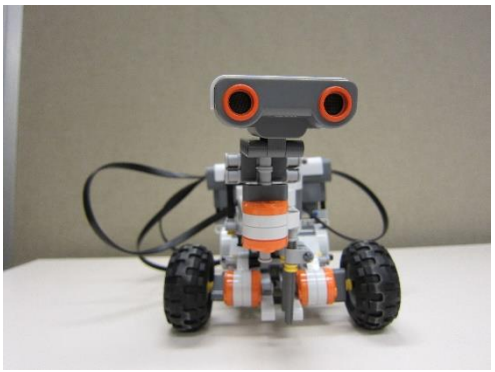
5.5 Mechanical Design of our NXT robot

LEGO NXT robot provides various pieces in order to let hobbyist, students, and researchers to design their own mechanical structure to fulfill their requirements. Our requirement for the experiment is designing a mobile robot using a differential drive steering model with a scanning ultrasonic sensor. The two-wheel differential drive mobile robot performs better than its four or six wheels counterparts in working with its nonholonomic motion constraints, as the two-wheel one can spin in the ground when difficult motions are required.

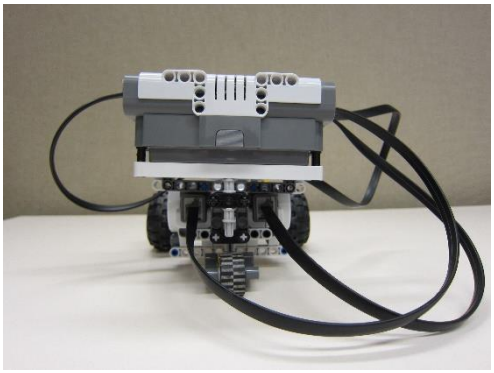
In our design of the robot, except our introduction about three servo motors, we design a castor wheel with one degree of freedom (rotation about an axis perpendicular to the ground). This design of the castor wheel is for providing more stability for the robot. It is mounted in the rear of the robot supporting the robot for better balance performance, and it can rotate in 360 degrees normal to the ground just like an omni-directional wheel. The overview of the robot we used is illustrated in Figure 5.12.



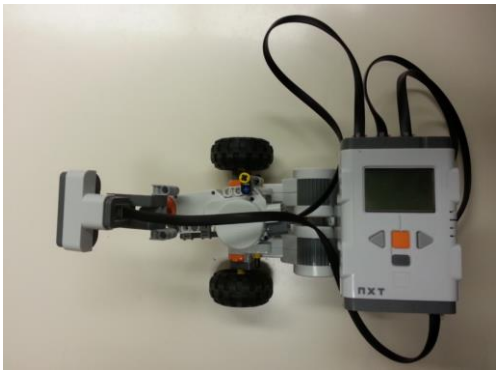
(a)



(b)



(c)



(d)

Figure 5.12 (a) Side view of robot; (b) front view of robot; (c) back view of robot; (d) top view of robot.

Chapter 6
Simulation Results

In this chapter, the simulation tool and the simulation results will be presented. Our simulations are conducted using MATLAB. The corresponding robot navigation with obstacles avoidance algorithm have been the topic of Ch. 3, and the algorithm regarding to the FastSLAM approach was introduced in Chap. 4. Our simulations are relied on these algorithms. Section 6.1 will introduce how we setup our MATLAB simulations including some basic MATLAB simulation elements and our simulation algorithms. Section 6.2 will illustrates all of the simulation results.

6.1 MATLAB Simulation Design

The simulation tool we used is MATLAB 2013a. MATLAB, the abbreviation of matrix laboratory, is a multi-paradigm numerical computing environment and high-level programming language developed by MathWorks. It helps students, researchers and other users develop algorithms, analyze data, and create models and applications. It is a perfect academic application tool and we can implement mathematical algorithm to the program in MATLAB visualizing the corresponding results. For robotic simulations, MATLAB enables fast development and execution of robotic controllers. In our simulations, it helps analyze and implement the feasibility of our navigation algorithms.

Our simulations in MATLAB were created in the MATLAB Editor, which is more convenient to edit, manage our programs and run the programs simultaneously in Command Window, using the MATLAB programming language. In order to verify and clarify our algorithms regarding robot navigation, we conducted simulations in the form of animations.

6.1.1 MATLAB Simulation Elements

MATLAB simulation animations are implemented by running the codes consisting of certain basic elements. In our simulations, the robot will move in a 2D plane which is in the Cartesian coordinate system. A filled arrow field is used to indicate the pose of the robot illustrated in Figure 6.1, in which robot's yaw angle is depicted by the direction of the arrowhead at each time step. Non-holonomic movement of the mobile robot could easily be visualized through the representation of the arrows.

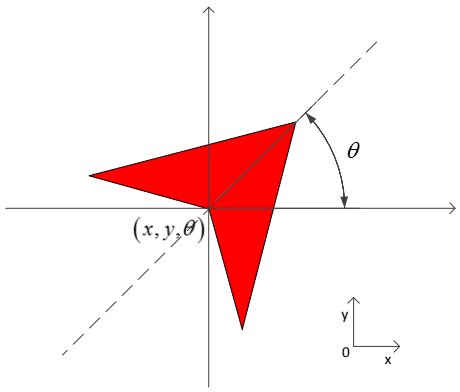


Figure 6.1 An arrow representing robot's pose.

After depicting the pose of the robot, sensors have to be added to the robot model since robot uses them to sense outer environment and provide data to controllers to obtain reactive commands. The robot has a detecting range based on its bearing sensors introduced in Section 3.5. Meantime, the safety range in which robot will apply obstacles avoidance algorithm when obstacles are into it is added to the robot's controller. Sensing configuration of the robot is shown in Figure 6.2.

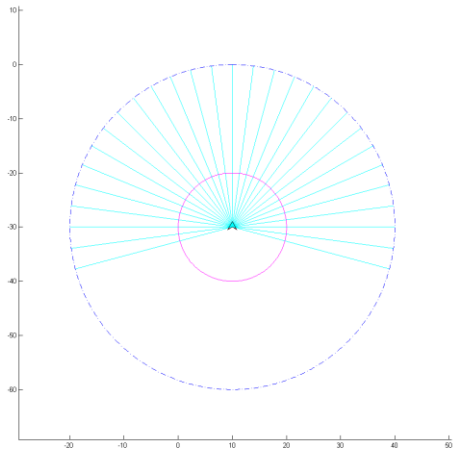
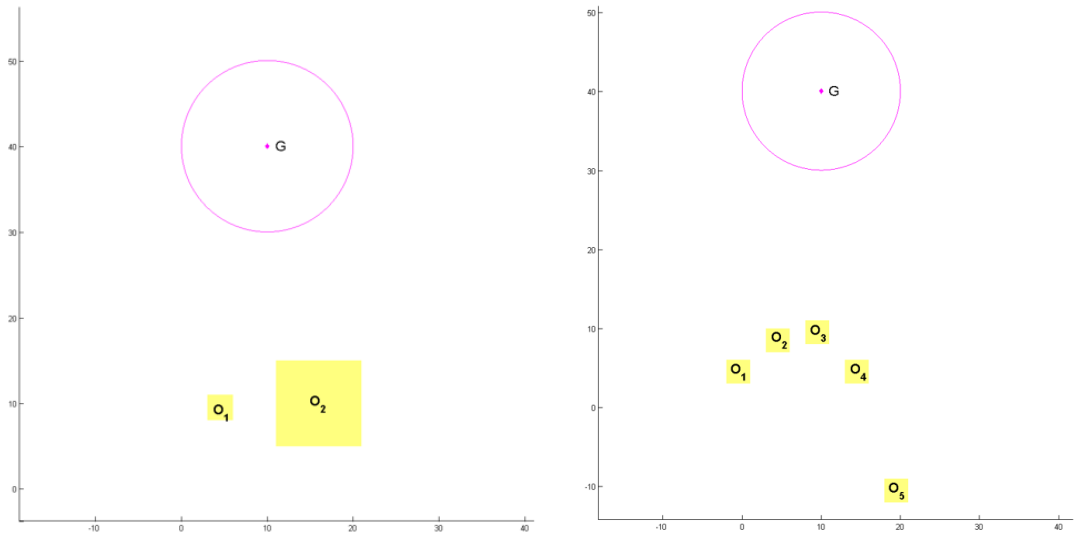


Figure 6.2 Sensing configuration around robot.

In Figure 6.2, the blue dash-dot line indicates the sensing range of robot sensors. As we have presented in Section 3.5, the total sensing range is a 240 degrees fan shaped area. Cyan beams describe sensing beams extending from the center of the robot. Each beam is separated by an angle of $\pi/24$, and all beams are connected to a circle indicating the detecting range of the robot. When obstacles enter into this area, the FastSLAM approach starts working.

There is a magenta solid circle inside the detecting range of the robot, which is called the safety range of the robot, as illustrated in Section 3.5. In case that the robot travels too close to obstacles so that obstacles show up inside the safety range of the robot, obstacle avoidance algorithm will be applied in order to turn the robot around the obstacles.

Obstacles are depicted by geometric shapes with text on them in order to better illustrate them, and the goal is plotted by a magenta diamond. We use squares to represent obstacles with distinct dimensions differences. The obstacles are colored yellow in all of our simulations with some transparency in order to show better possible estimation of obstacles.



(a) (b)

Figure 6.3 (a) Two obstacles with different dimensions; (b) Five obstacles forming a dissymmetric concave obstacle.

In Figure 6.3 (a), capital "G" indicates the goal robot will travel towards, O_1 and O_2 denote two obstacles, of which O_1 has smaller dimension than O_2. In the simulations, robot detects both obstacle O_1 and O_2, builds a local map regarding to left and right sensing section and choose proper turning direction to avoid obstacles when it is too close to them.

Likewise, in Figure 6.3 (b), the goal is also marked by capital "G", and there are five same sized obstacles O_1, O_2, O_3, O_4 and O_5. However, the locations of five obstacle are not symmetric, as shown in Figure 6.3 (b). If we put these obstacles together, they could be considered as a single large dissymmetric concave obstacle, to be avoided as an ensemble. Our controller was designed to lead the robot and find an optimal trajectory to escape the concave trap and reach the goal.

Finally, we need to define several constant constraints in the simulation. For example, we could modify the size of the terrain, sensor beam radius, the maximum velocity and angular velocity of the robot, simulation time and time steps, etc.

6.1.2 MATLAB Simulation Algorithm

The algorithms we proposed have been explained in Chap. 3 and Chap. 4; here we just review them and elaborate more specifically. Our algorithms include two parts, one is robot navigation with obstacles avoidance using proposed artificial velocity fields method and the other is robot navigation using the FastSLAM approach. These algorithms are applied when the robot meets certain criteria, which in our simulations is mainly about the distance from the robot to obstacles or the goal.

For the robot, we defined two radiuses, R_{sensor} and R_{safety}, in which R_{sensor} denotes the sensing radius of robot, and R_{safety} indicates the safety radius of robot into which obstacle avoidance algorithm will be applied if obstacles invade. At the same time, we defined two distances, d_{goal} and $d_{obstacle}$, where d_{goal} represents the distance form robot to the goal, and $d_{obstacle}$ indicates the distance form robot to the closest obstacle to the robot. When the distance from the goal or the closest obstacle to the robot changes, the algorithms applied to the robot are different. Table 6.1 demonstrates algorithms applying conditions along with robot navigation.

In Table 6.1, robot navigation process is divided by five criteria about whether robot senses the goal or the obstacles. There are six combinations regarding to these five criteria. Based on different combinations, we assign different algorithms.

Criteria	$d_{goal} \le R_{sensor}$	$d_{goal} > R_{sensor}$
$d_{obstacle} \le R_{safety}$	The FastSLAM approach wrt the goal and obstacle avoidance algorithm	The FastSLAM approach wrt obstacles and obstacle avoidance algorithm
$R_{safety} < d_{obstacle} \le R_{sensor}$	The FastSLAM approach wrt the goal	The FastSLAM approach wrt obstacles
$d_{obstacle} > R_{sensor}$		_____

Table 6.1 algorithms applying conditions along with robot navigation.

6.2 MATLAB Simulation Results

In the simulations, we consider two scenarios. One is robot navigation in the case of two obstacles with distinct dimension differences, shown in Figure 6.3 (a); the other is robot navigation with five same-dimension obstacles emulating of a large single concave obstacle with dissymmetrical shape, illustrated in Figure 6.3 (b). We got the inspiration from [38], in which dimensions of obstacles are always the same, and the robot will always go from one direction, i.e. the algorithm of obstacles avoidance is not a switching controller. Our simulations are conducted by using a switching controller such that the robot will choose proper turning direction based on the local map it formed.

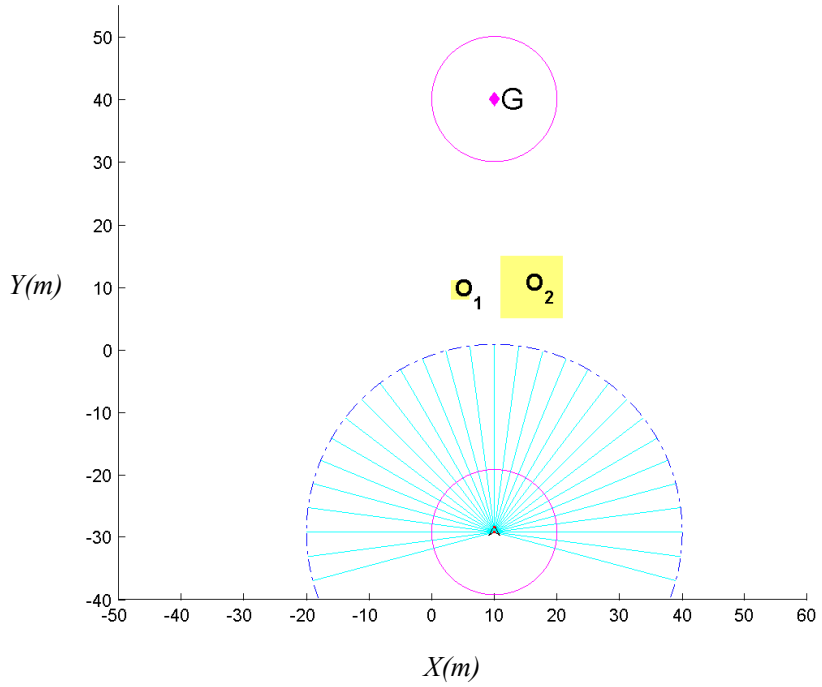
In these simulations, we can see that the robot successfully chose a direction with higher efficiency and time-saving to avoid obstacles and finally reach the goal. In the process, while the robot was travelling towards the goal, the estimations of robot path and the positions of landmarks (obstacles) were obtained.

6.2.1 Robot Travelling Around Two Obstacles

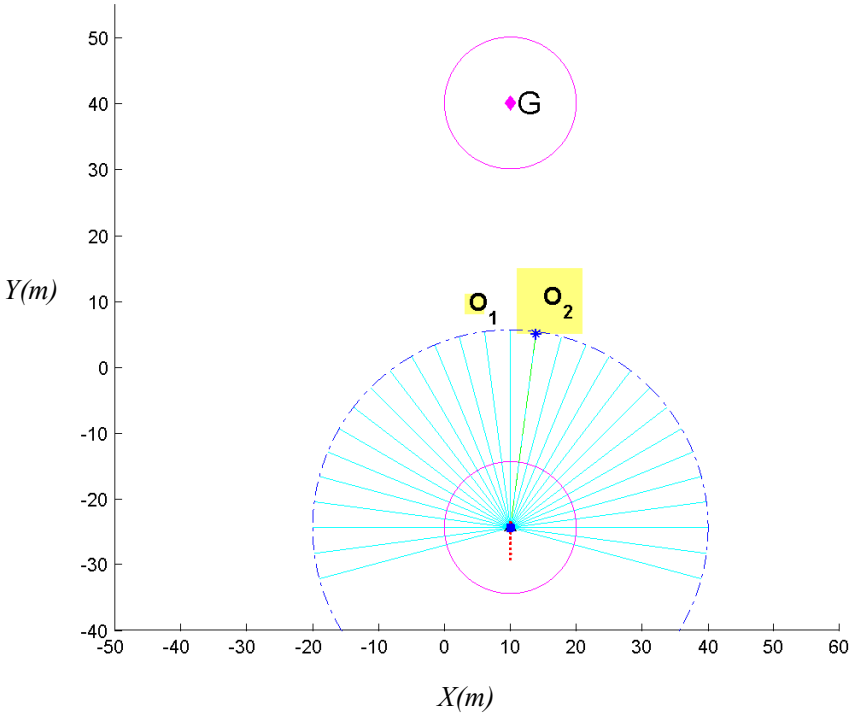
Figure 6.4 (a-j) shows robot navigation avoiding two obstacles using the FastSLAM approach (i.e. obtaining the estimations of robot path and positions of obstacles). In this simulation, the robot built a local map finding that O_2 has larger dimension than O_1, so that it turned left when too close to the obstacles. At the same time, the red dotted line indicates the ideal robot path based on our controller, and the blue square illustrates the estimated robot path; we can also see there are asterisk signs around the obstacles which indicate the estimated positions of obstacles. The obstacle avoidance algorithm is based on the proposed velocity potential fields approach, and the estimations with regard to robot path and obstacles are performed by using the FastSLAM approach.

We introduce in Table 6.2the basic legends about various symbols, which are used in all simulations. In Figure 6.4 (a-j), 10 snapshots are shown regarding robot navigation using the FastSLAM approach with obstacles avoidance. When the robot detected obstacles, estimations of the robot path indicate to measurements about obstacles. However, when robot bypassed obstacles and sensed the goal, the estimation of the robot path is based on measurements wrt the goal only. The estimated robot path and the real robot path converges finally when robot

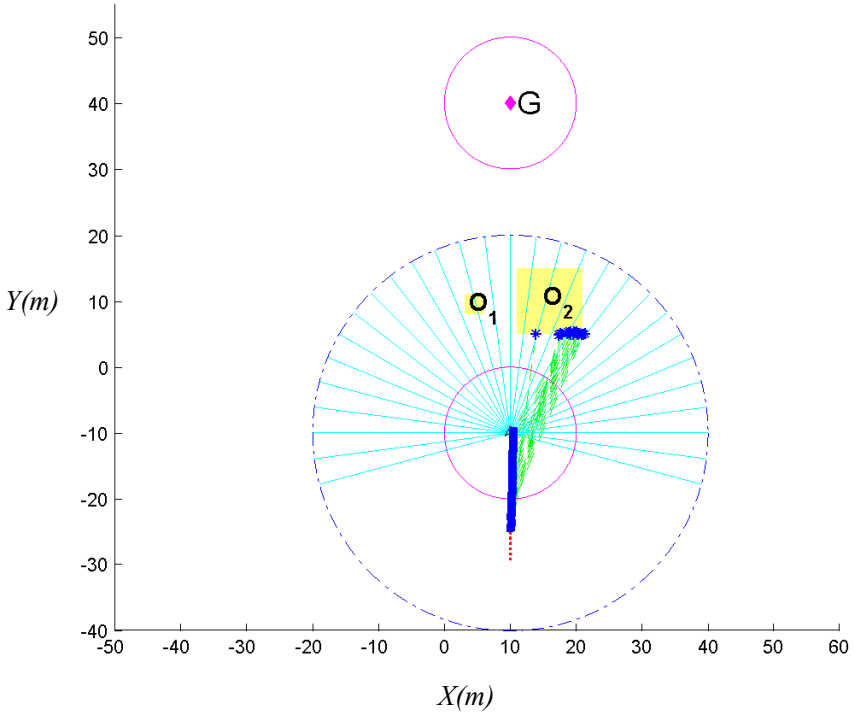
reaches the goal.



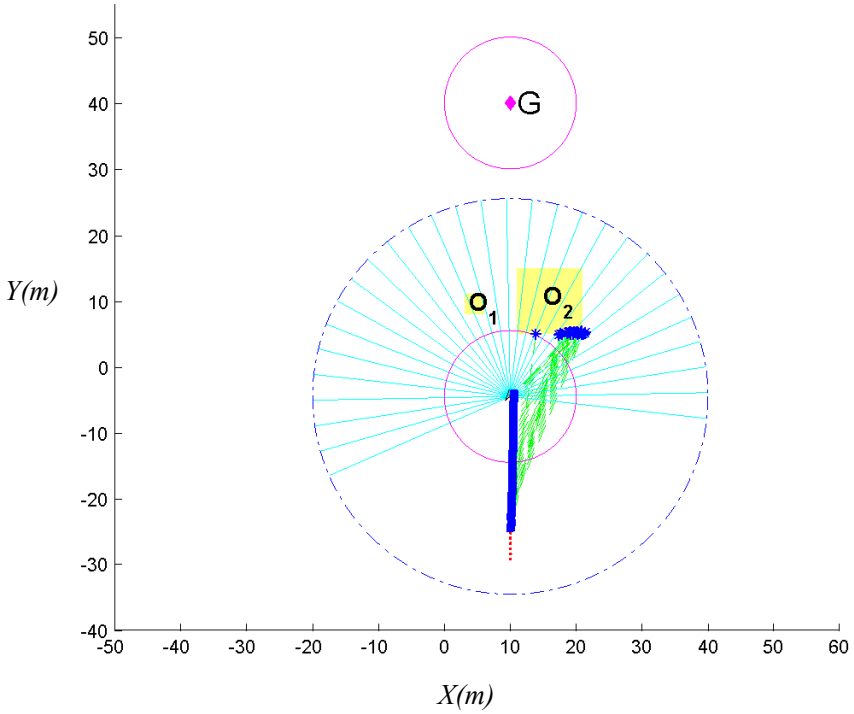
(a)



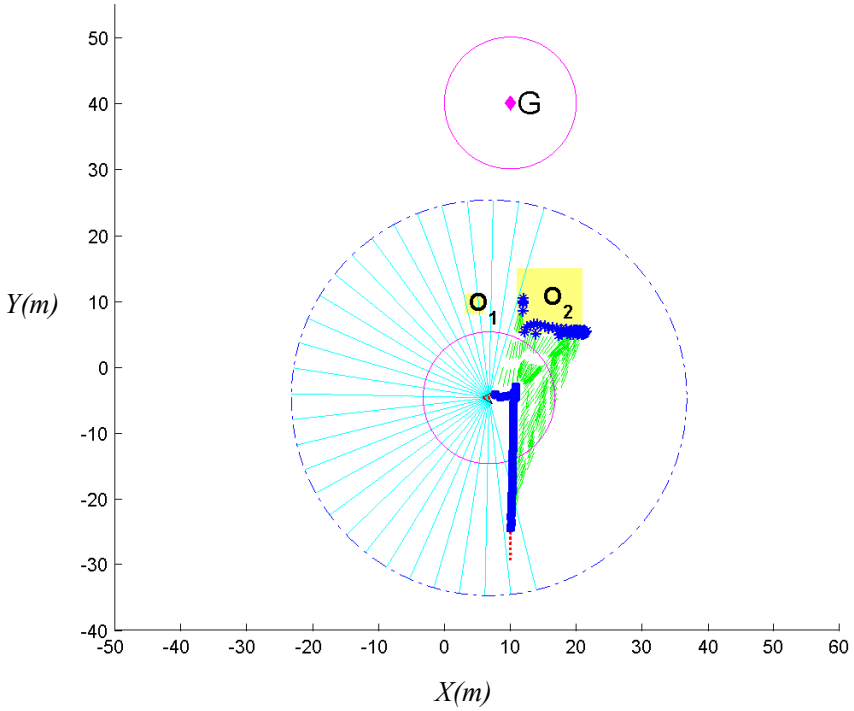
(b)



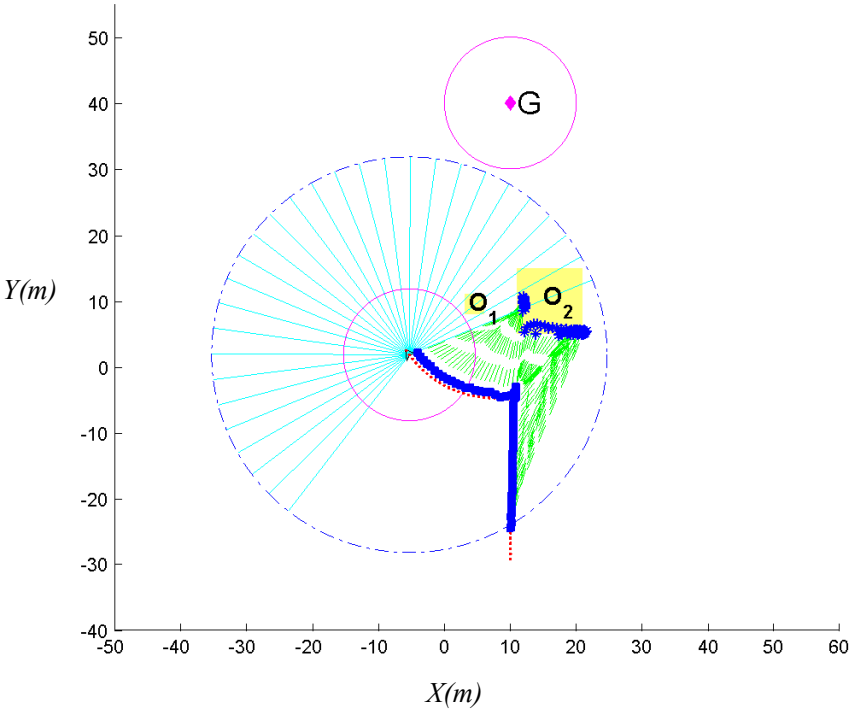
(c)



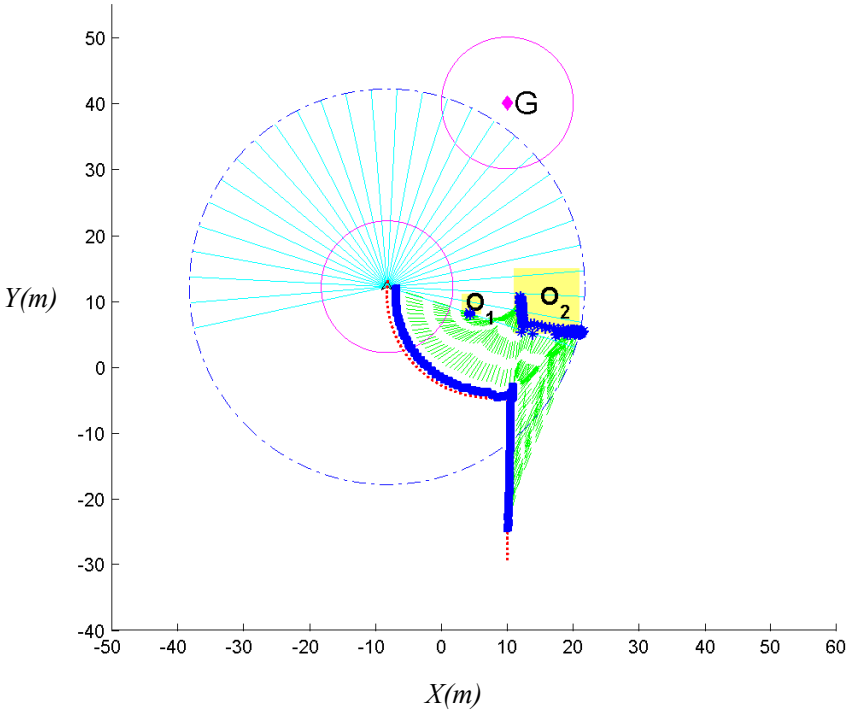
(d)



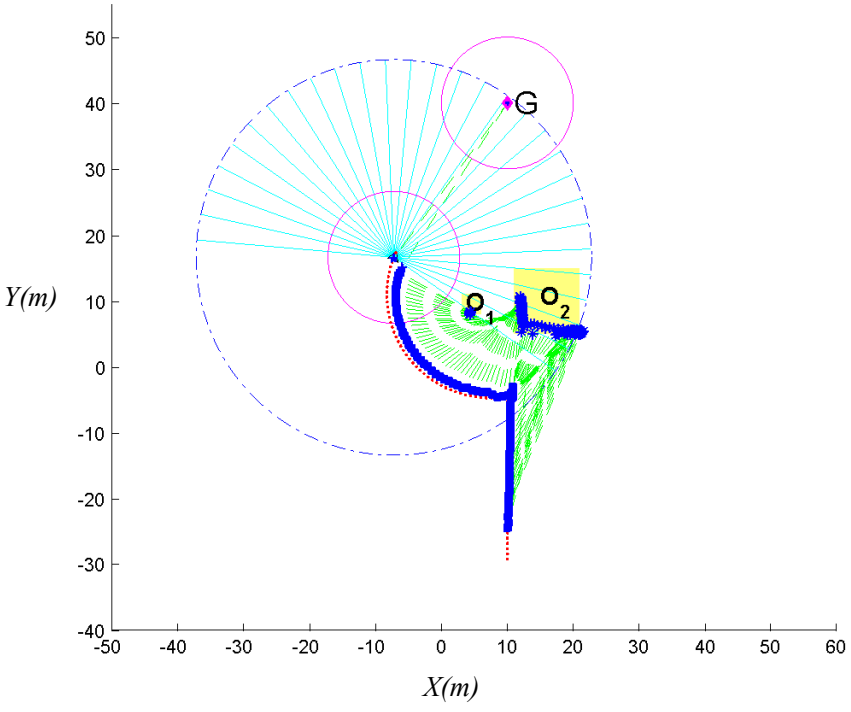
(e)



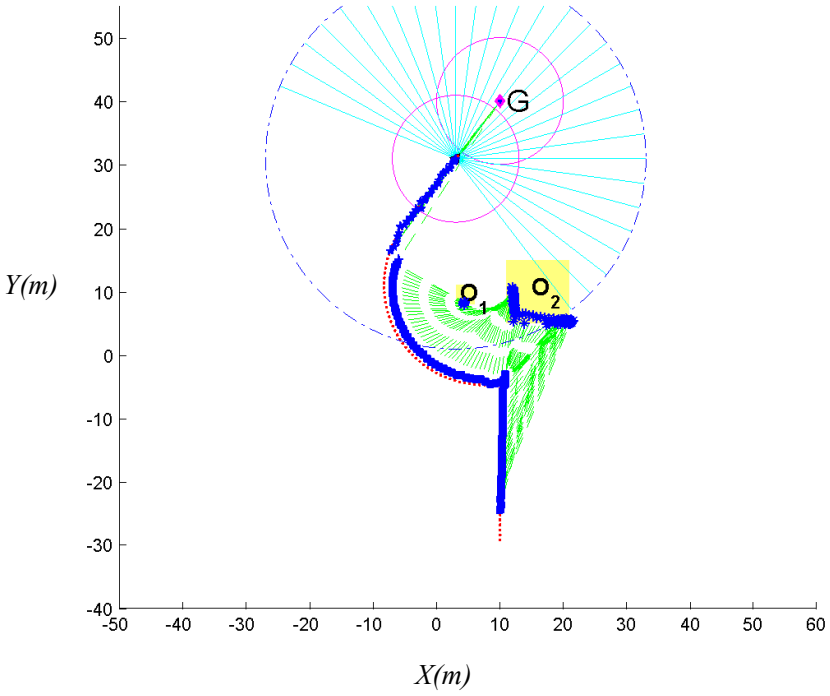
(f)


G
O1
O2
Y(m)
X(m)
50
40
30
20
10
0
-10
-20
-30
-40
-50
-40
-30
-20
-10
0
10
20
30
40
50
60

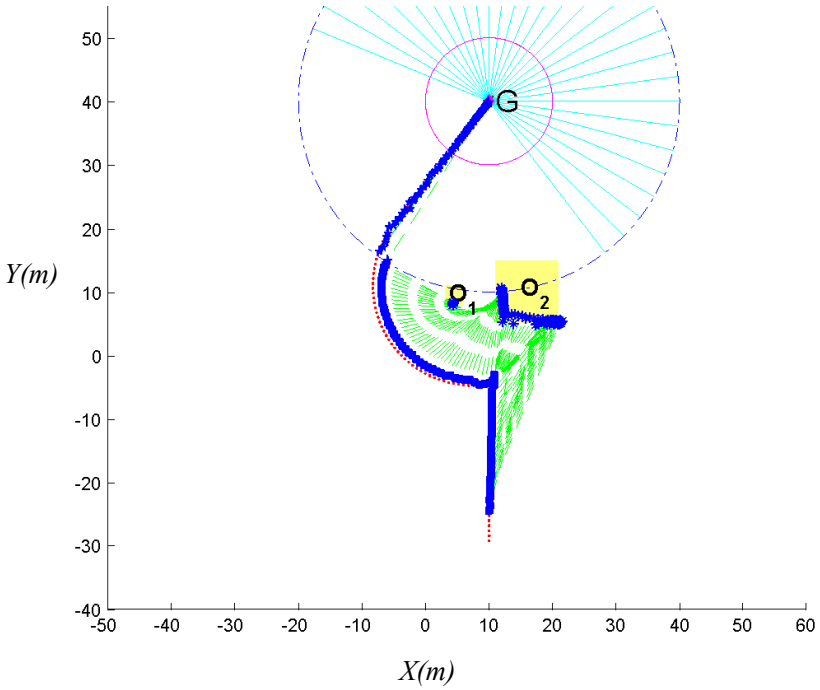
(g)


G
O1
O2
Y(m)
X(m)
50
40
30
20
10
0
-10
-20
-30
-40
-50
-40
-30
-20
-10
0
10
20
30
40
50
60

(h)



(i)



(j)

Figure 6.4 Robot travelling around two obstacles with the FastSLAM approach.

In Figure 6.5, we give particle clouds representations of estimations of robot positions with regard to obstacles. Instead of using a red dotted line as the ideal robot positions, we presented the ideal robot path as a solid line with upward-pointing triangle markers in order to obtain better illustration. Figure 6.6 (a-d) demonstrates the zooming-in figures of Figure 6.5 as we want to present a detailed illustration. Figure 6.6 (a) indicates that when robot first detected obstacles, the FastSLAM algorithm starts working. Afterwards, robot travelled too close to obstacles, and had to turn, as shown in Figure 6.6 (b). Figure 6.6 (c) illustrates the process of robot bypassing obstacles. And finally, when robot bypassed obstacles and sensed the goal, estimation about obstacles ended which is shown in Figure 6.6 (d).

Likewise, Figure 6.7 describes estimations of robot positions with regard to the goal using particle clouds representations. The zooming-in figures of Figure 6.7 are illustrated in Figure 6.8 (a-d). Figure 6.8 (a) denotes when robot just sensed the goal first time, estimations of robot positions about obstacles changed to estimations wrt the goal. Figure 6.8 (b-d) shows the particles distribution along with robot moving towards the goal.

Symbols	*Meanings*
(The magenta diamond)	The goal
(The magenta circle around the goal)	The area of the goal
(The yellow square with text on it)	The obstacle with the number of it
(The red dotted line)	Ideal robot path
(The blue square)	Estimated robot position wrt obstacles
(The blue asterisk)	Estimated obstacle position
(The blue pentagram)	Estimated robot position wrt the goal
(The green dashed line)	Estimated measurement line

Table 6.2 Legend for our simulations.

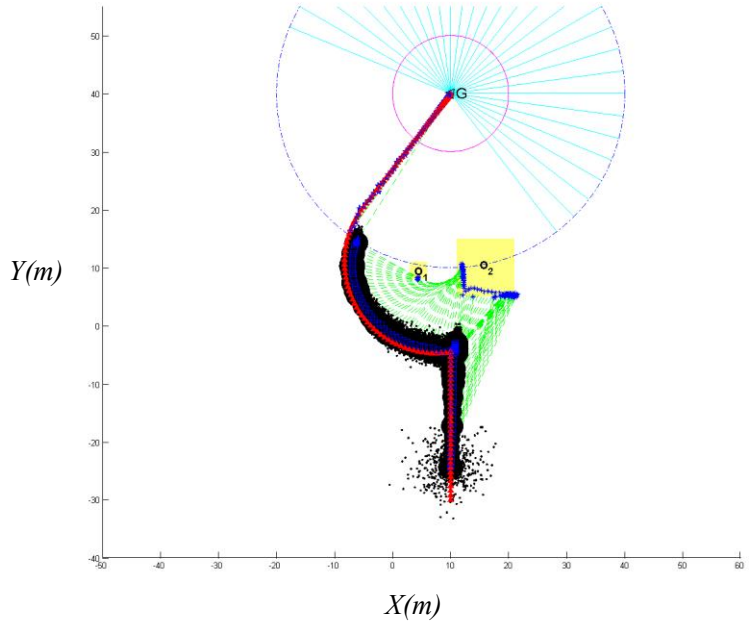
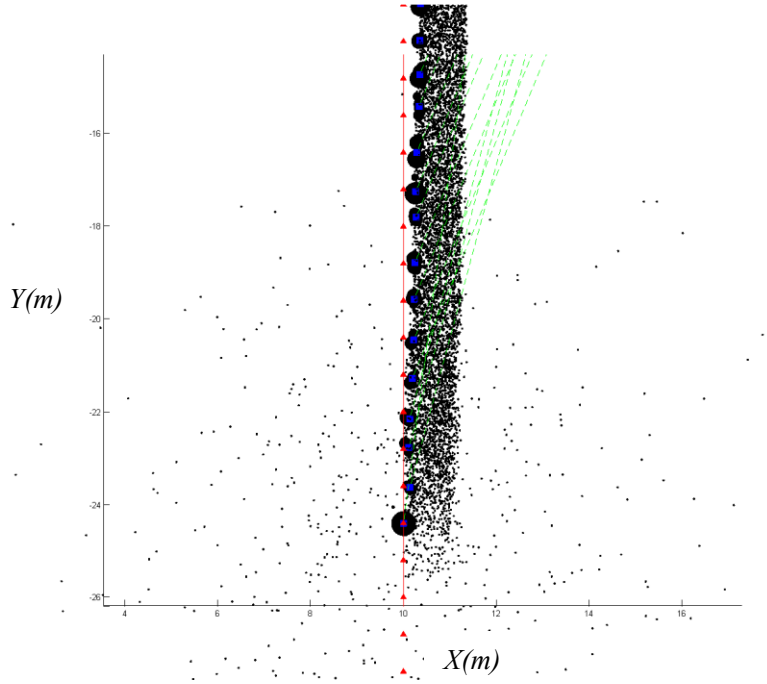
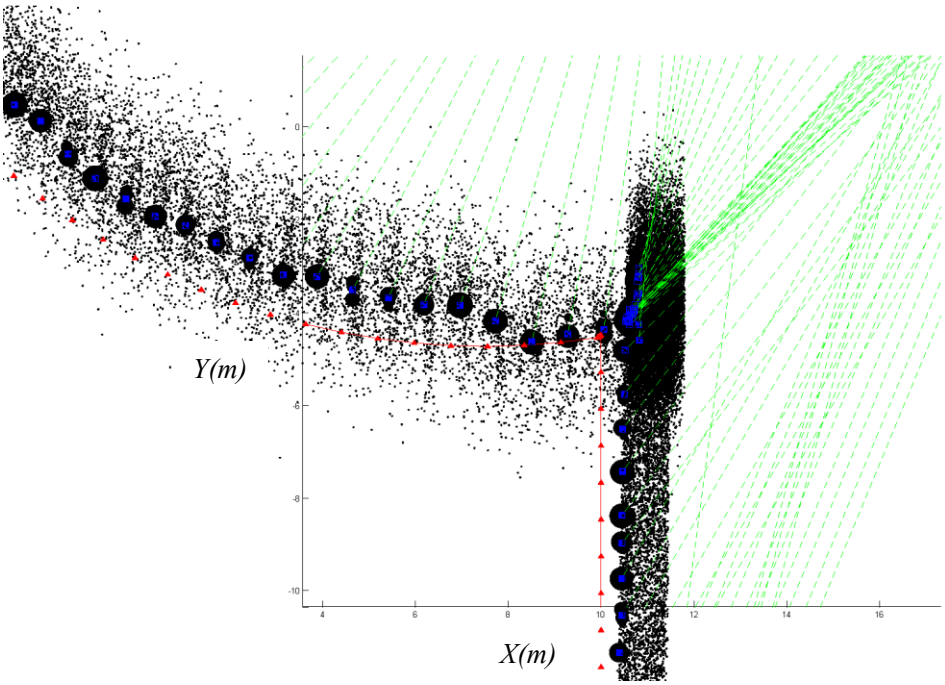


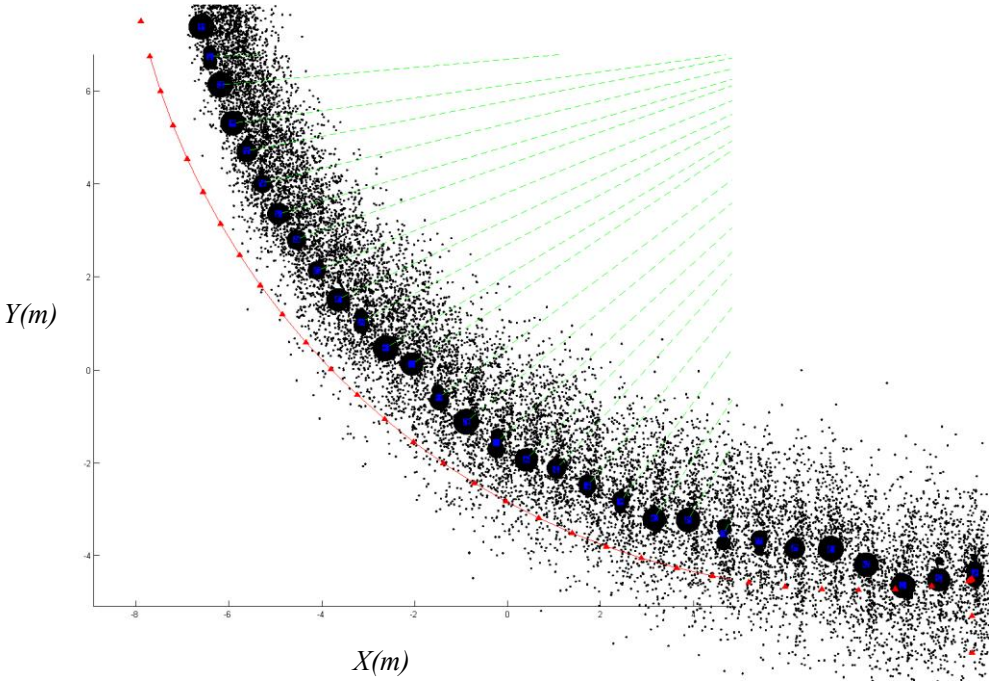
Figure 6.5 Particle clouds representations for estimations with regard to two obstacles.



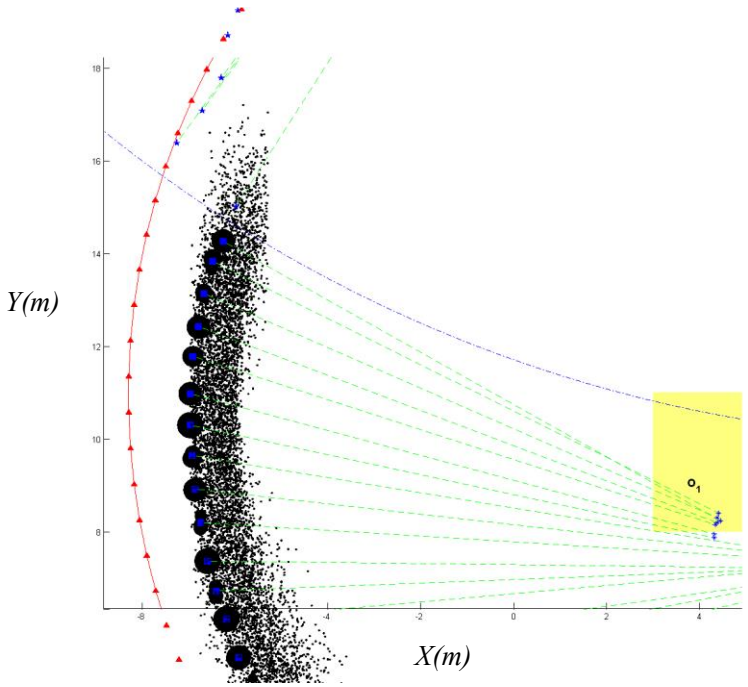
(a)



(b)



(c)



(d)

Figure 6.6 Zooming-in figures for estimations with regard to two obstacles.

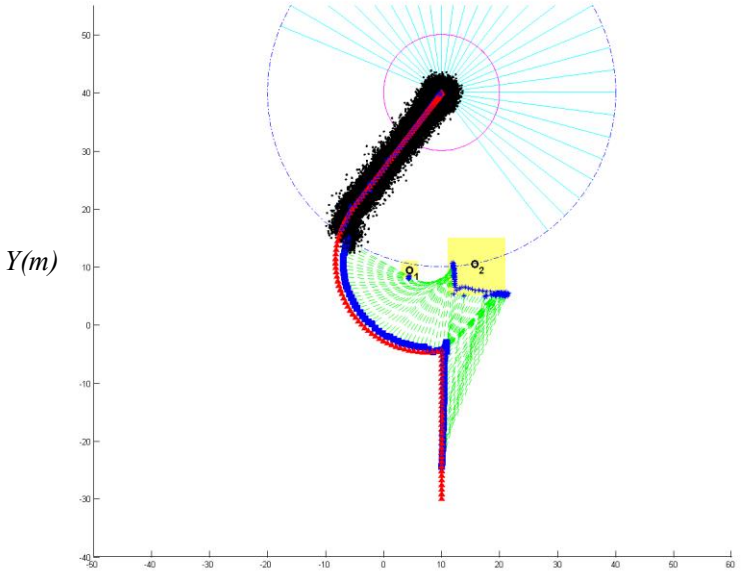
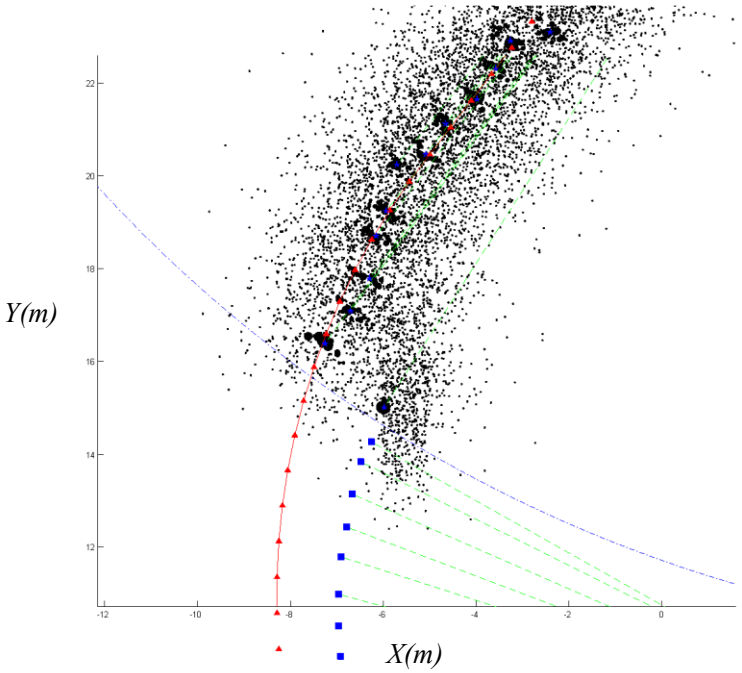
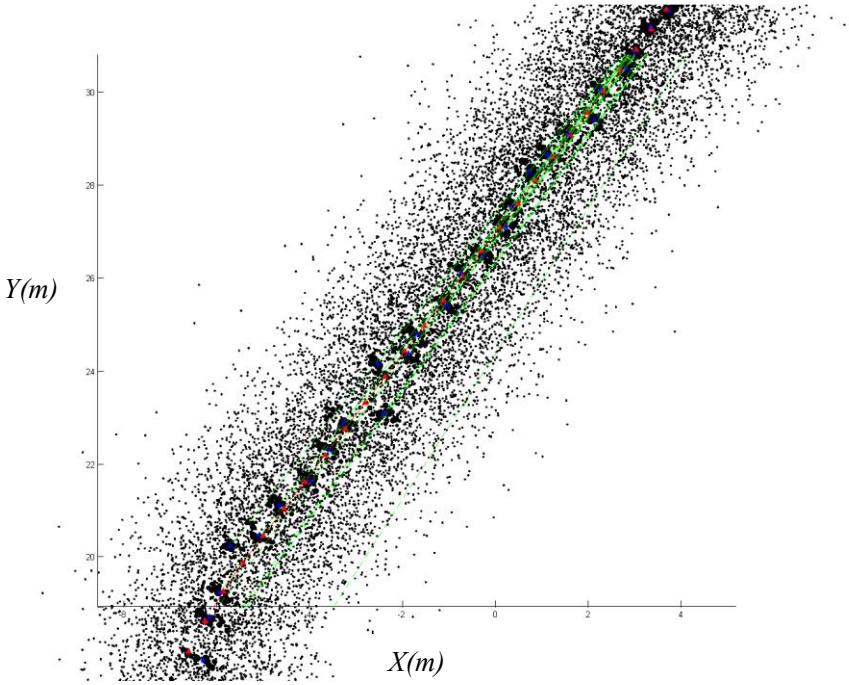


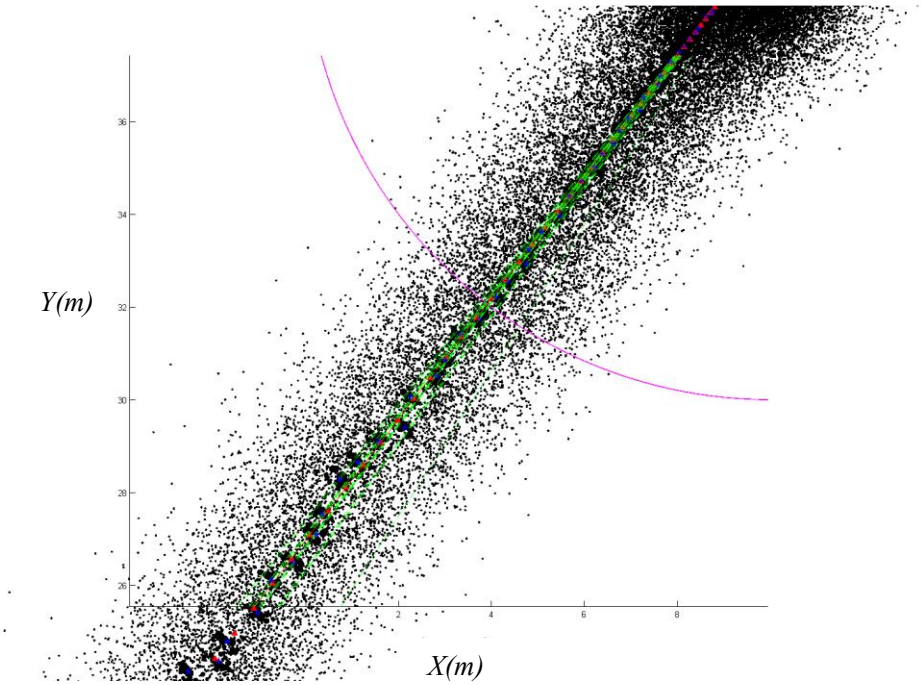
Figure 6.7 Particle clouds representations for estimations with regard to the goal 1.



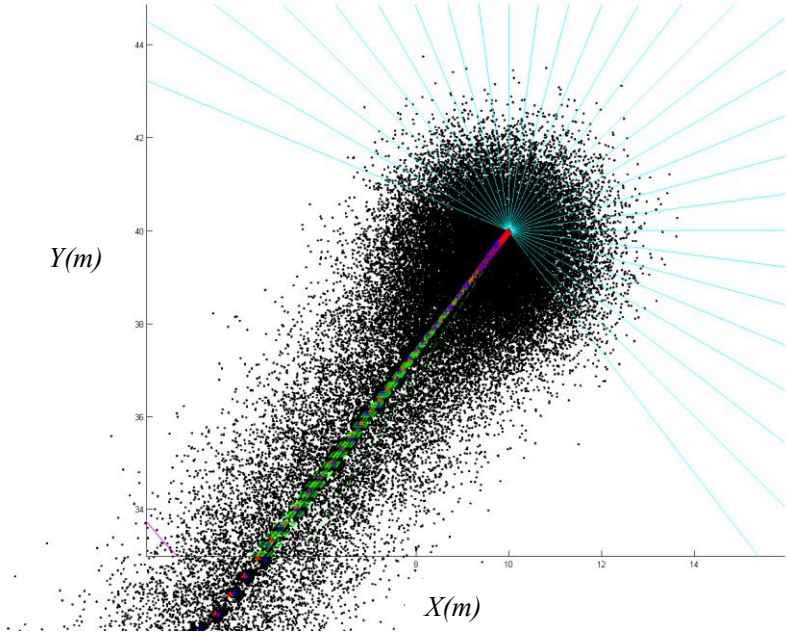
(a)



(b)

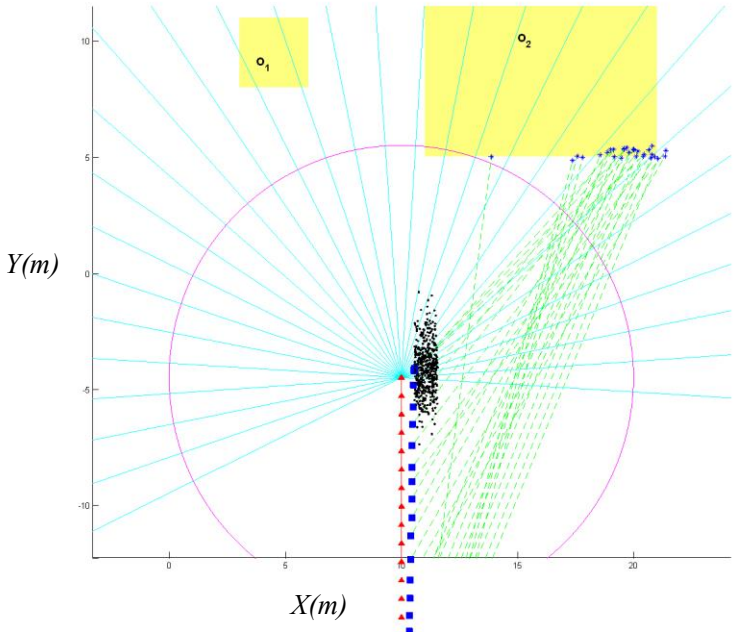


(c)

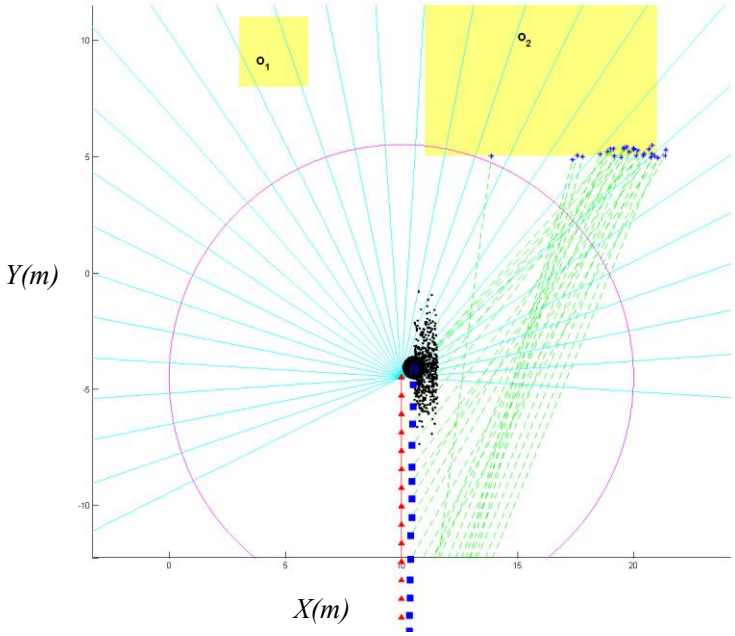


(d)

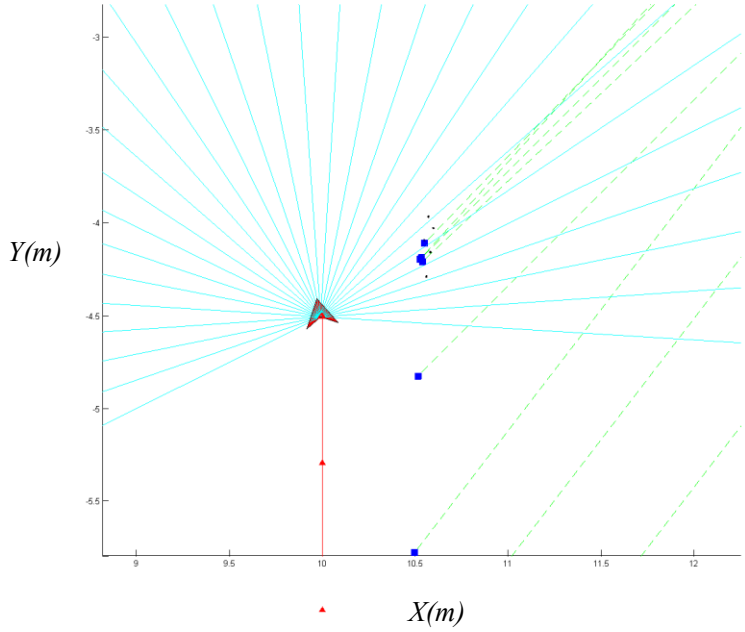
Figure 6.8 Zooming-in figures of Figure 6.7 for estimations with regard to the goal.



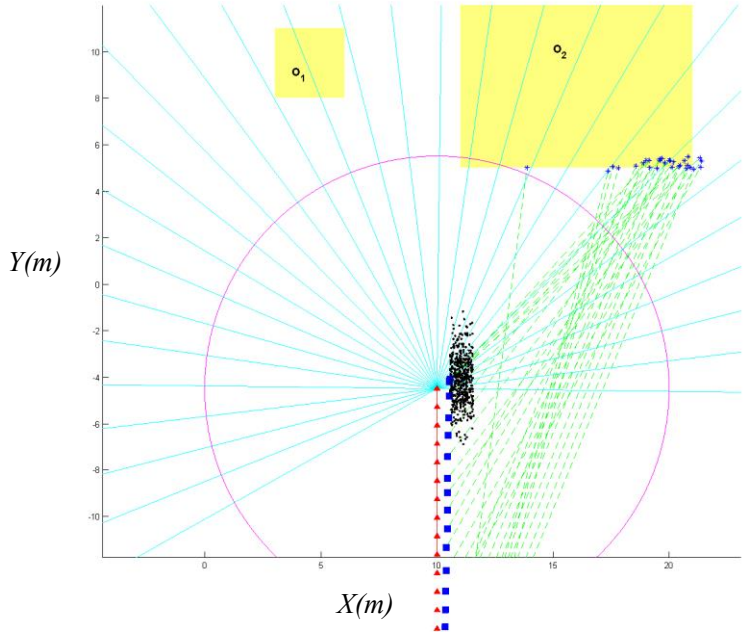
(a)



(b)



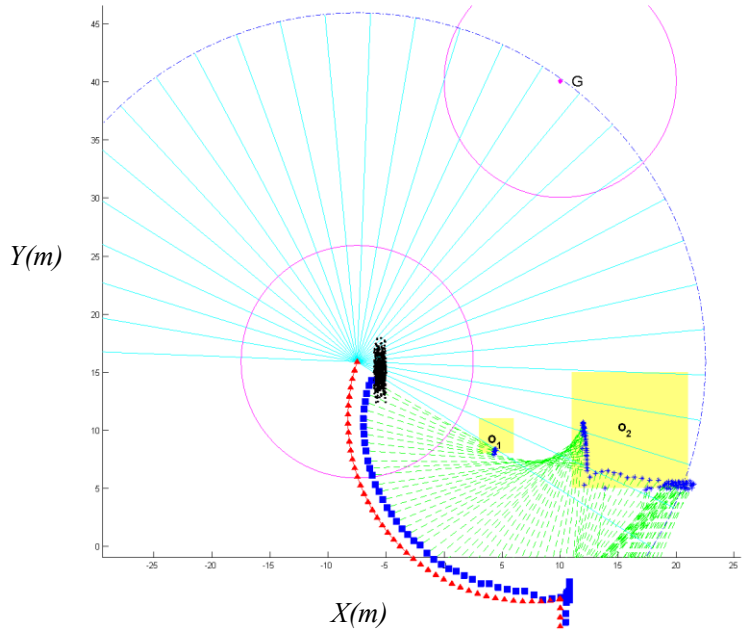
(c)



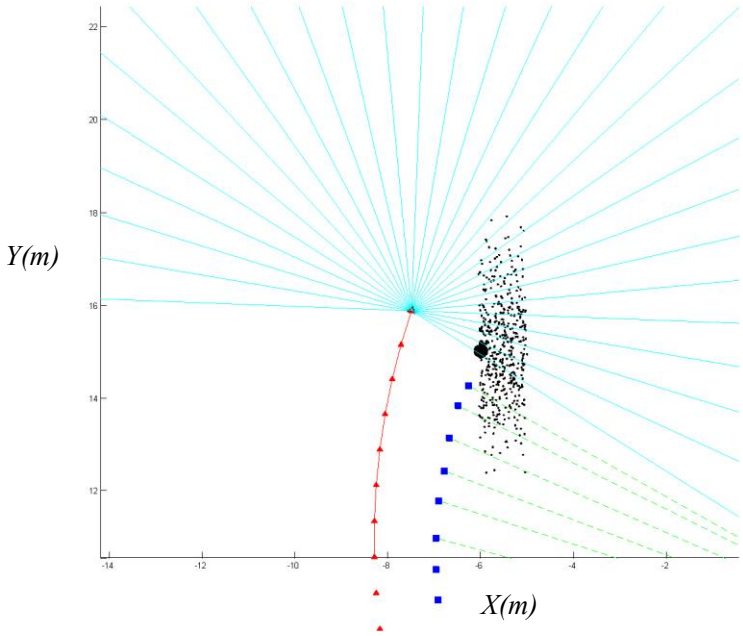
(d)

Figure 6.9 (a) *Prediction* of one step near obstacle O_2; (b) *Update* using measurement wrt obstacles; (c) *Update* after resampling; (d) *Prediction* of next step.

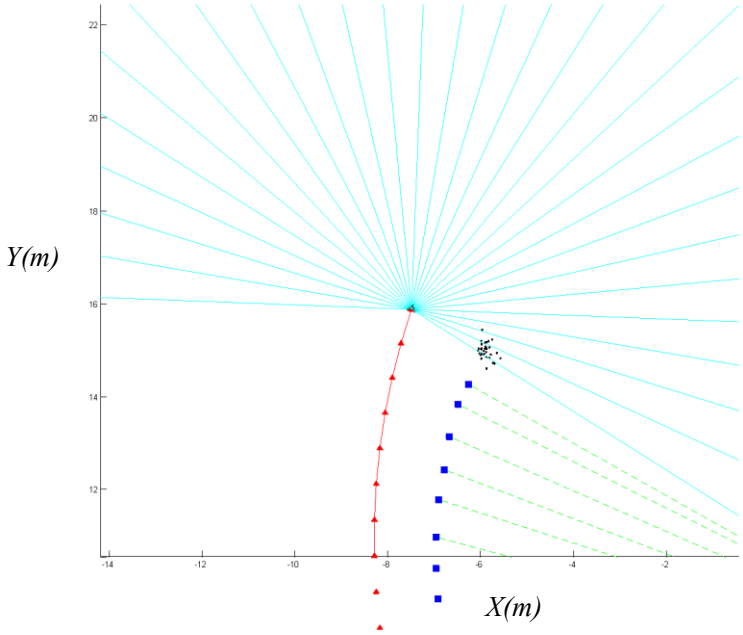
Figure 6.9 presents an illustration of the FastSLAM process with particle filtering when the robot got close to obstacles. In Figure 6.9, (a) and (d) present the *prediction* phase and (b) and (c) describe the *update* phase. In Figure 6.9 (a) particles form a cloud indicating proper position of the robot from the *prediction* phase of the last step. Figure 6.9 (b) presents the *update* phase based on the measurement with regard to obstacles. The area of each particle stands for its weight. The larger its area is, the higher its weight would be. We can see that there are some particles with much higher weights than others. The resampling process is illustrated in Figure 6.9 (c), particles with higher weights in last step survive, and particles with smaller weights are eaten, which we can see in Figure 6.9 (c) just a small part of particles survive to be ready to spread out. Finally, Figure 6.9 (d) presents the second *prediction* phase, which is derived from particles survived in last step propagating to the next generation.



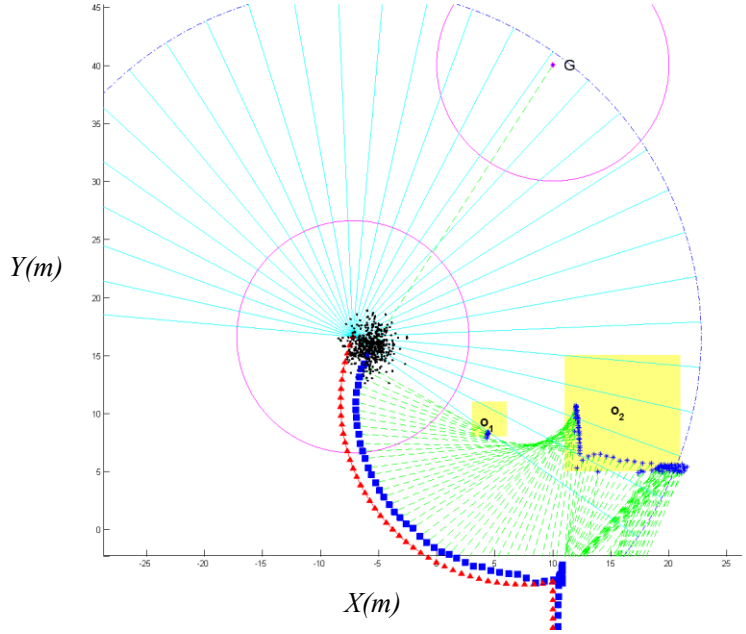
(a)



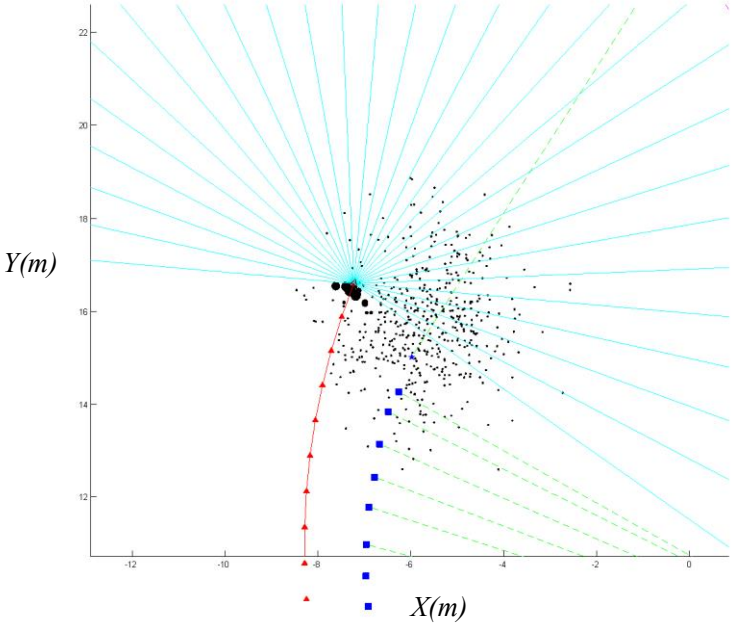
(b)



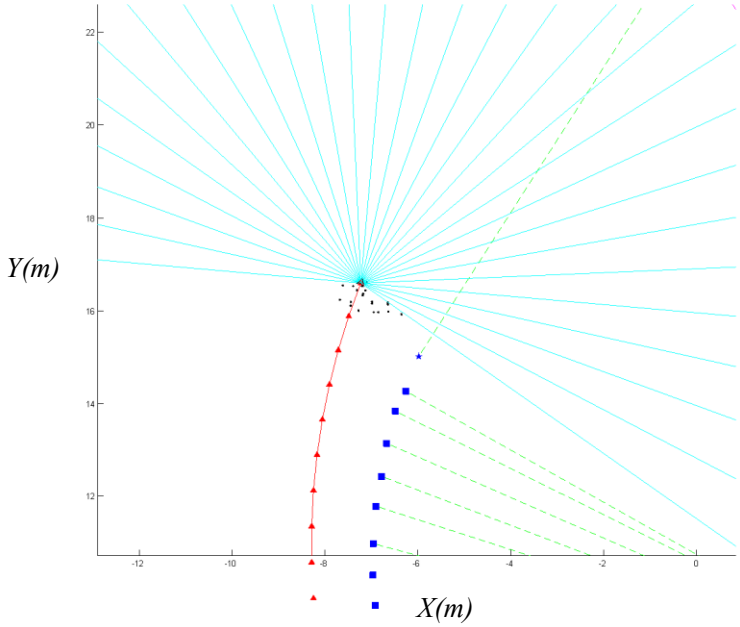
(c)



(d)



(e)



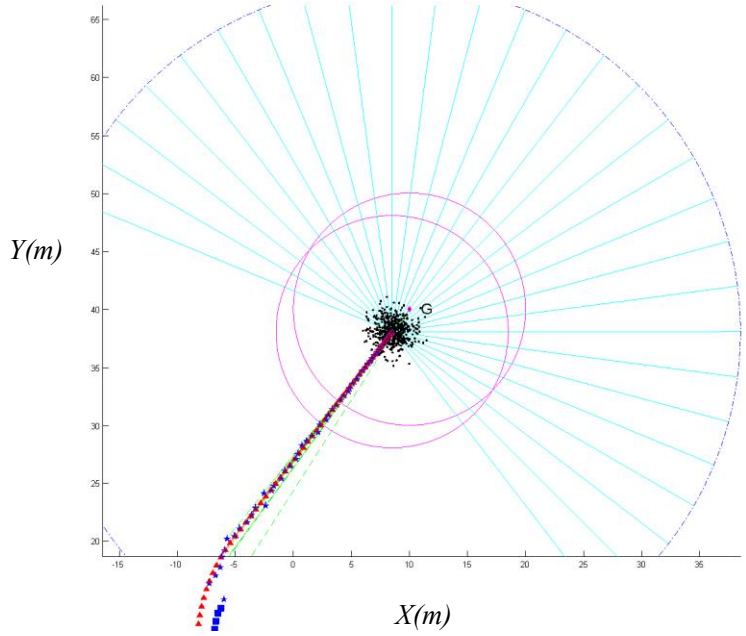
(f)

Figure. 6.10 (a) *Prediction* of one step where robot just sensing the goal. (b) *Update* using measurement wrt the goal. (c) *Update* after resampling. (d) *Prediction* of next step. (e) *Update* using measurement wrt the goal. (f) *Update* after resampling.

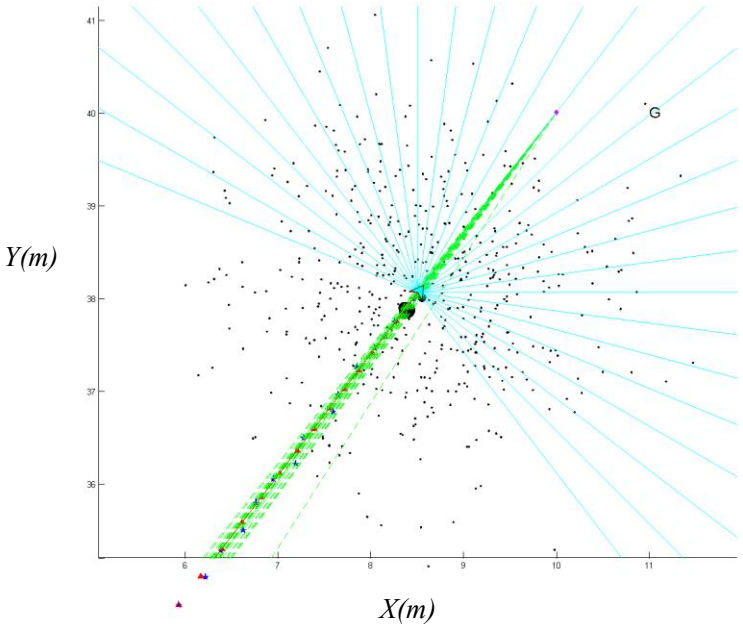
Similarly, Figure 6.10 presents an illustration of the FastSLAM process with particle filtering when robot just sensed the goal. In Figure 6.10, (a) and (d) present the *prediction* phase and (b), (c), (e) and (f) describe the *update* phase. In Figure 6.10 (a) particles form a cloud indicating proper position of the robot from the *prediction* phase of the last step. Figure 6.10 (b) presents the *update* phase based on the measurement with regard to the goal. Also the area of each particle stands for its weight. The larger its area is, the higher its weight would be. We can see that there are some particles with much higher weights than others. The resampling process is illustrated in Figure 6.10 (c), particles with higher weights in last step survive, and particles with smaller weights are gone away, which we can see in Figure 6.10 (c) just a small part of particles survive to be ready to spread out. Figure 6.10 (d) presents the second *prediction* phase, which is derived from particles survived in last step propagating to the next generation.

Different from Figure 6.9, we have shown two more *update* phases (Figure 6.10 (e) and (f)) in order to show that estimation of robot path wrt the goal for which we know the absolute

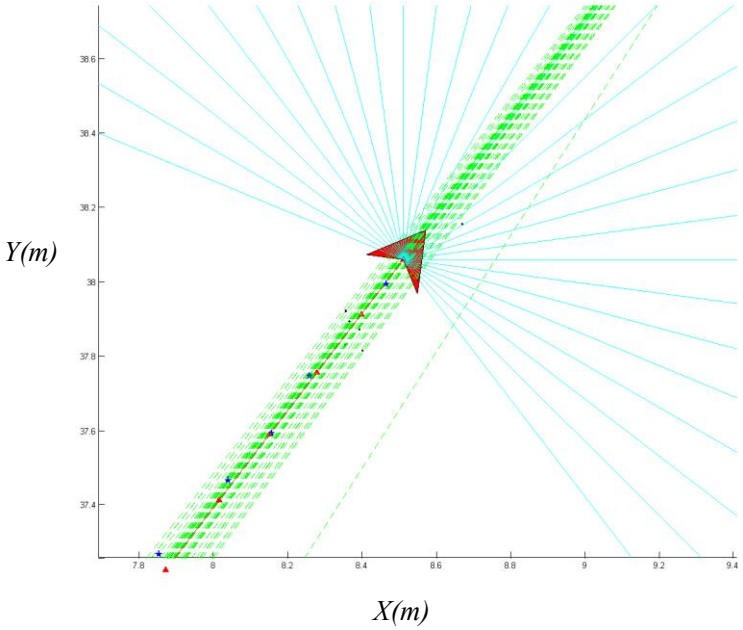
position of have better performance than estimation wrt obstacles for which we do not know the absolute position of. In Figure 6.10 (e), we can see that particles we want to propagate to the next generation (shown in particles with bigger area) become more realistic, with smaller residual error with the ideal robot path. Then after resampling in Figure 6.10 (f), particles with higher weights live on.



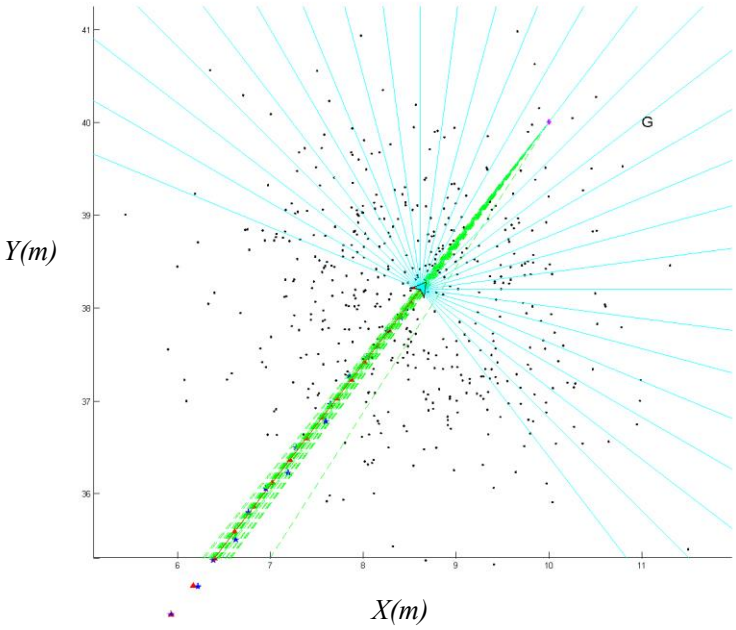
(a)



(b)



(c)



(d)

Figure 6.11 (a) *Prediction* of one step close to the goal. (b) *Update* using measurement wrt the goal. (c) *Update* after resampling. (d) *Prediction* of next step.

Finally, when the robot moved near the goal, we illustrated some particle filtering snapshots as well. Since we have already explained the FastSLAM process with particle filtering in Figure 6.9 and 6.10, we do not present more about it here. Likewise, Figure 6.11 (a) and (d) present the *prediction* phase and (b) and (c) describe the *update* phase based on the measurement wrt the goal. We can see that the closer robot is from the goal, the smaller residual error of robot ideal path and estimation of robot path is.

6.2.2 Robot Travelling Around a Concave Obstacle

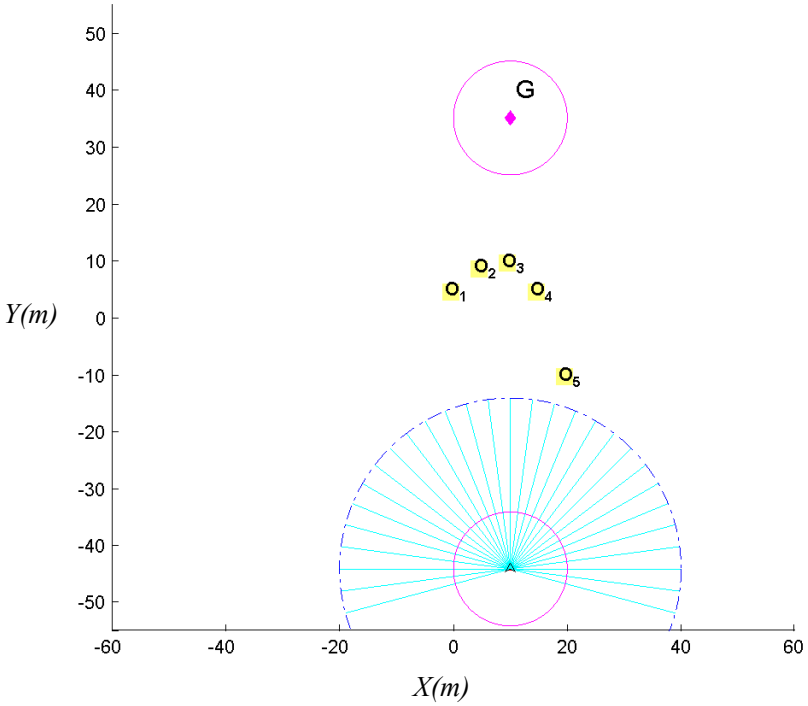
In Section 6.2.1 we have presented simulation results about robot travelling around two obstacles. In this section, we will talk about simulations with regard to robot travelling around a concave obstacle emulated of five small obstacles too close to one-another to let the robot go in-between. The configuration of our simulation environment has already introduced in Figure 6.3 (b) where the concave obstacle has a dissymmetric shape. The robot will find a proper

turning direction to escape the concave obstacle efficiently and finally reach the goal.

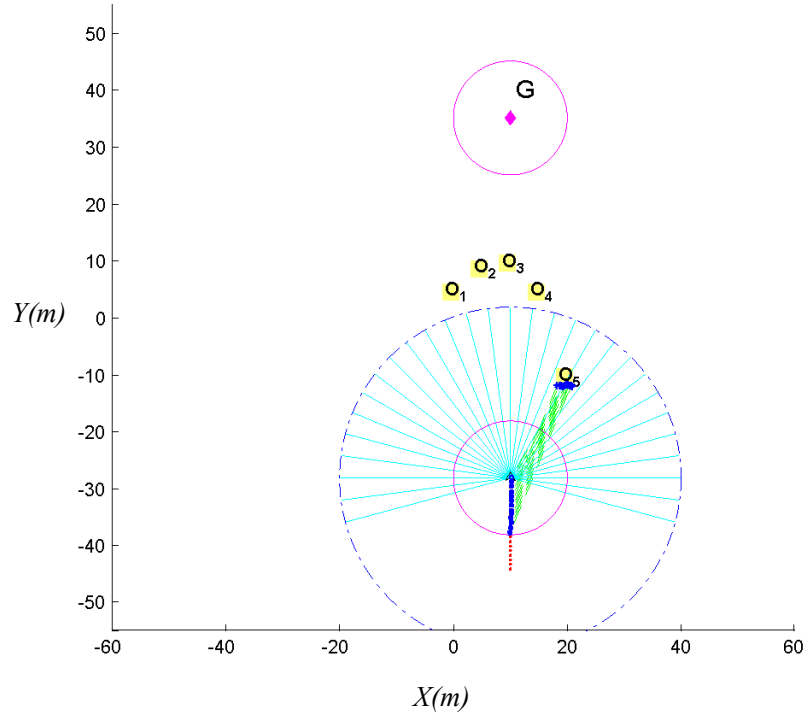
Figure 6.12 (a-j) shows robot navigation escaping a concave avoidance with the FastSLAM approach (obtaining the estimations of robot path and positions of obstacles). In this simulation, for each time step robot built a local map to choose proper turning direction based on comparison about dimension of obstacles in left section of robot sensing range and in right section of robot sensing range. And it turns out that robot turned left to escape the concave obstacle efficiently. The legend of the figures are exactly the same as what we used in Table 6.2.

The obstacle avoidance algorithm is based on the velocity potential fields approach as well, and the estimations with regard to robot path and obstacles are also performed by using the FastSLAM approach. In Figure 6.12 (g), the robot did not detect any obstacle or the goal, thus the FastSLAM approach did not work right in that period because of no measurements. The controller was just leading robot to the goal at that time with system errors accumulated. We can see that there are no estimations of robot positions and corresponding estimation of obstacles in that period in Figure 6.12 (g). In Figure 6.12 (h), after robot turning its head nearly towards the goal, it detected obstacles again and had measurements wrt obstacles resulting in estimations about robot positions and obstacles could be obtained. When robot found the goal, it has taken measurement with regard to the goal instead of obstacles shown in Figure 6.12 (i).

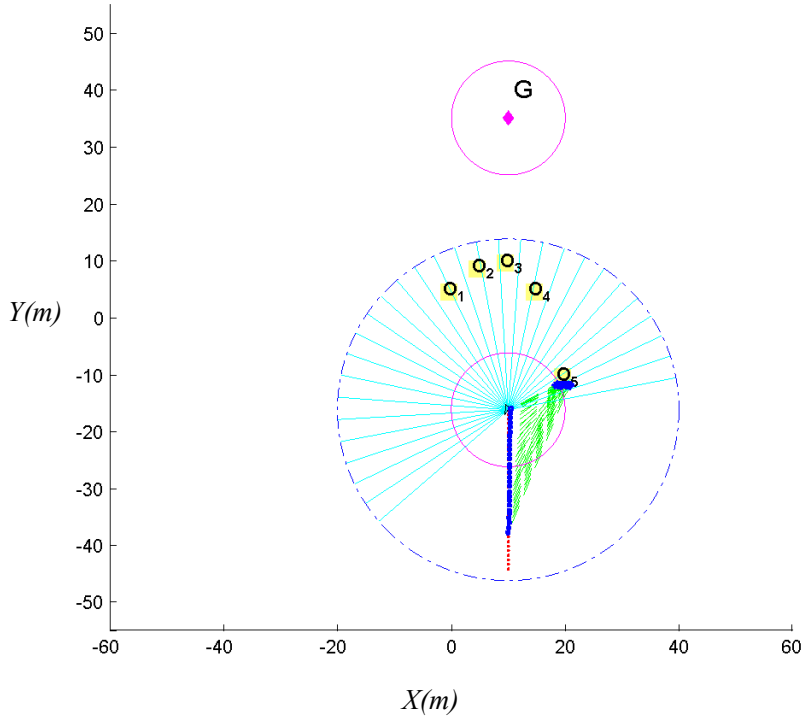
In Figure 6.12 (a-j), 10 snapshots are shown to robot navigation using the FastSLAM approach successfully escaping a concave obstacle. When robot detected obstacles, estimations of the robot path is regarding to measurements about obstacles. However, when robot bypassed obstacles and sensed the goal, the estimation of the robot path is based on measurements wrt the goal. The estimated robot path and the real robot path converged finally when robot reached the goal, and we find the estimation wrt obstacles has more residual error with the real robot positions than the estimation wrt the goal.


G
O1
O2
O3
O4
O5
Y(m)
X(m)
50
40
30
20
10
0
-10
-20
-30
-40
-50
-60
-40
-20
0
20
40
60

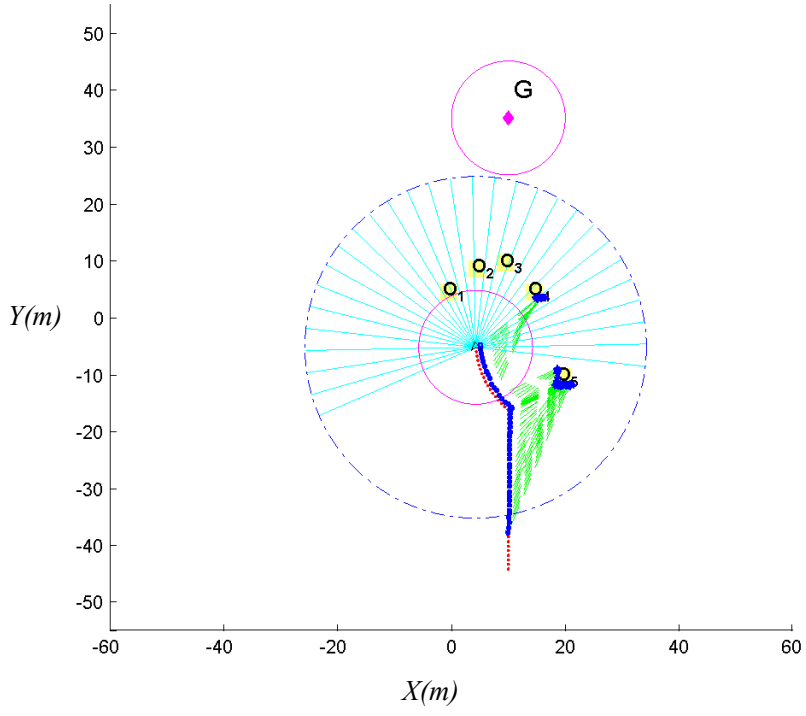
(a)


G
O1
O2
O3
O4
O5
Y(m)
X(m)
50
40
30
20
10
0
-10
-20
-30
-40
-50
-60
-40
-20
0
20
40
60

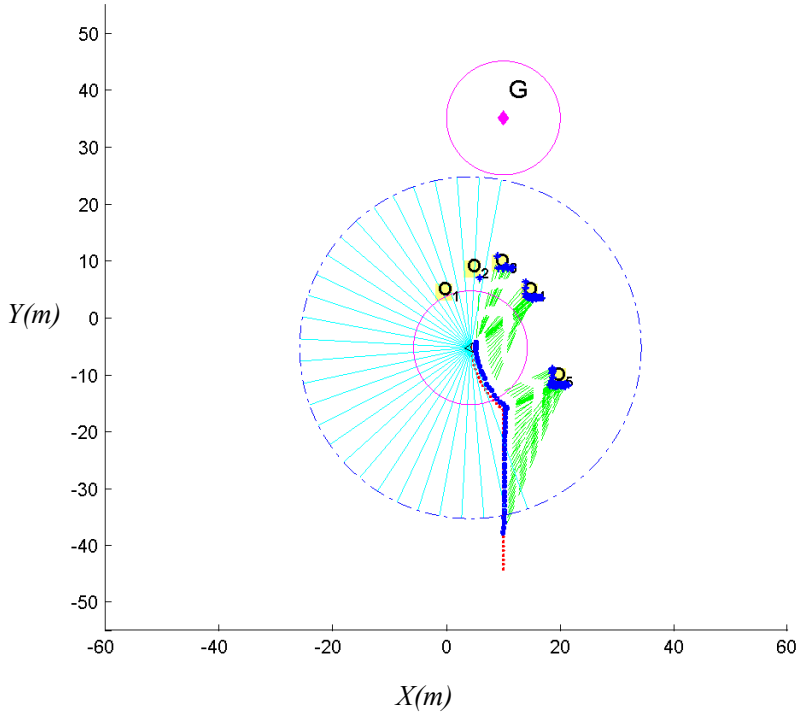
(b)


G
O1
O2
O3
O4
O5
Y(m)
X(m)
50
40
30
20
10
0
-10
-20
-30
-40
-50
-60
-40
-20
0
20
40
60

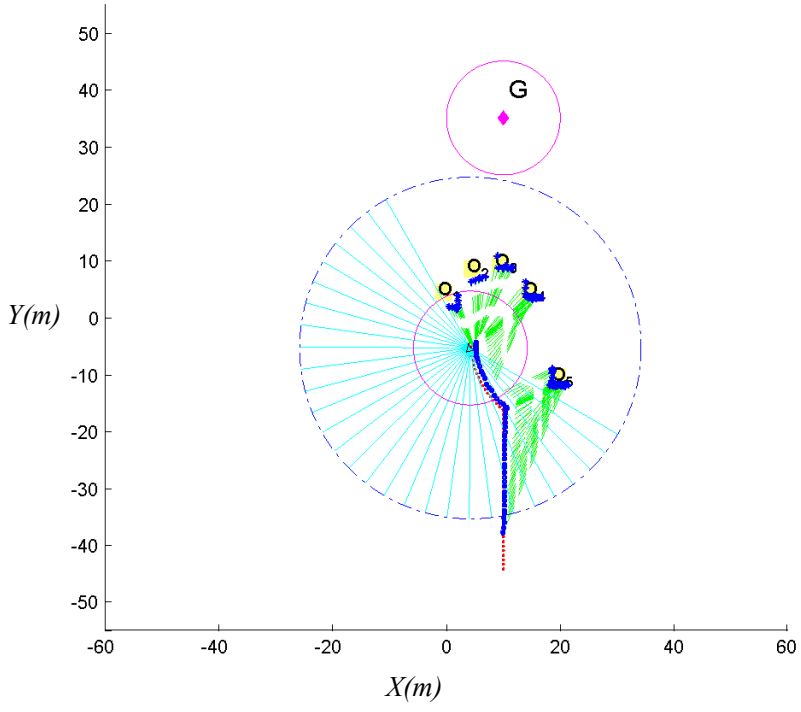
(c)


G
O1
O2
O3
O4
O5
Y(m)
X(m)
50
40
30
20
10
0
-10
-20
-30
-40
-50
-60
-40
-20
0
20
40
60

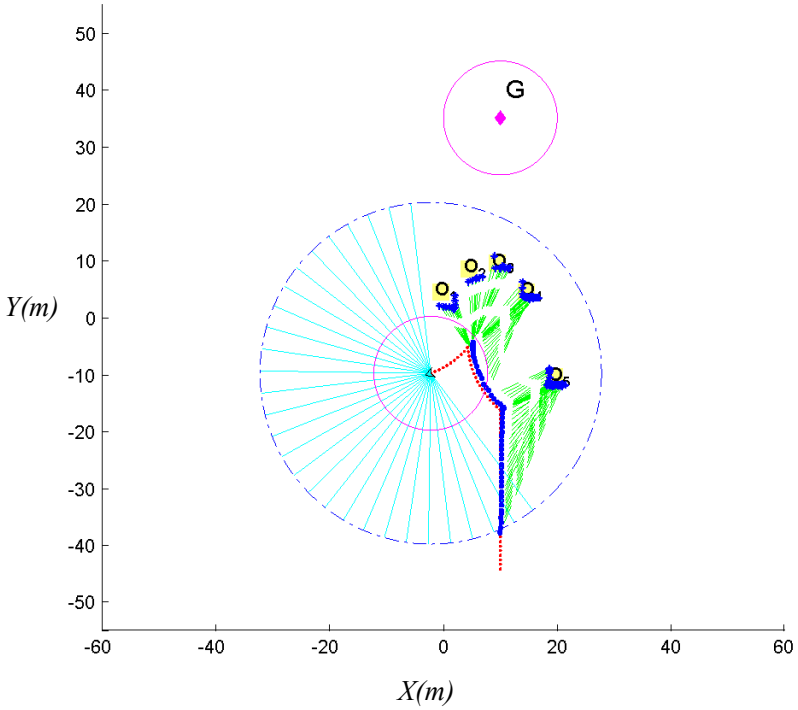
(d)



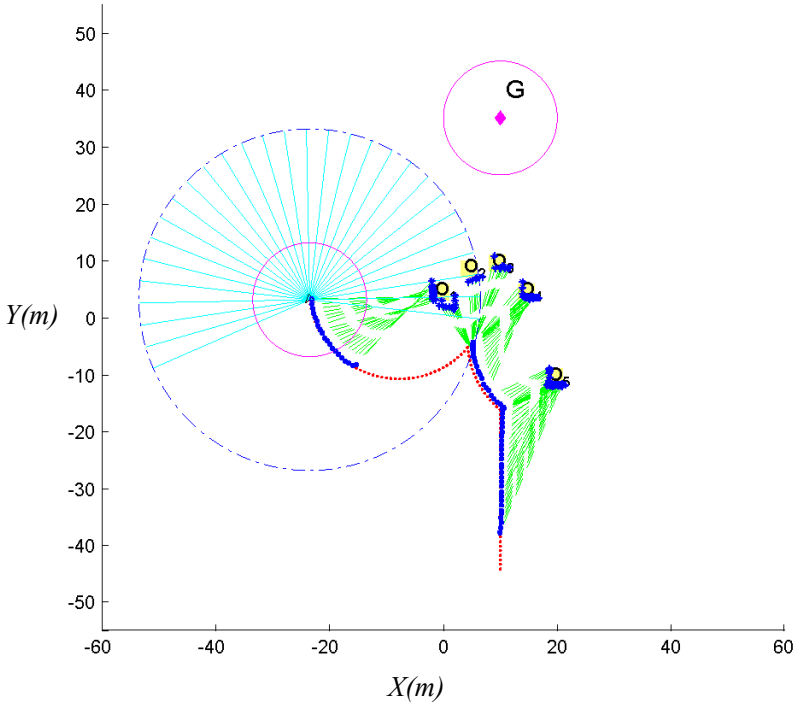
(e)



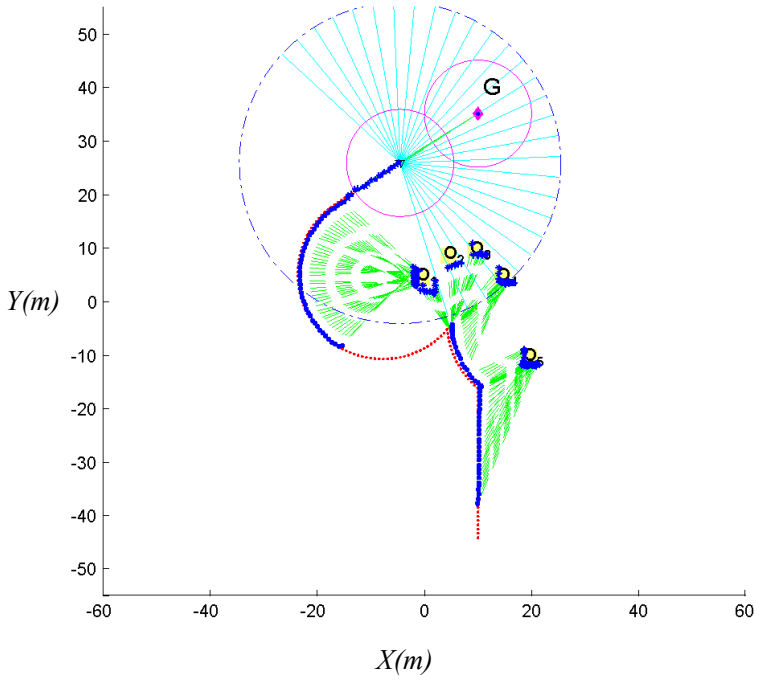
(f)



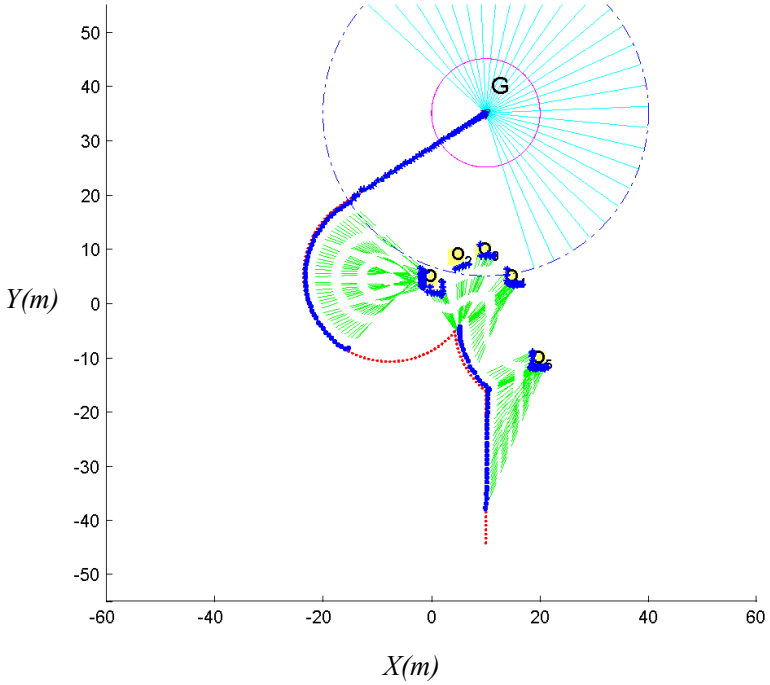
(g)



(h)



(i)



(j)

Figure 6.12 Robot travelling around a concave obstacle with the FastSLAM approach.

In Figure 6.13, we give particle clouds representations of estimations of robot positions with regard to obstacles shown in Figure 6.3 (b). Instead of using a red dotted line as the ideal robot positions, we presented the ideal robot path as a solid line with upward-pointing triangle markers in order to obtain better illustration.

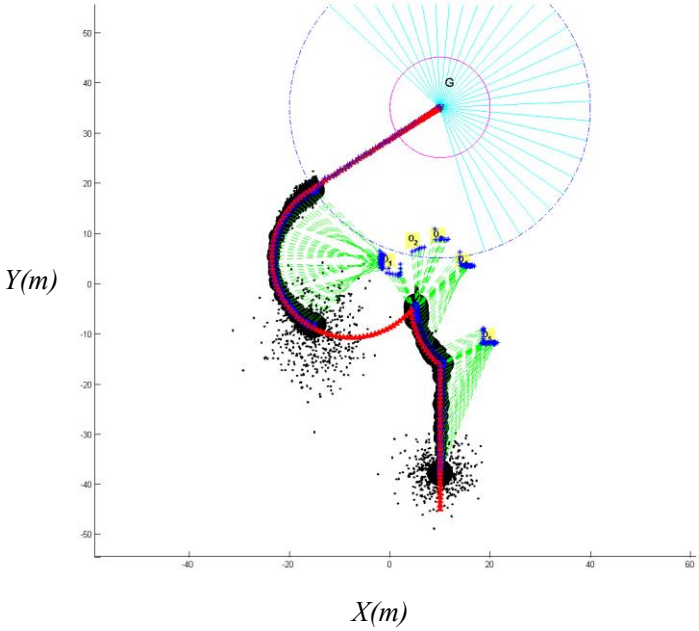
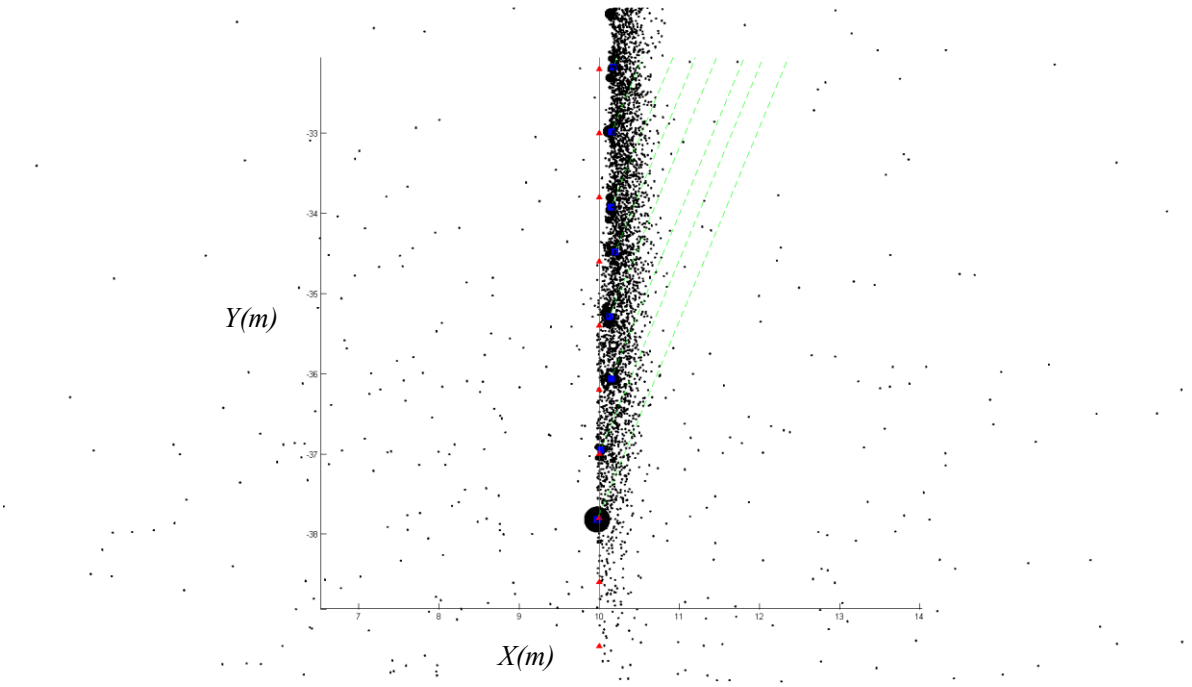
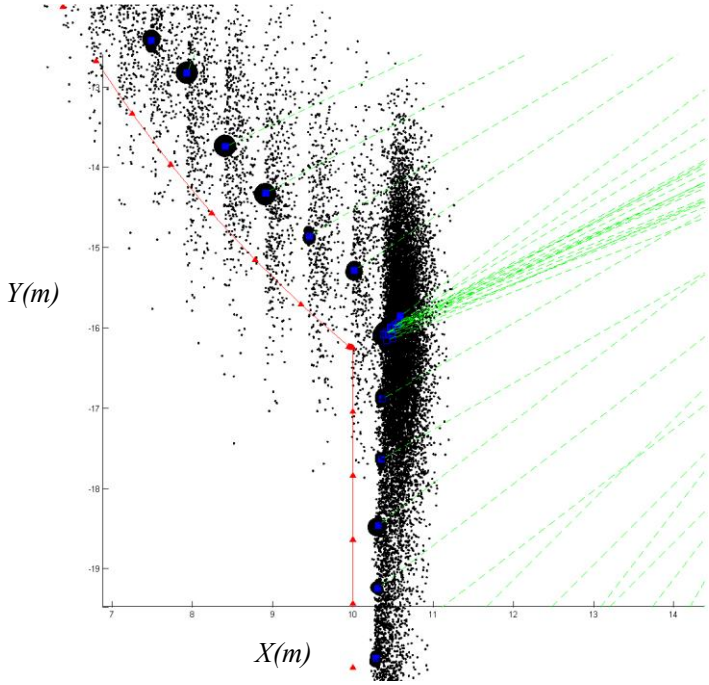


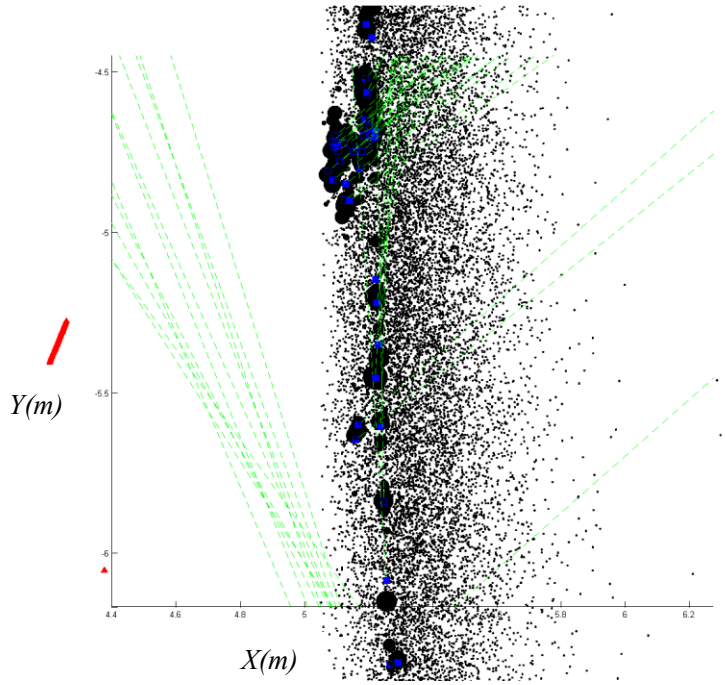
Figure 6.13 Particle clouds representations for estimations with regard to five obstacles.



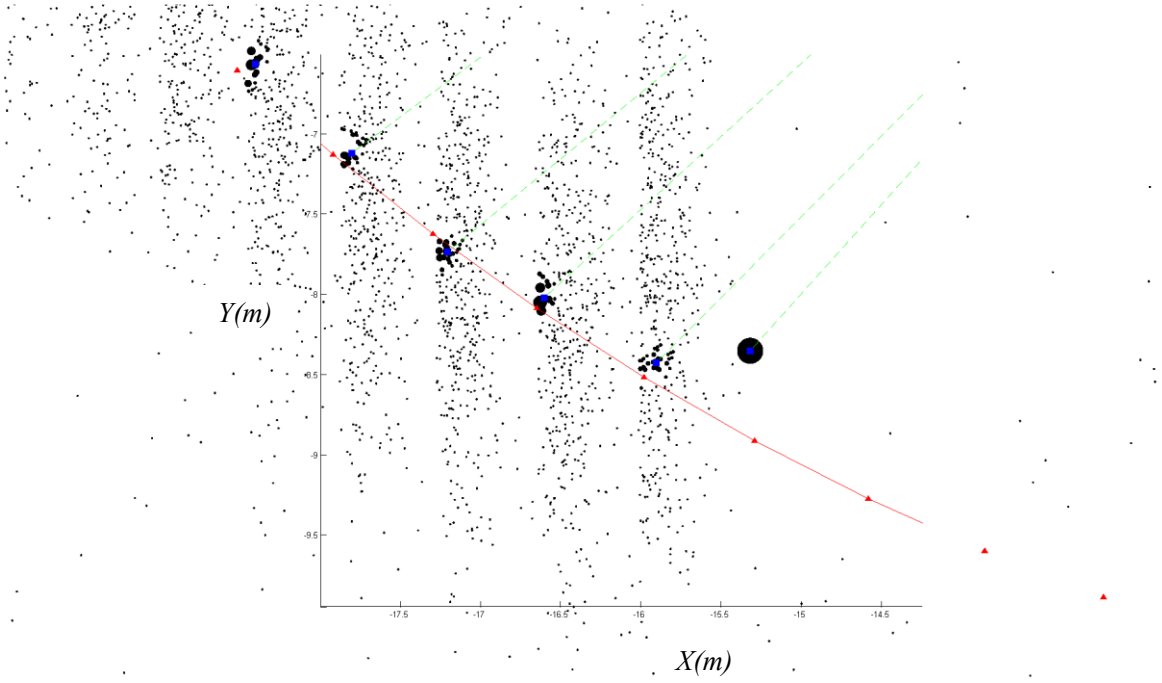
(a)



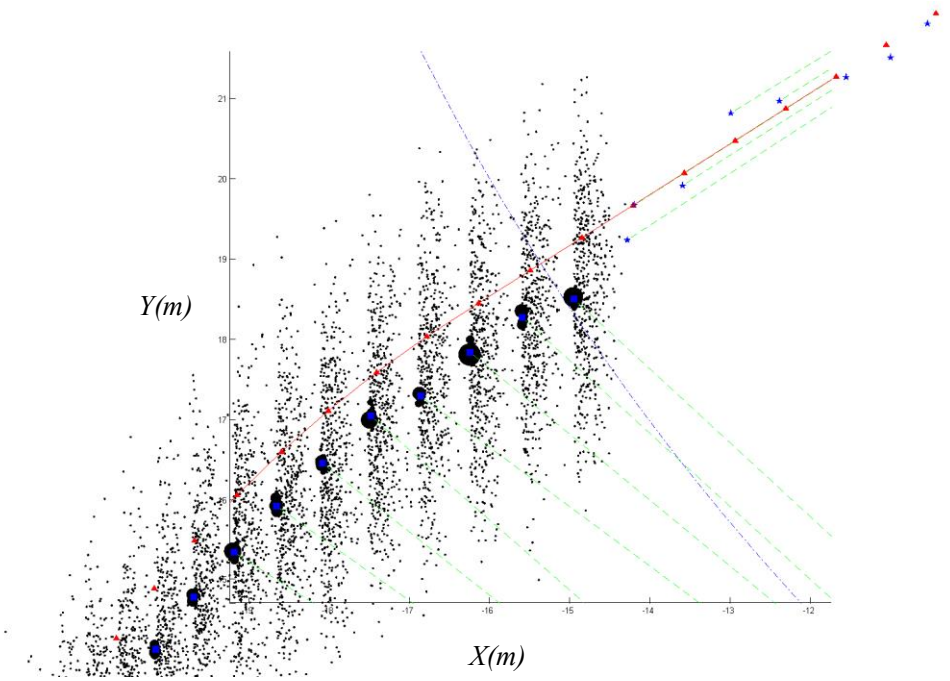
(b)



(c)



(d)



(e)

Figure 6.14 Zooming-in figures for estimations with regard to five obstacles.

Figure 6.14 (a-e) presents the zooming-in figures of Figure 6.13 as we want to present detailed illustration. Figure 6.14 (a) indicates when robot first detected obstacles, the FastSLAM algorithm started working. Afterwards, robot travelled too close to obstacle O_5, and had to turn, this process is shown in Figure 6.14 (b). Figure 6.14 (c) describes robot went too close to obstacle O_1, and turned left to go out of the concave obstacle. When robot went out of the concave obstacle, in a short time it did not detect obstacles and the goal, hence there were not estimations about robot positions. Figure 6.14 (d) indicates that the robot once again detected obstacle O_1 after turning its head to the goal and particle filtering process resumed working. Finally, when robot bypassed obstacles and sensed the goal, estimation about obstacles ended as shown in Figure 6.14 (e).

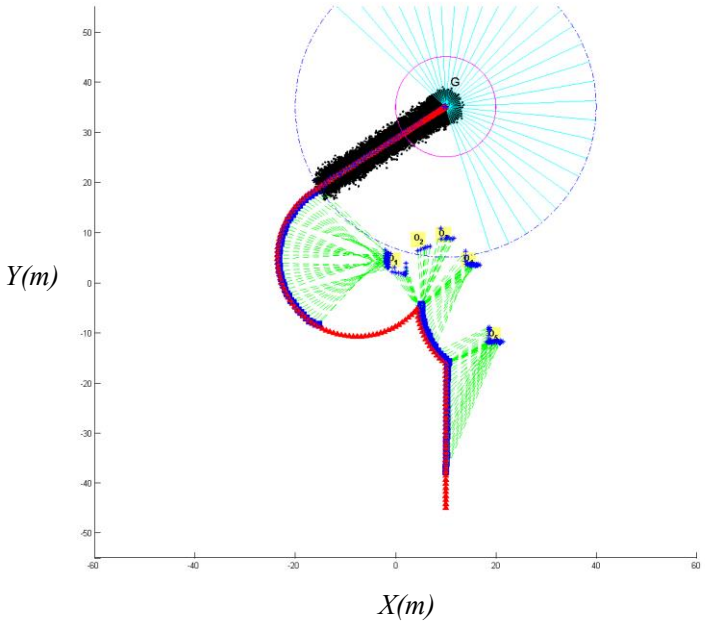
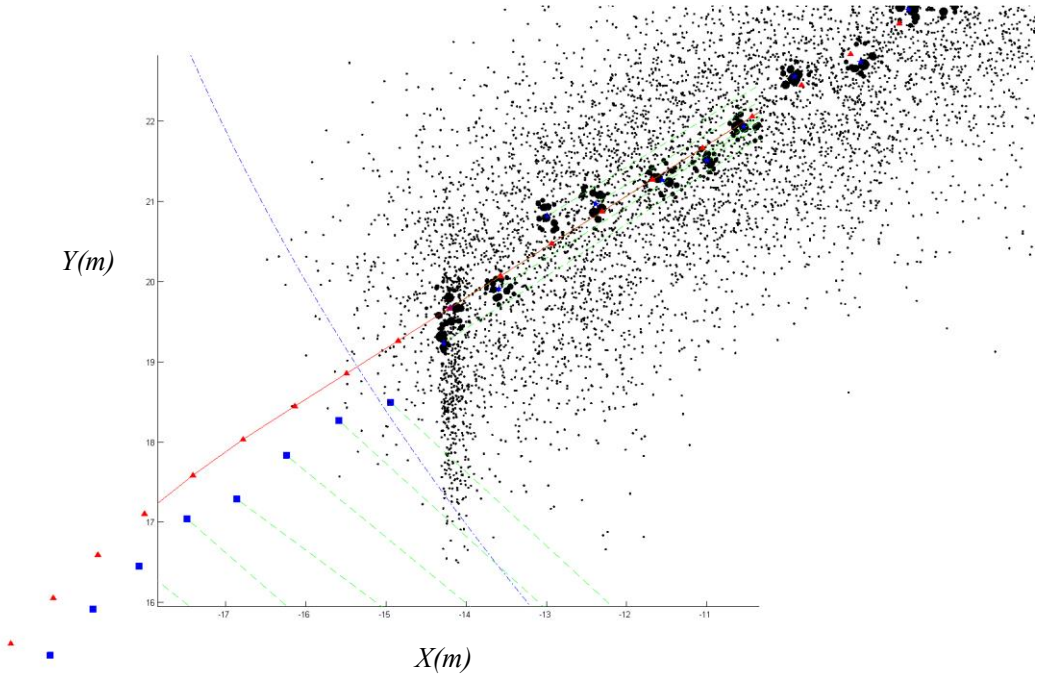
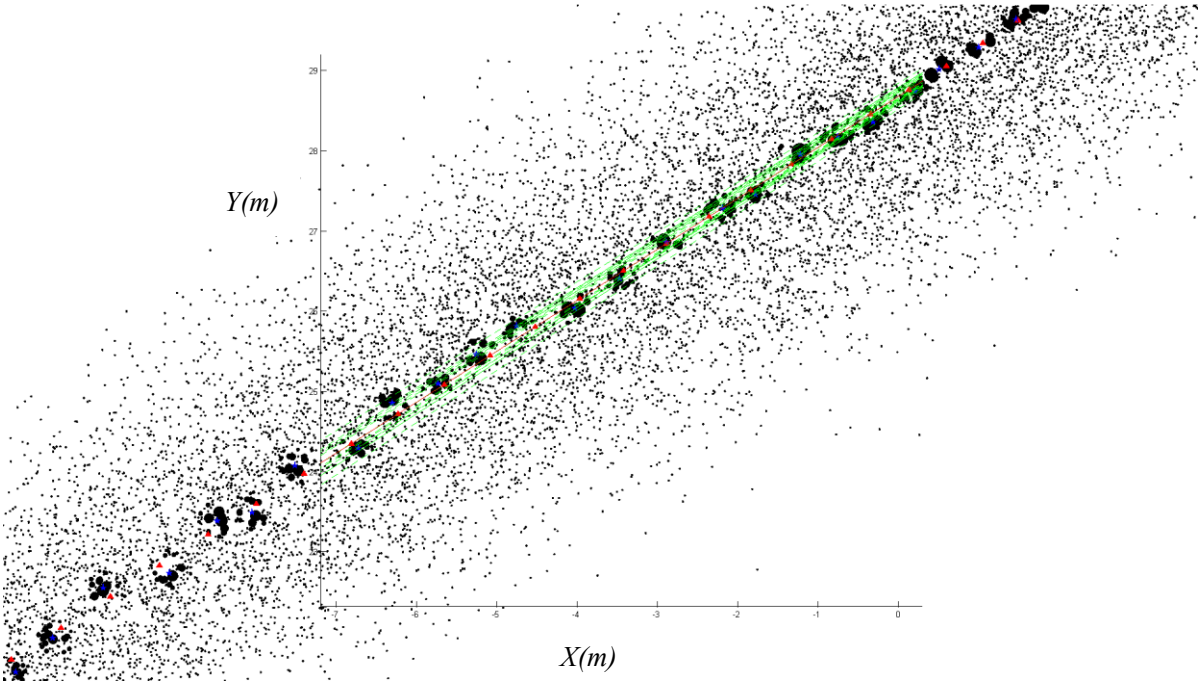


Figure 6.15 Particle clouds representations for estimations with regard to the goal 2.

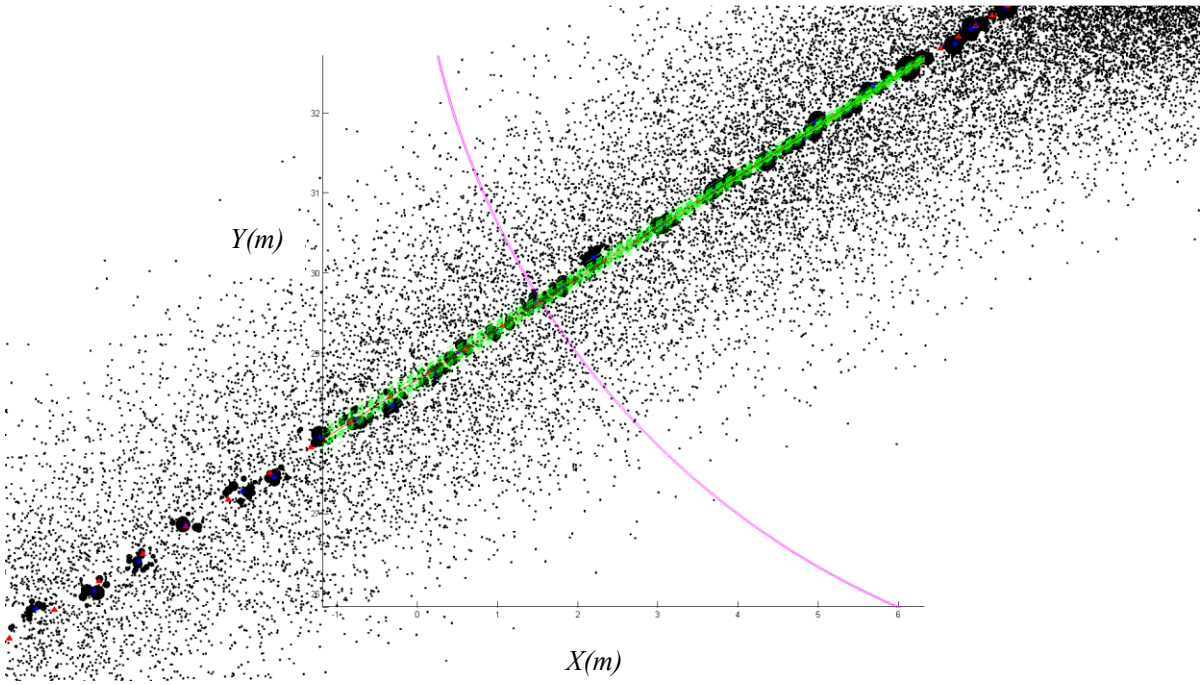
Likewise, Figure 6.15 describes estimations of robot positions with regard to the goal using particle clouds representations. The zooming-in figures of Figure 6.15 are illustrated in Figure 6.16 (a-d). Figure 6.16 (a) denotes when robot just sensed the goal first time, estimations of robot positions about obstacles changed to estimations wrt the goal. Figure 6.16 (b-d) shows the particles distribution along with robot moving towards the goal.



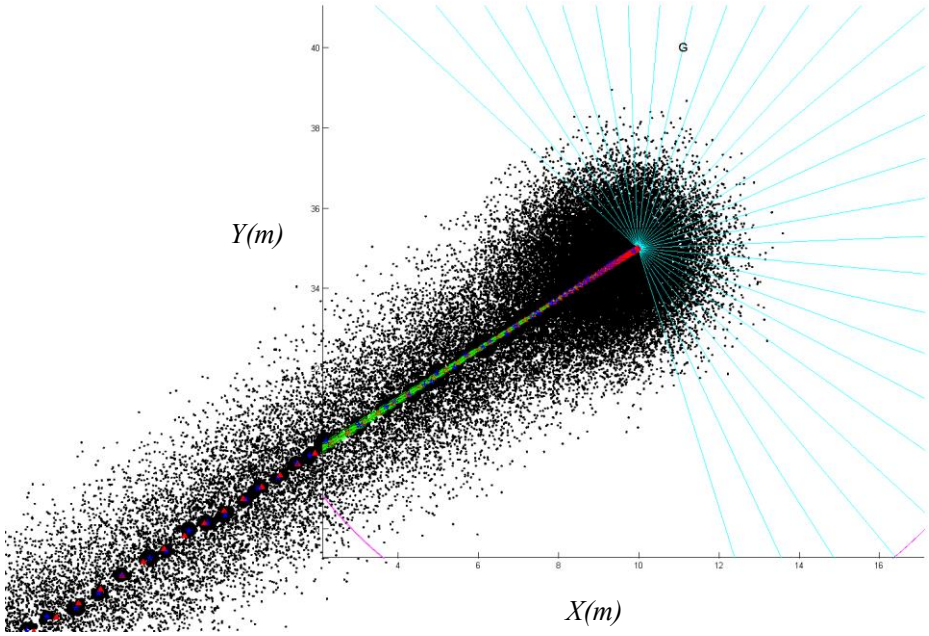
(a)



(b)

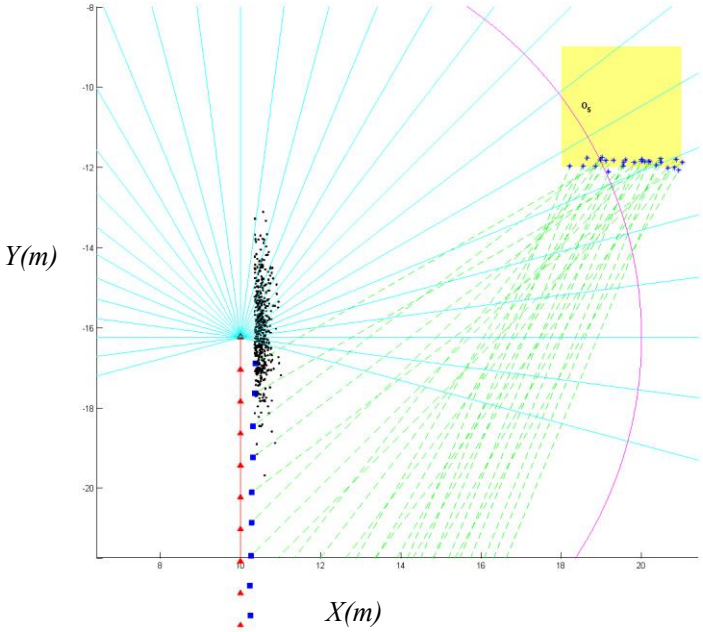


(c)

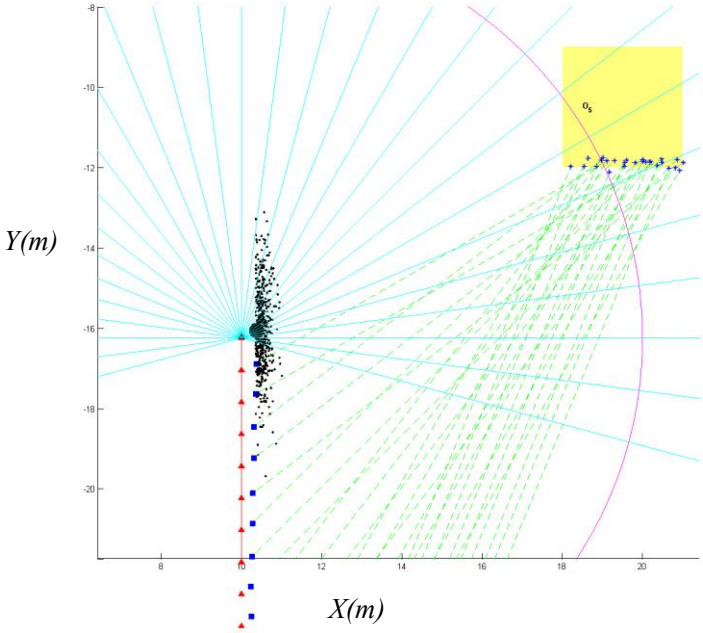


(d)

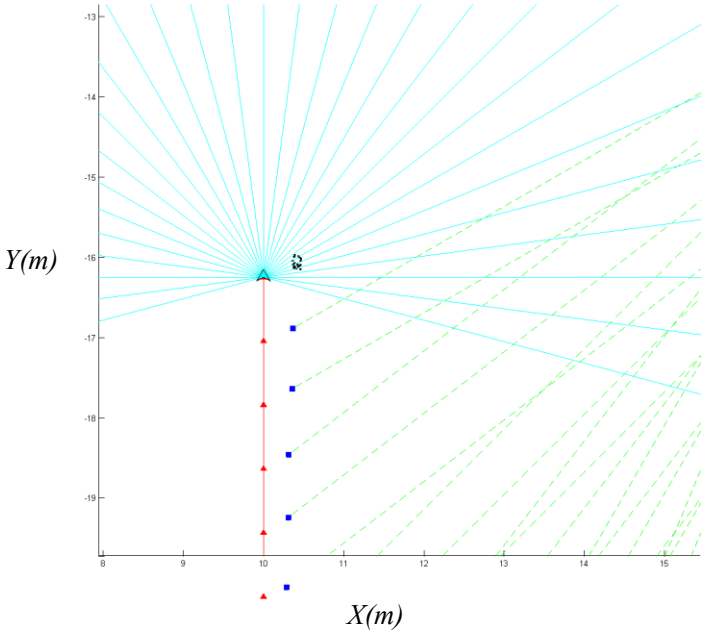
Figure 6.16 Zooming-in figures of Figure 6.15 for estimations with regard to the goal.



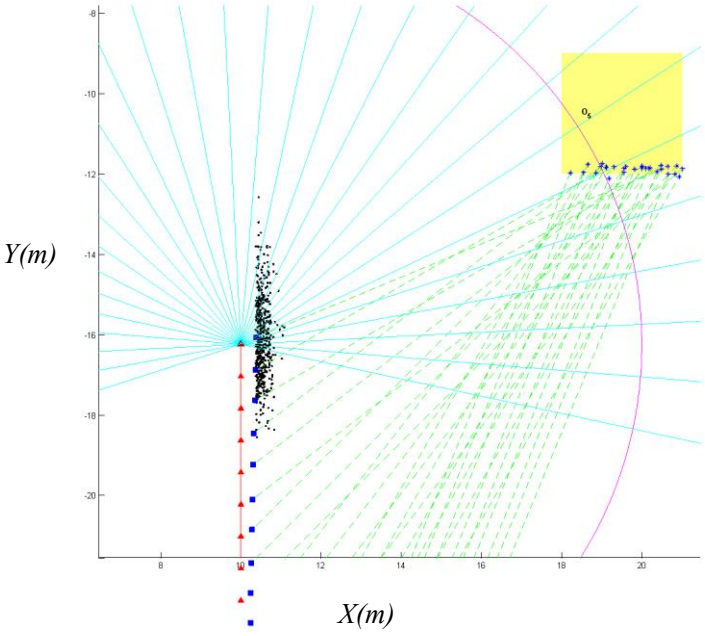
(a)



(b)



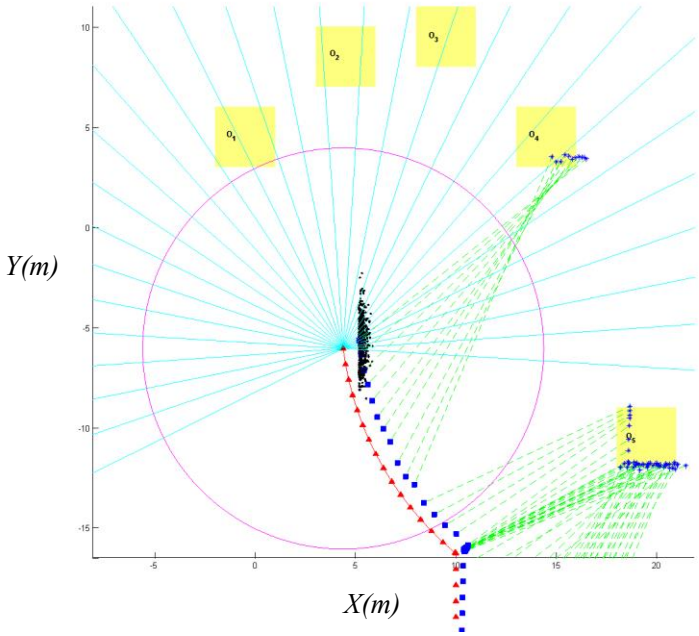
(c)



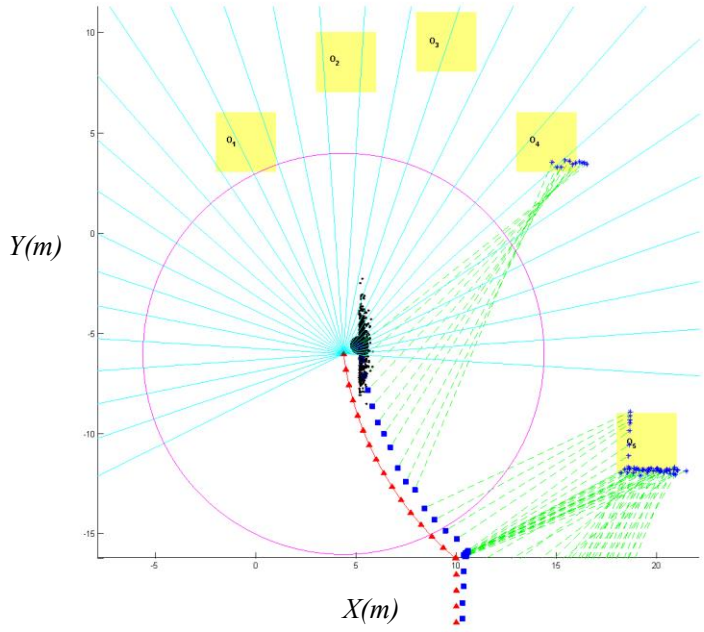
(d)

Figure 6.17 (a) *Prediction* of one step near obstacle O_5; (b) *Update* using measurement wrt obstacles; (c) *Update* after resampling; (d) *Prediction* of next step.

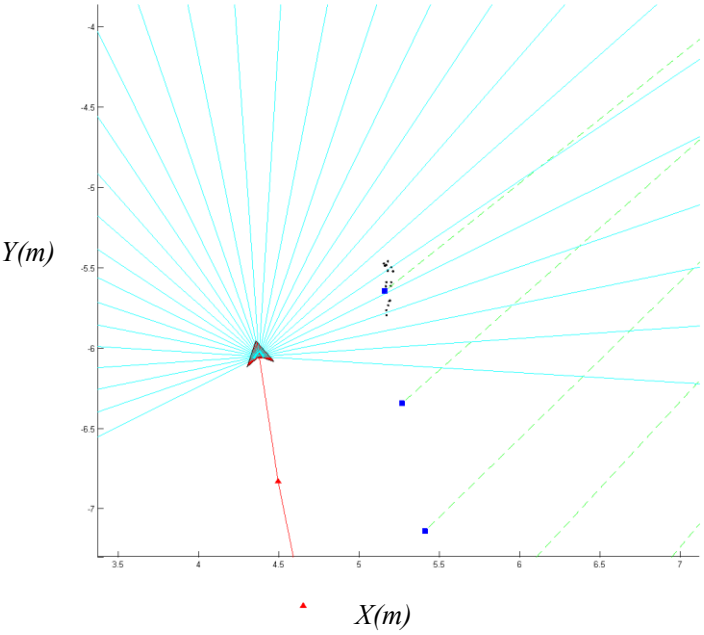
Figure 6.17 presents an illustration of the FastSLAM process with particle filtering when robot got close to obstacle O_5. In Figure 6.17, (a) and (d) present the *prediction* phase and (b) and (c) describe the *update* phase. In Figure 6.17 (a) particles form a cloud indicating proper position of the robot from the *prediction* phase of the last step. Figure 6.17 (b) presents the *update* phase based on the measurement with regard to obstacles, which in this case obstacle O_5. The area of each particle stands for its weight. The larger its area is, the higher its weight would be. We can see that there are some particles with much higher weights than others. The resampling process is illustrated in Figure 6.17 (c), particles with higher weights in last step survive, and particles with smaller weights are eaten, which we can see in Figure 6.17 (c) just a small part of particles survive to be ready to spread out. Finally, Figure 6.17 (d) presents the second *prediction* phase, which is derived from particles survived in last step propagating to the next generation.



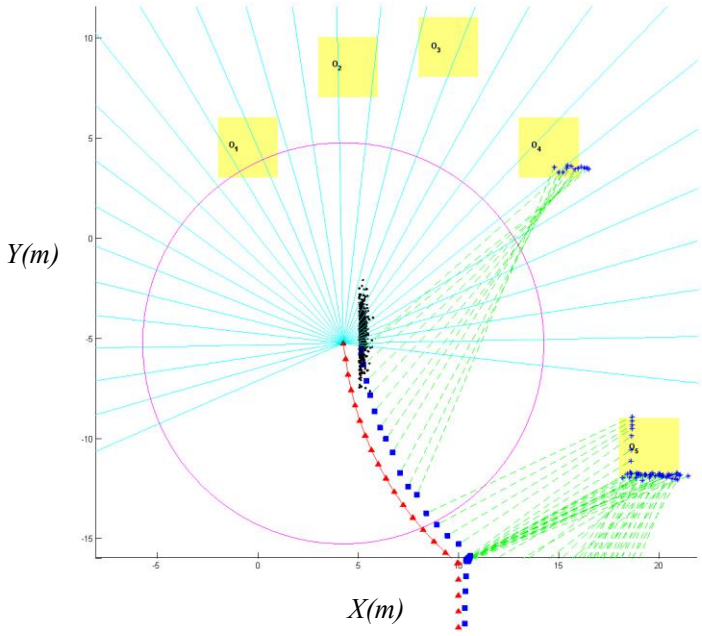
(a)



(b)



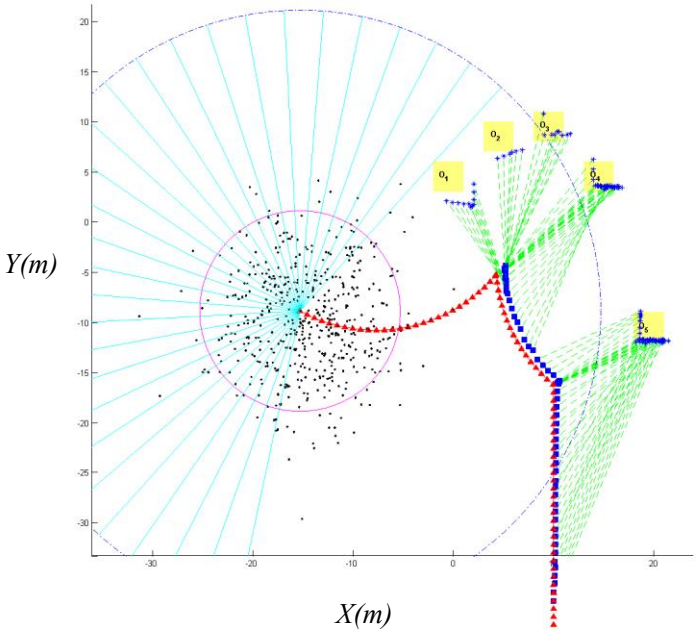
(c)



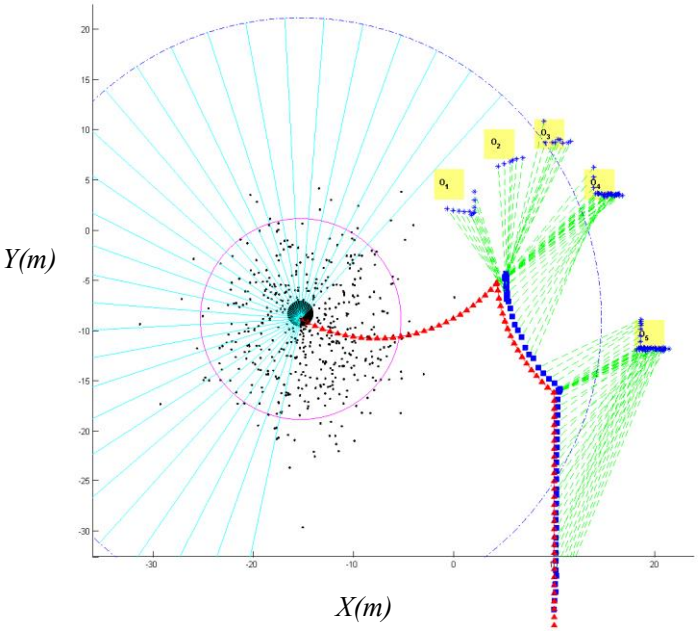
(d)

Figure 6.18 (a) *Prediction* of one step near obstacle O_1; (b) *Update* using measurement wrt obstacles; (c) *Update* after resampling; (d) *Prediction* of next step.

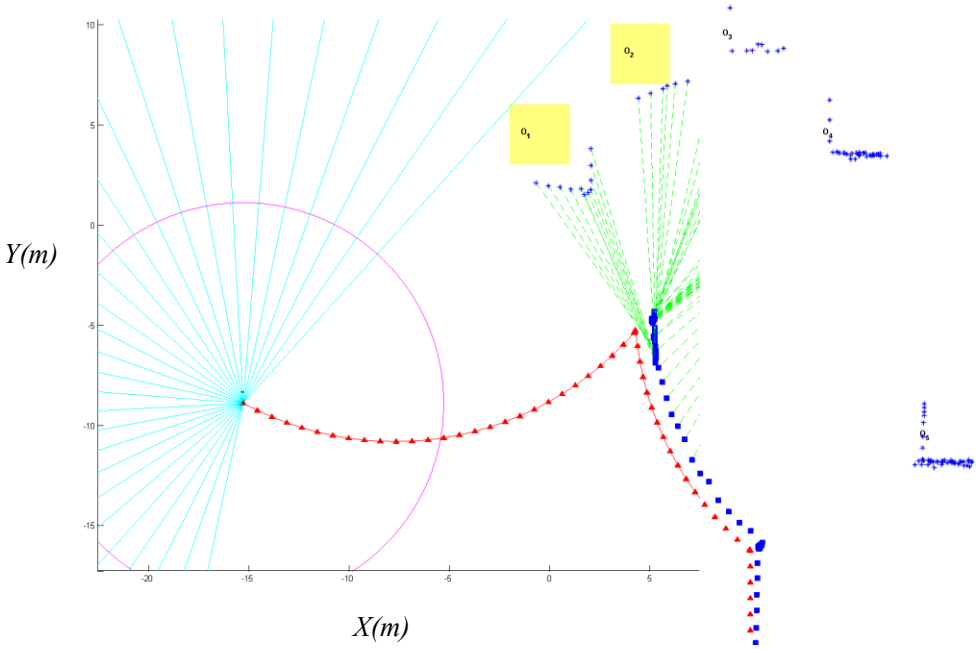
Figure 6.18 presents an illustration of the FastSLAM process with particle filtering when robot got close to obstacle O_1. As we can see when robot turned left to avoid obstacle O_5, it went to the concave trap due to affection by the attractive velocity of the goal. Likewise, robot chose turning left to escape the concave obstacle based on the local map it built with sensor data. The whole particle filtering process is the same as the process in Figure 6.17, so we will not present details.



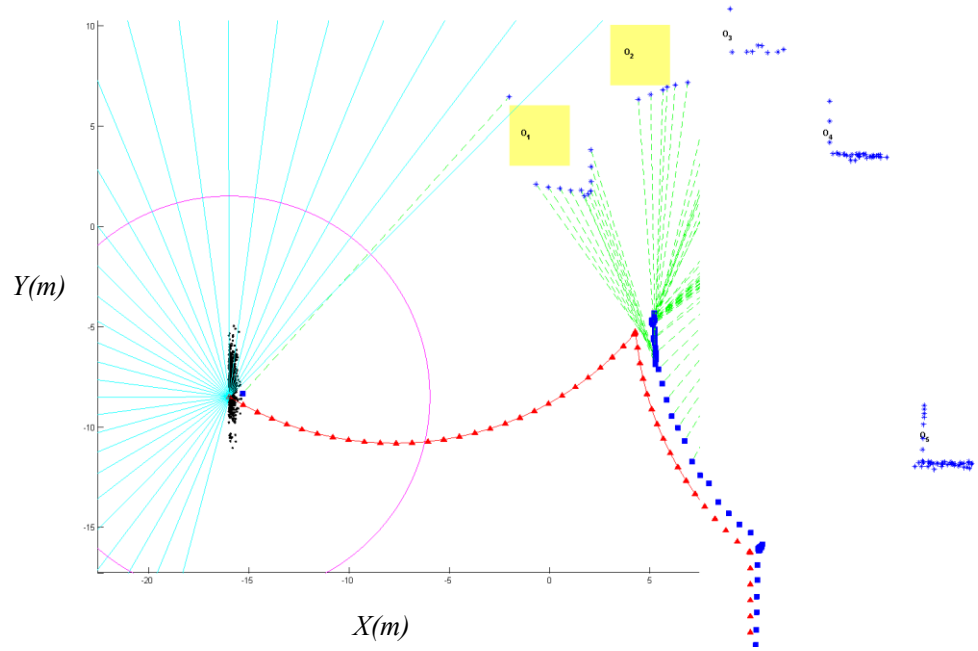
(a)



(b)



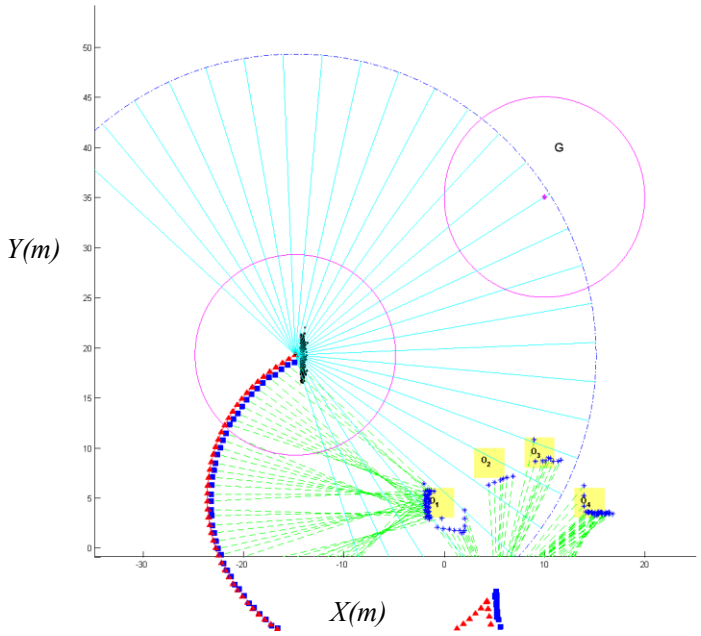
(c)



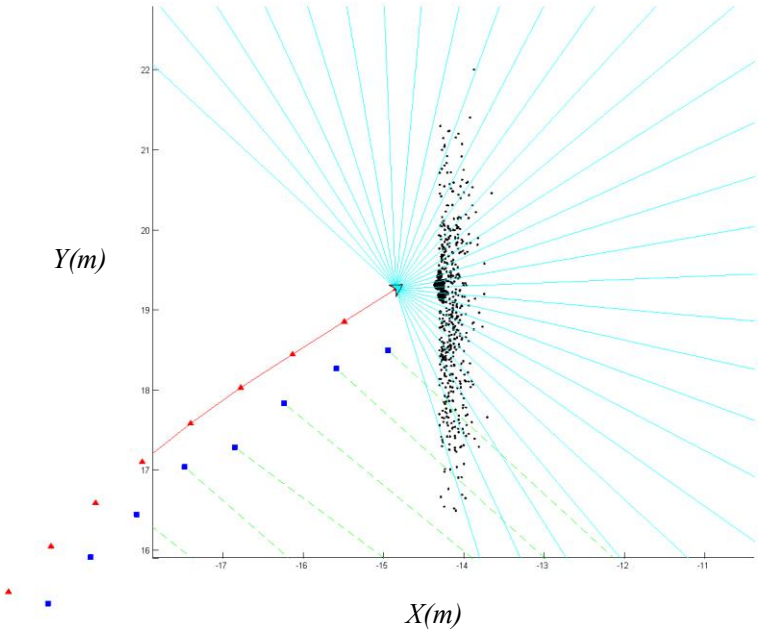
(d)

Figure 6.19 (a) *Prediction* of one step just sensing obstacle O_1 after losing obstacles and the goal; (b) *Update* using measurement wrt obstacles; (c) *Update* after resampling; (d) *Prediction* of next step.

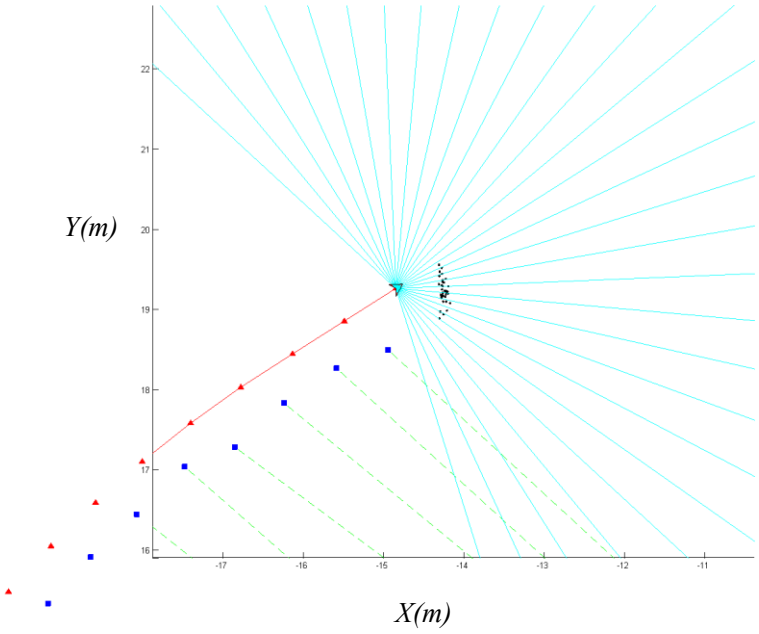
Figure 6.19 presents an illustration of the FastSLAM process with particle filtering when robot just sensed obstacle O_1 after losing obstacles and the goal. When robot successfully escaped the concave obstacle, the head of robot is far away from pointing the goal. Then robot had to make a "U-turn" to adjust it pointing direction to the goal. While during this process, robot lost detection of obstacles and the goal due to the sensing configuration of robot. Hence, particles will be spread out based on the system equations without resampling process based on the measurements. That is why we see the particles were scattered far away before robot detected obstacles again. However, once robot detected obstacle O_1 like in Figure 6.19 (b), *update* phase was implemented based on measurement wrt obstacle O_1. Figure 6. 19 (c) represents particles lived on, and Figure 6.19 (d) indicates particles propagated to the next generation.



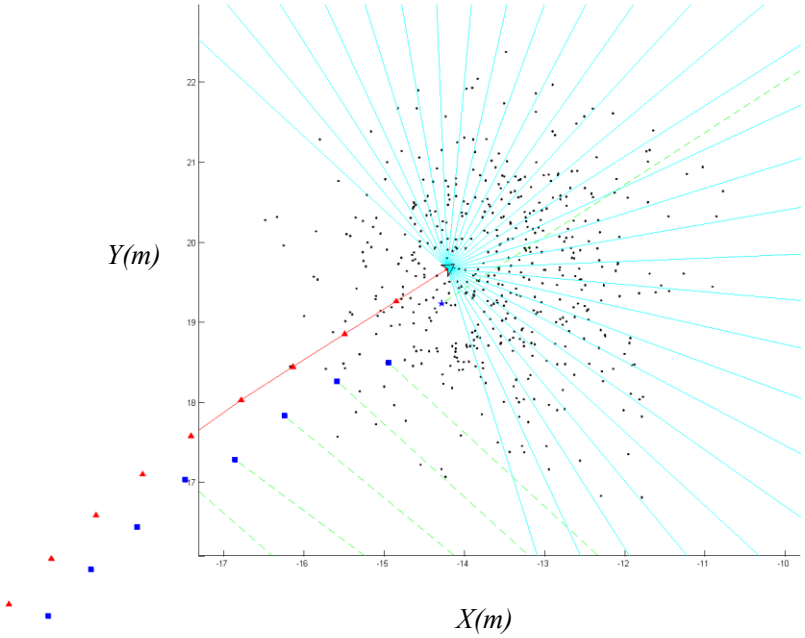
(a)



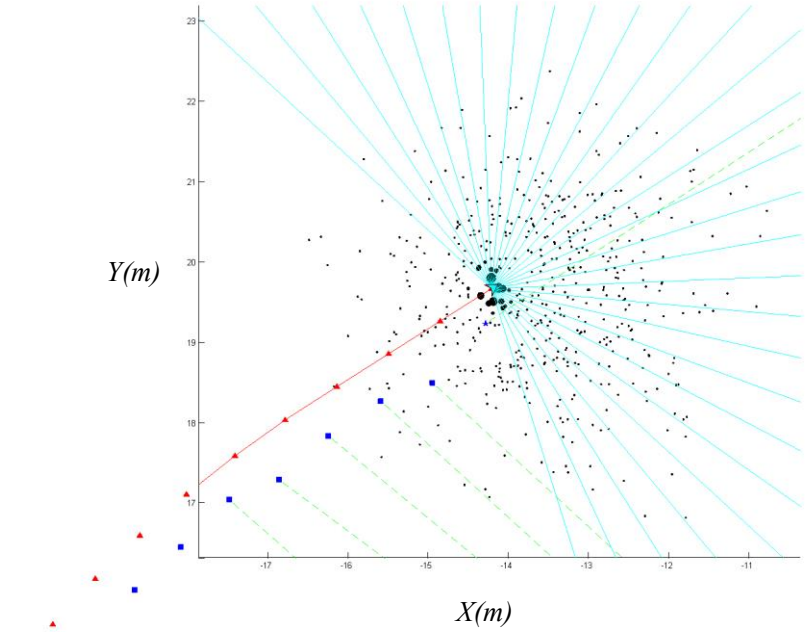
(b)



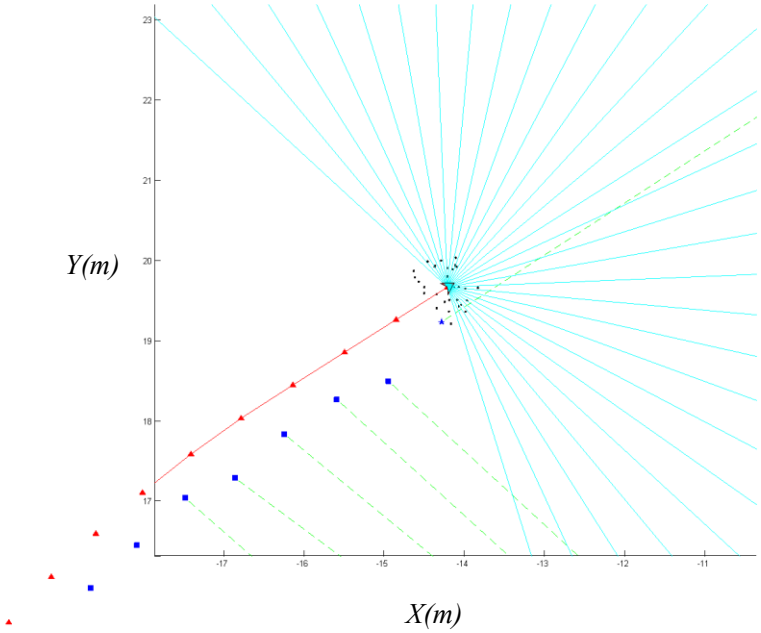
(c)



(d)



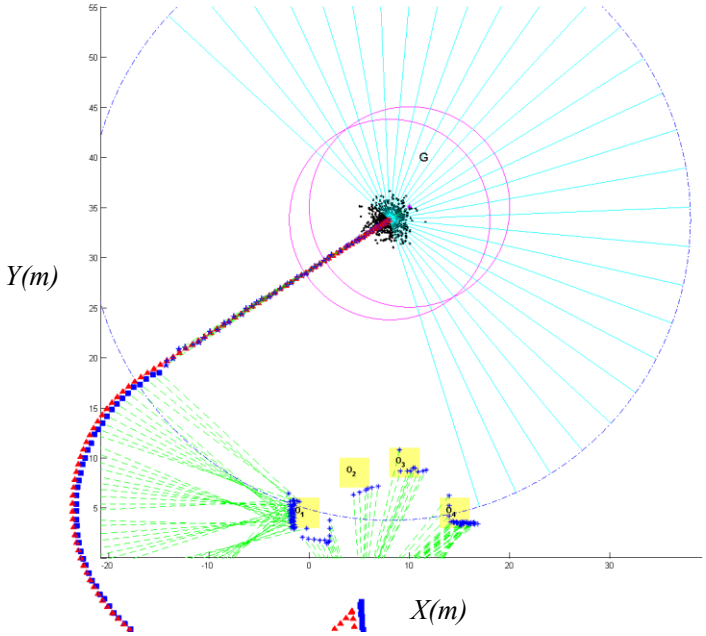
(e)



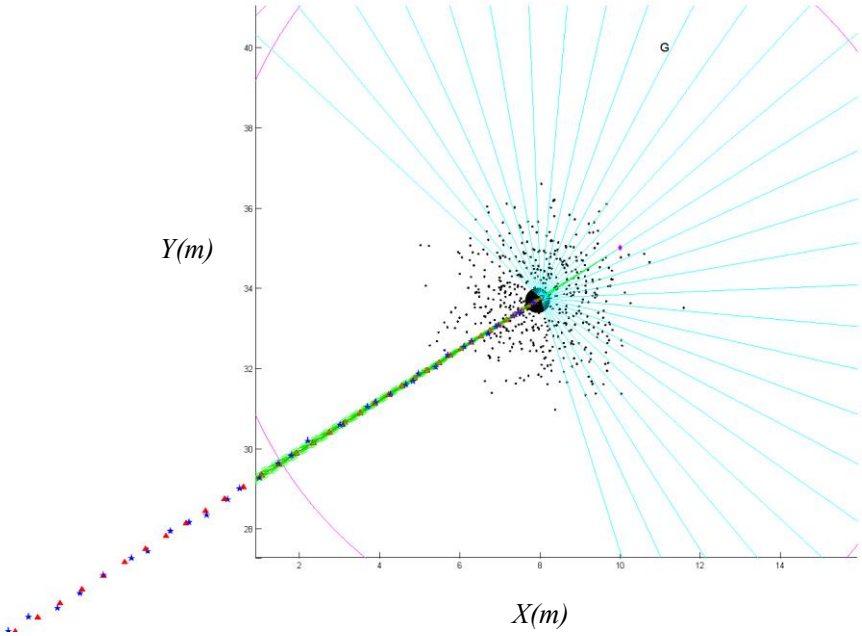
(f)

Figure. 6.20 (a) Prediction of one step where robot just sensing the goal. (b) Update using measurement wrt the goal. (c) Update after resampling. (d) Prediction of next step. (e) Update using measurement wrt the goal. (f) Update after resampling.

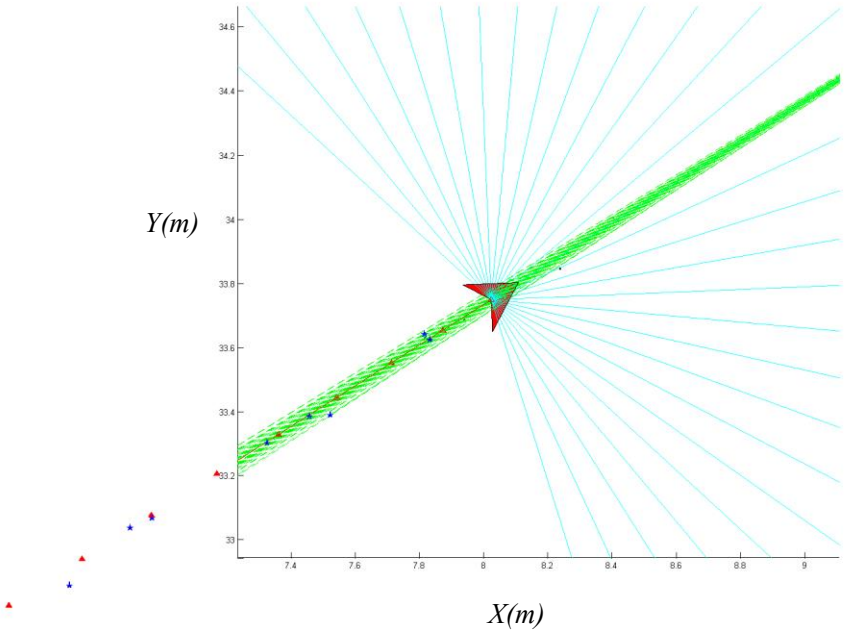
Figure 6.20 presents an illustration of the FastSLAM process with particle filtering when robot just sensed the goal, which is similar to Figure 6.10. Figure 6.20 (a) and (d) present the *prediction* phase and (b), (c), (e) and (f) describe the *update* phase. Details about how this process did, please refer to explanations about Figure 6.10.



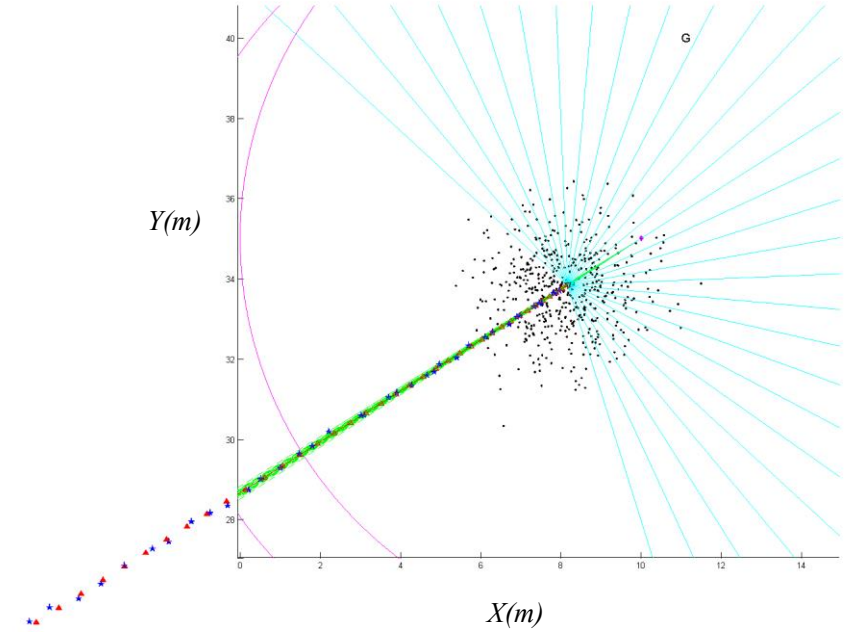
(a)



(b)



(c)



(d)

Figure 6.21 (a) *Prediction* of one step close to the goal. (b) *Update* using measurement wrt the goal. (c) *Update* after resampling. (d) *Prediction* of next step.

Finally, when robot moved to be really close to the goal, we illustrated some particle filtering snapshots as well in Figure 6.21. Since we have already explained the FastSLAM process with particle filtering in Figure 6.19 and 6.20, we do not talk more about it again. Likewise, Figure 6.21 (a) and (d) present the *prediction* phase and (b) and (c) describe the *update* phase based on the measurement wrt the goal. We can see that the closer robot is from the goal, the smaller residual error of robot ideal path and estimation of robot path is.

Chapter 7
Experimental Results

LEGO MINDSTORMS NXT robot has been discussed in Chap.5. The experiments were conducted by the help of MATLAB and LabVIEW implemented in LEGO NXT robot. We have introduced briefly MATLAB in chap.6, in this chapter we will present the software applied to be the development environment of experiments, i.e. LabVIEW.

Our experiments were accomplished by using LabVIEW and MATLAB. LabVIEW is applied to control the robot reaching the goal without obstacles collision, and collect data regarding to measurements about obstacles and the goal. After getting the measurement data, we utilized MATLAB to build the estimated map and the estimated robot path based on data we collected.

Section 7.1 introduces NI LabVIEW, LEGO MINDSTORMS NXT Module and how to combine them to create a NXT robot project. Section 7.2 describes how we implement our experimental algorithms. Finally, Section 7.3 illustrates all our experimental results.

7.1 Introduction of LabVIEW

LabVIEW, the abbreviation of Laboratory Virtual Instrument Engineering Workbench, is a development environment and system-design platform for a visual programming language from the company called National Instruments. The language LabVIEW uses is named the graphical language (“G”), which is a dataflow programming language. LabVIEW is usually used for instrument control (such as controlling our LEGO NXT robot), data acquisition and industrial automation on a variety of platforms including Microsoft Windows, Linux, various versions of UNIX and Mac OS X.

LabVIEW 2013 was utilized in our experiments in order to control LEGO NXT robot. LabVIEW programs are called virtual instruments (VIs). Each VI has three components: a

front panel, a block diagram and a connector pane. The front panel is a user interface built using controls and indicators. Controls are inputs allowing a user to supply information to the VI, and indicators are outputs indicating and displaying the results based on the inputs given. The back panel is a block diagram containing the graphical source code which is our main program. The connector pane is used in a subVI (similar to a sub function) in order to define how the inputs and outputs of the subVI appear on a VI and how the inputs and outputs relate to the controls and indicators. Details about LabVIEW 2013, please refer to Appendix C.1.

7.1.1 LEGO MINDSTORMS NXT Module

National Instruments supports plenty of add-ons of LabVIEW 2013 in order to help LabVIEW adapt to various application platform. For example, it developed NI LabVIEW Module for LEGO MINDSTORMS. With the help of this module, we can control LEGO NXT robot directly by LabVIEW.

This module includes the following features designed specifically for LEGO MINDSTORMS [44]:

- Robot Project Center: Incorporate lesson plan content and share student results in one place.
- Remote Control Editor: Visually configure and control your NXT using a joystick or keyboard.
- Piano Player: Play your own sounds and songs on the NXT device.
- NXT Terminal: Manage NXT programs and memory right from your screen.
- Remote Display: View all of your NXT screens and buttons on your computer monitor.
- Schematic Editor: Graphically configure and test motor and sensor connections.
- Sensor Viewer: View the data from your project sensors in real time.
- Data Viewer: Easily log and analyze the data you collect from your NXT.
- Picture Editor: Create your own images to display on the NXT screen.

We can download this module on the official website of National Instruments in Support & Services section. Firstly, we need download LabVIEW 2013 also on the official website of National Instruments where it is very easily to be found. Secondly, install LabVIEW 2013 on your PC or laptop properly. Finally, install the module we downloaded already on LabVIEW

2013. After finishing these steps, we should see there is an item called "New NXT" in the dropdown menu of "File" in the Menu Bar. Figure 7.1 illustrates the start menu of LabVIEW with LEGO MINDSTORMS NXT Module add-on successfully installed.

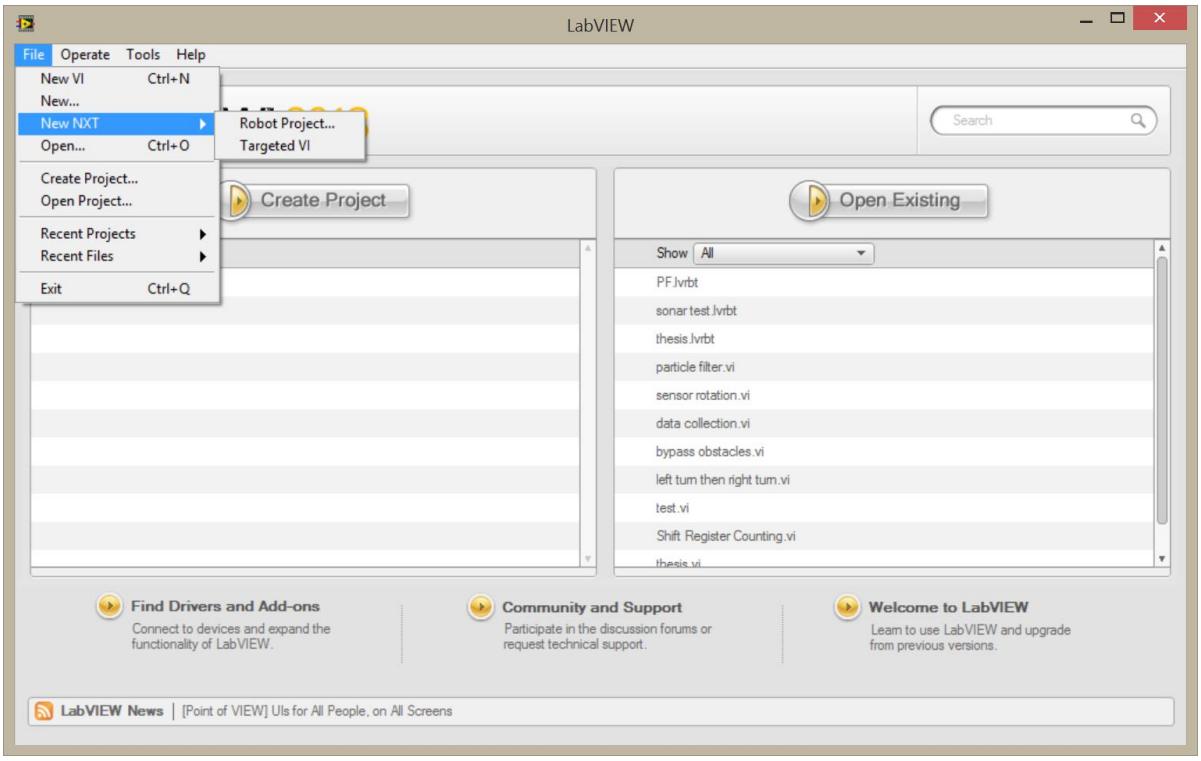


Figure 7.1 Start menu of LabVIEW with LEGO MINDSTORMS NXT Module.

7.1.2 LabVIEW Elements

With LEGO MINDSTORMS NXT Module successfully installed, we can create a robot project to configure and control our LEGO NXT robot. Details about how to configure our robot with LabVIEW can be found in Appendix C.2.

After configuring our robot with LabVIEW, we can write our own codes to control the NXT robot. Here we want to give a brief introduction to our LabVIEW basic elements we will use. Table 7.1 illustrates certain basic LabVIEW blocks with their meanings.

LabVIEW graphical block	Block meaning
	Reads the ultrasonic sensor
	Turns on the motor and move it forward
	Turns on the motor and move it backward
	Stops the motor by braking
	Turns on the motor and drives a specific distance (in degrees)
	Exponential function
	A subVI icon created by a user

Table 7.1 Introduction of basic LabVIEW graphical blocks with NXT robot.

7.2 Experimental Algorithm

The experiments were conducted using both LabVIEW and MATLAB. LabVIEW 2013 with LEGO MINDSTORMS NXT Module was utilized to robot navigation with obstacle avoidance. Meantime, we also need obtain the local map and the estimations of robot path, which cannot be accomplished by just using LabVIEW. In order to deal with this issue, we implemented MATLAB cooperated with LabVIEW together to overcome it.

Our algorithms we proposed have been briefly explained in Chap. 3 and Chap. 4, while the algorithms in our experiments are the same except that we implement them through two softwares, namely LabVIEW and MATLAB. In order to better illustrate our algorithms for the experiment, we show our experimental algorithm flow chart in Figure 7.2.

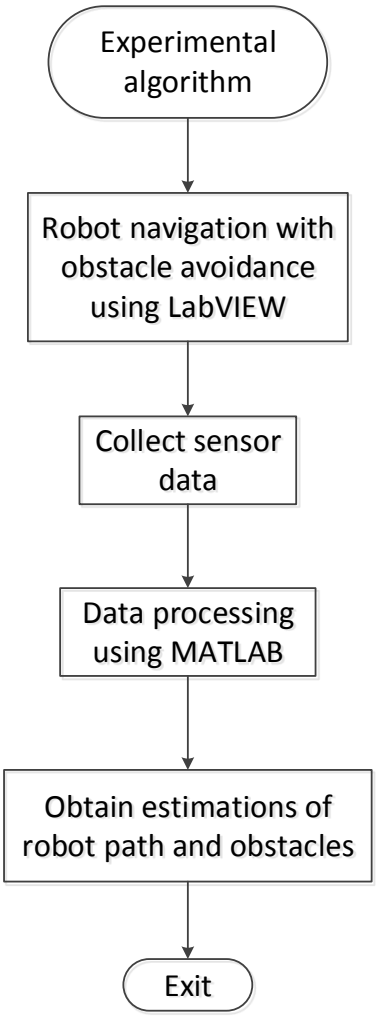


Figure 7.2 The algorithm of experiments.

In Figure 7.2, we can see that our algorithms include two parts, one is robot navigation with obstacles avoidance using proposed artificial velocity fields method which is accomplished through LabVIEW; the other one is robot navigation using the FastSLAM approach done by MATLAB after processing data collected from sensors.

7.3 Experimental Results

In the experiments, we consider two scenarios as well like what we have done in simulations. One is robot navigation with two obstacles with distinct dimension differences; the other is robot navigation with five same dimension obstacles composed of a big concave obstacle with dissymmetrical shape.

7.3.1 Robot Travelling Around Two Obstacles

In this section, experiment about robot travelling around two obstacles will be illustrated as two parts. The first part is regarding to robot navigation with two obstacles avoidance using the velocity potential field approach accomplished by LabVIEW. During this navigation process, we collected sensor data for each robot step and recorded them. Due to the accuracy of the rotation sensor imbedded in the LEGO servo motor, we recorded the sensor data every $\pi/8$ of sensor rotation.

The second part of our experiment is obtaining the estimations of robot path and obstacles based on the sensor data we collected in order to build the local map. This was carried out by using MATLAB.

7.3.1.1 Robot Navigation with Two Obstacles Avoidance

In Figure 7.3, the grey cylinder in the left top corner of the snapshot indicates the goal robot will travel towards. Between the robot and the goal lies two obstacles, the blue trash bin has bigger dimension than the small obstacle which is denoted as a wood block. Robot will turn left when close to obstacles since robot can build a local map of obstacles then choose proper direction to turn in order to save time and energy.

10 snapshots are shown in Figure 7.3. We can see our controller successfully drove the robot avoiding obstacles and reaching the goal finally. Meantime, turning left is a proper turning direction based on obstacles dimensions shown in the Figure 7.3.

(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

(i) (j)

Figure 7.3 Robot navigation with two obstacles avoidance reaching the goal.

7.3.1.2 Robot Navigation Using the FastSLAM Approach with Two Obstacles Avoidance

Sensor data were collected in the robot navigation process shown in Figure 7.3. Table 7.2 lists all the sensor data for each robot step.

Robot moving step	Angle measurements chosen (based on local coordinate system of robot)	Distance measurements (in centimeters)
1	3*π/8	77
2	3*π/8	50
3	3*π/8	25
4	0	29
5	-π/8	22
6	-π/8	29
7	-π/8	18
8	-π/8	21
9	-π/8	50
10	π/2	59
11	π/2	26
12	π/2	6

Table 7.2 Data collected during robot navigation process with two obstacles avoidance.

In Table 7.2, the first nine robot moving steps were conducted experiment based on measurements with regard to obstacles, and the rest was about measurements about the goal. That is because when robot was moved at the tenth step, it sensed the goal already. Through the data collected, we can build a map with the estimations of robot positions and the local map robot sensed in Figure 7.4.

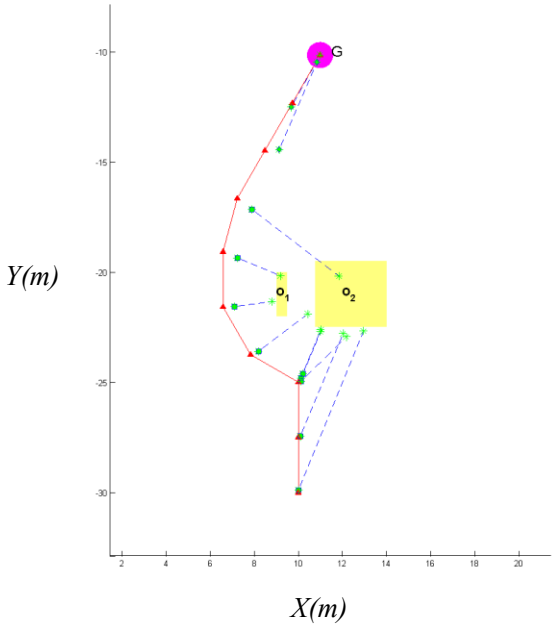


Figure 7.4 Estimations of robot positions and two obstacles.

In Figure 7.4, the magenta circle indicates the goal robot will travel to. The red solid line indicates the ideal robot path to the goal without obstacles collision, the green square denotes the estimation with regard to obstacles robot sensed, and the green diamond illustrates the estimation with regard to the goal. The green asterisk sign means the local map robot built based on the sensor data. We can see that when robot detects the goal, the estimation of robot position performs better than that based on the measurements with regard to obstacles which we do not know the absolute positions of. However, we know the absolute position of the goal in the global map. We can see obstacle O_2 is bigger than O_1, therefore robot will choose turning left to avoid obstacles. The particle clouds representation of Figure 7.4 is shown in Figure 7.5.

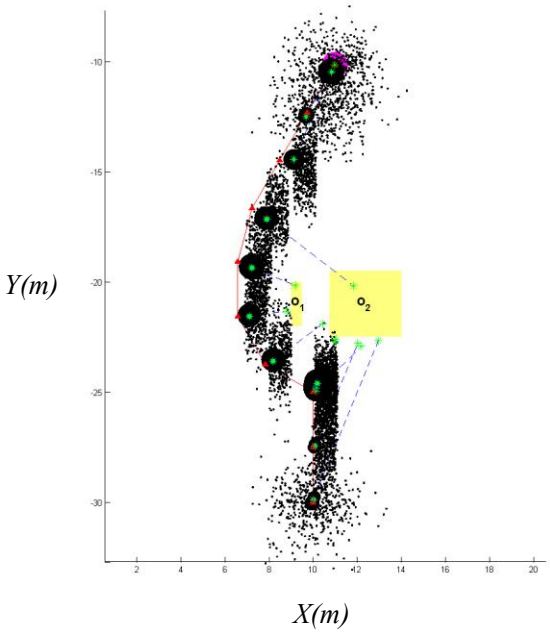


Figure 7.5 Particle clouds representation of Figure 7.4.

7.3.2 Robot Travelling Around a Concave Obstacle

In this section, experiment about robot travelling around a concave obstacle will be shown as two parts as well. The first part is regarding to robot navigation with a concave obstacle composed of five same dimensioned obstacles avoidance using the velocity potential field approach accomplished by LabVIEW. During this navigation process, we collected sensor data for each robot step and recorded them. Due to the accuracy of the rotation sensor imbedded in the LEGO servo motor, we recorded the sensor data every $\pi/8$ of sensor rotation also.

The second part of our experiment is obtaining the estimations of robot path and obstacles based on the sensor data we collected in order to build the local map. This was carried out by using MATLAB.

7.3.2.1 Robot Navigation with a Concave Obstacle Avoidance

10 snapshots in Figure 7.6 show the robot navigation process with a concave obstacle avoidance. We can see our controller successfully drove the robot escaping the concave obstacle with efficiency and reaching the goal finally. Meantime, turning left is a proper turning

direction based on obstacles dimensions shown in the Figure 7.6.

(a)

(b)

(c)

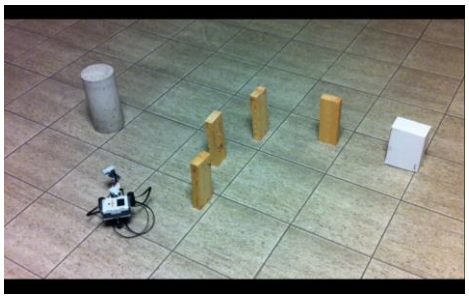
(d)

(e)

(f)

(g)

(h)



(i)

(j)

Figure 7.6 Robot navigation with a concave obstacle avoidance reaching the goal.

In Figure 7.6, the grey cylinder in the left top corner of the snapshot indicates the goal robot will travel towards. Between the robot and the goal lies five obstacles composing a big concave obstacle. Four wood blocks and a paper box denote five obstacles with dissymmetry. When robot moved close to obstacles, it will have its surrounding environment map, then it will have obstacles dimensions on two sensing sections of robot which we have discussed before. Based on the dimensions of obstacles wrt two sections of robot, robot will choose the optimal turning direction to avoid obstacles faster and reach the goal quicker, which in this particular case, robot chose turning left and finally reached the goal.

7.3.2.2 Robot Navigation Using the FastSLAM Approach with a Concave Obstacle Avoidance

Table 7.3 lists sensor data obtained during robot navigation escaping a concave obstacle. The measurements about the first nine robot moving steps were based on measurements with regard to obstacles, and the rest was about measurements about the goal. Through the data obtained, we can build a map with the estimations of robot positions and the local map robot sensed based on them in Figure 7.7.

Robot moving step	Angle measurements chosen (based on local coordinate system of robot)	Distance measurements (in centimeters)
1	π/8	53
2	0	49
3	0	53
4	0	42
5	-π/8	41
6	0	12
7	-π/8	18
8	-π/8	35
9	-π/8	42
10	π/2	55
11	π/2	25
12	π/2	7

Table 7.3 Data collected during robot navigation process with a concave obstacle avoidance.

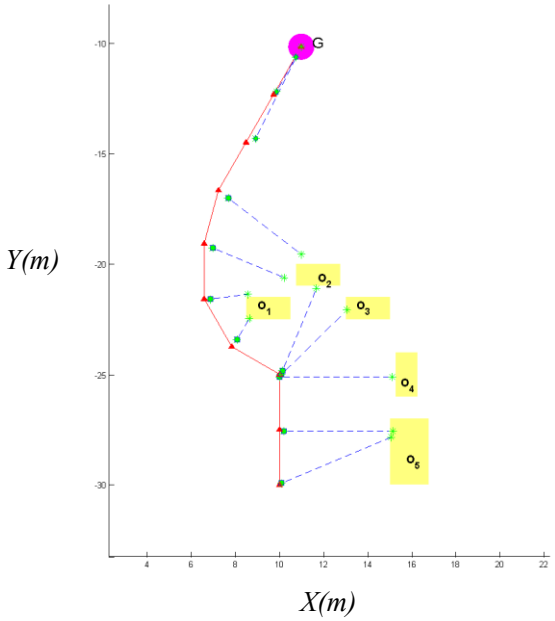


Figure 7.7 Estimations of robot positions and five obstacles.

In Figure 7.7, the meanings of corresponding signs are denoted the same as ones in Figure 7.4. We can see that when robot senses the goal, the estimation of robot position performs better than that based on the measurements with regard to obstacles which we do not know the absolute positions of. However, we know the absolute position of the goal in the global map, which means estimations with regard to a point we know the absolute position of are more

accurate than ones about a point we do not know the absolute position of.

In Figure 7.7, we can see that a concave obstacle consists of obstacles O_1, O_2, O_3, O_4 and O_5. The concave obstacle they formed has a dissymmetric shape which has a small opening. The robot built its surrounding environment into a local map and found the small opening of the concave obstacle. Then the controller drove the robot go through the small opening of the concave obstacle. The particle clouds representation of Figure 7.7 is shown in Figure 7.8.

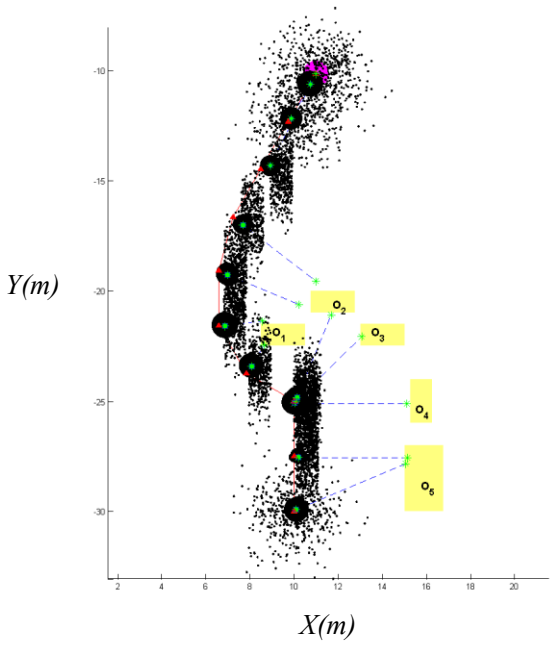


Figure 7.8 Particle clouds representation of Figure 7.7.

Chapter 8
Conclusions

8.1 Summary

In this thesis, we proposed a switching controller for robot navigation without obstacles collision. This controller is used to determine proper turning direction for the robot with the help of applying a modified velocity potential field approach. The robot will choose proper turning direction based on the local map of its surrounding environment built from sensor measurements.

The feasibility of this switching controller is verified in two representative scenarios. One is for the that robot travels towards the goal when there are two obstacles with distinct dimension difference lying in between its path; the other one is the case when in between robot path to the goal lie five obstacles emulating a single big concave obstacle. In the first scenario, the robot obtained the two obstacles dimension, and chose a proper turning direction avoiding them efficiently using the proposed controller. In the same way, for the second scenario, the robot also build its local map of obstacles based on sensing, found a shorter path around the concave obstacle, escaped successfully and finally reached the goal.

After successfully testing our controller in robot navigation with obstacle avoidance, we applied the FastSLAM approach into the navigation process. This is justified because in robot navigation process, noises do indeed exist. We took a consideration both the system noise and the measurement noise. The FastSLAM approach was used in our research for obtaining estimated robot path and estimation with regard to obstacles in order to help building the local map of robot's surrounding environment.

In our research, the initial pose of the robot and the absolute position of the goal are assumed known. Estimation of robot path regarding to obstacles is executed using a particle filter; the measurements relative to obstacles are obtained but we do not know the absolute position of

in the global coordinate system. When robot bypassed obstacles and sensed the goal, estimated robot path will be obtained with regard to the measurements about the goal, for which we know the absolute position. The results show that robot path estimation wrt the goal performs better than that wrt obstacles without known absolute positions. When the robot gets close to the goal, the estimated robot positions are more and more close to the ideal robot expected path. In fact, estimated robot path with regard to obstacles has larger residual error than the wrt the goal. After obtaining estimation of robot path, we can get estimated positions of obstacles based on the measurements regarding to obstacles after removing the measurement noise. Therefore, we obtained the estimation over robot path and about unknown obstacles simultaneously.

The proposed algorithm was tested in simulations and experiments. For simulations, MATLAB was utilized. And for experiments we used both LabVIEW and MATLAB. LabVIEW was used to control and communicate with LEGO MINDSTORMS NXT robot to implement obstacle avoidance algorithm along with collecting measurement data for each robot step. MATLAB was used to obtain estimation over robot path and obstacles based on the system model and measurement data.

8.2 Main Research Contributions

The proposed robot navigation using the FastSLAM approach with obstacle avoidance derived from a modified velocity potential field method has been presented in this thesis. The design of a switching controller improved the efficiency of robot navigation with obstacles avoidance. This switching controller allowed robot to find an efficient turning direction to avoid obstacles based on the local map it built.

At the process of building the local map, the FastSLAM approach was proposed to be embedded in the algorithm. The proposed FastSLAM approach was not only been applied to obtain estimation with regard to obstacles we do not know the absolute position of in the global coordinate system, and also to obtain the estimation corresponding to the goal for which we already know the absolute position. Not knowing the absolute position of obstacles, we only can obtain the approximate positions of obstacles. We used ideal robot pose and ideal measurement without noises to obtain empirical positions of obstacles. Then taking the system noises and measurement noises into consideration, we obtained optimal estimated robot path

based on the empirical positions of obstacles. Technically, this process was accomplished using a particle filter. After receiving estimation of robot path, we can obtain optimal estimated obstacles positions. This was carried out by combining the estimation of robot positions and the non-noisy measurements for each robot step.

The results in our simulations illustrate that through our proposed FastSLAM approach, robot path estimation regarding to the goal with known absolute position performs better than robot path estimation with regard to obstacles with unknown absolute positions. The residual errors between the estimated robot path and the ideal robot path become smaller and smaller as the robot traveled towards the goal. Finally, when robot reached the goal, the estimated robot position and the ideal robot position converge. Experiments were conducted in LEGO NXT robot and verified the simulation results.

8.3 Future Work

In our research, estimation of robot path was obtained by using a particle filter and estimation of obstacles (landmarks) and was executed using the measurement without noise conditioned on the estimated robot path. For the future work, Kalman filters can be applied into estimation of obstacles conditioned on estimated robot path in order to obtain more accurate results. Also, in each robot moving step, we have many measurements from the bearing sensor of robot. However, for the FastSLAM approach, we just took one measurement into consideration for each moving step for mathematical convenience and efficiency. Next prospective research could be thinking of all measurements for each robot step and obtaining multiple estimations of obstacles in each moving step.

In our simulations and experiments, just two scenarios were taken into consideration. Both the two scenarios are just about stationary obstacles. For the next step, more complicated scenarios could be considered such as accounting for moving obstacles, human, even a moving human. In complex environments, our robot controller may need to be upgraded to higher level in order to deal with these scenarios accurately and efficiently.

We can also improve and adjust our experiment design and setup. For example, in our experiment, just one ultrasonic sensor was utilized, and the angle measurement was implemented by a rotation sensor embedded into a NXT servo motor. In practice, the rotation

sensor attached in a servo motor was not accurate without stability. In the future work, we can think about using another sensor such as the angle sensor from the third party HiTechnic instead of the rotation sensor embedded in a servo motor if possible, since the HiTechnic Angle Sensor has one degree accuracy dominating the attached rotation sensor we used.

References

[1] S.B. Niku. *Introduction to Robotics – Analysis, Systems, Applications.* California: John Wiley & Sons Inc., 2011, pp. 3-5.

[2] I.J. Cox and G.T. Wilfong. *Autonomous Robot Vehicles.* Springer-Verlag New York Berlin Heidelberg, 1990, pp. xix-xxi.

[3] D. Fletcher. *Matilda Infantry Tank (New Vanguard 8).* Oxford: Osprey Publishing, 1938–45, pp. 40.

[4] K.P. Valavanis, D. Gracanin, M. Matijasevic, R. Kolluru and G.A. Demetriou. "Control Architectures for Autonomous Underwater Vehicles," Control Systems, IEEE(vol.:17, iss.:6), Dec 1997, pp. 48-64.

[5] S.W. Kandebo. *AW&ST and AUVSI 1997-98 International Guide to Unmanned Vehicles.* New York: McGraw-Hill, Inc., 1997, pp. 83-117.

[6] S. Mastellone et al. "Formation Control and Collision Avoidance for Multi-agent Non-holonomic systems: Theory and Experiments". *The International Journal of Robotics Research,* 2008, pp.107-126.

[7] J. Liu and J. Wu. *Multi-Agent Robotics Systems*. CRC Press, 2001.

[8] R. Siegwart and I. Nourbakhsh. *Introduction to Autonomous Robots,* MIT Press, 2004.

[9] A.V. Topalov. *Recent Advances in Mobile Robotics.* InTech, Dec 14, 2011.

[10] P. Reignier. "Fuzzy logic techniques for mobile robot obstacle avoidance," *Robotics and Autonomous Systems*, vol. 12, iss. 3-4, April 1994, pp.143-153.

[11] J. Borenstein and Y. Koren. "Real-time Obstacle Avoidance for Fast Mobile Robots in Cluttered Environments," *Proceedings of the 1990 IEEE International Conference on Robotics and Automation,* pp. 572-577.

[12] D. Wang. "A Generic Force Field Method for Robot Real-time Motion Planning and Coordination." PhD dissertation, University of Technology, Sydney, Australia, 2009.

[13] A.A. Masoud, "Using hybrid vector-harmonic potential fields for multi-robot, multi-target navigation in a stationary environment," *Proceedings of the IEEE International Conference on Robotics and Automation*, Minneapolis, Minnesota, 1996, pp. 3564-3571.

[14] H. Seraji and B. Bon, "Real-time collision avoidance for position-controlled manipulators," *IEEE Transactions on Robotics and Automation,* vol. 15, pp. 670-677, 1999.

[15] M.G. Park and M.C. Lee, "Artificial Potential Field Based Path Planning for Mobile Robots Using a Virtual Obstacle Concept," *Proceedings of the 2003 IEEE/ASME International Conference on Advanced Intelligent Mechatronics,* 2003, vol. 2, pp. 735-740.

[16] O. Khatib**,** "Real-time Obstacle Avoidance for Manipulators and Mobile Robots," *International Journal of Robotics Research,* vol. 5, no. 1, pp 90-98, 1986.

[17] A.K. Das, R. Fierro, R.V. Kumar, J.P. Ostrowski and J. Spletzer. "A Vision-Based Formation Control Framework," *IEEE Transactions on Robotics and Automation,* volume 18, Issue 5, pp. 813-825, 2002.

[18] R. Smith, M. Self, and P. Cheeseman. Estimating uncertain spatial relationships in robotics. In *Autonomous Robot Vehnicles*, Springer, 1990.

[19] I. Horswill. "Polly: A Vision-Based Artificial Agent", *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAA1'93),* Washington DC, MIT Press, July 11-15, 1993.

[20] E.D. Dickmanns and A. Zapp, "Guiding Land Vehicles along Roadways by Computer Vision", *Congres Automatique,* 1985, pp. 233-244.

[21] C.J. Taylor, J. KoSeckd, R. Blasi and J. Malik, "A comparative study of vision-based lateral control strategies for autonomous highway driving," *The Int. J. Robot. Research,* vol. 18, no. *5*, pp, 42-453, 1999.

[22] M. Faisal, R. Hedjar, M.A. Sulaiman and K. Al-Mutib, "Fuzzy Logic Navigation and Obstacle Avoidance by a Mobile Robot in an Unknown Dynamic Environment," *International Journal of Advanced Robotic Systems,* Vol. 10, InTech, 2013.

[23] S.H. Lian, "Fuzzy Logic Control of an Obstacle Avoidance Robot," *Proceedings of the Fifth IEEE International Conference on Fuzzy systems*, volume 1, pp. 26-30, 1996.

[24] http://en.wikipedia.org/wiki/EKF_SLAM#cite_note-Montemerlo2002-1. [Accessed: 11-Dec-2014]

[25] G. Dissanayake, P. Newman, S. Clark, H.F. Durrant-Whyte, and M. Csorba. An experimental and theoretical investigation into simultaneous localisation and map

building (SLAM). *Lecture Notes in Control and Information Sciences: Experimental Robotics VI*, Springer, 2000.

[26] J. Guivant and E. Nebot. "Optimization of the simultaneous localization and map building algorithm for real time implementation," *IEEE Transaction of Robotic and Automation*, May 2001.

[27] J.J. Leonard and H.J.S. Feder. A computationally efficient method for large-scale concurrent mapping and localization. *ISRR-99*.

[28] K.F. Uyanik, "A study on Artificial Potential Fields", Middle East Technical University, Ankara, Turkey, vol. 2, no. 1.

[29] L.M. Milne-Thomson. *Theoretical Hydrodynamics*. Macmillan, 1962.

[30] H. Chanson. *Applied Hydrodynamics.* CRC Press, 2009.

[31] H.K. Khalil. *Nonlinear Systems*. Prentice Hall, Inc., Upper Saddle River, NJ, 2002.

[32] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4, 1997.

[33] S. Thrun, D. Fox, and W. Burgard. A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning*, 31, 1998.

[34] M. Montemerlo, S. Thrun, D. Koller and B. Wegbreit. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem.

[35] http://en.wikipedia.org/wiki/Likelihood. [Accessed: 03-Oct-2014]

[36] A. Svensson. "A introduction to particle filters", Department of Information Technology, Uppsala University, Uppsala, Sweden.

[38] G. Nie, "Quasi-Harmonic Function Approach to Human-Following Robots," M.S. thesis, Dept. of Graduate Studies in Mechanical Engineering, Univ. Ottawa, Ottawa, ON, 2014.

[39] "What's NXT? LEGO Group Unveils LEGO MINDSTORMS NXT Robotics Toolset at Consumer Electronics Show (Press release)". Las Vegas, NV: LEGO Group. January 4, 2006.

[40] LEGO MINDSTORMS NXT User Guide.

[41] http://en.wikipedia.org/wiki/Speed_of_sound. [Accessed: 29-Nov-2014].

[42] http://www.generationrobots.com/en/content/65-ultrasonic-sonar-sensors-for-robots. [Accessed: 29-Nov-2014].

[43] http://nxttime.wordpress.com/2012/09/12/the-ultrasonic-sensor/. [Accessed: 29-Nov-

2014].

[44] http://sine.ni.com/nips/cds/view/p/lang/en/nid/212785. [Accessed: 7-Dec-2014].

[45] E. Pruner, D. Necsulescu, J. Sasiadek and B. Kim. "Control of Decentralized Geometric Formations of Mobile Robots," *17th International Conference on Methods and Models in Automation and Robotics (MMAR),* pp. 627-632, 2012.

[46] E. Pruner, "Control of Self-Organizing and Geometric Formations," M.S. thesis, Dept. of Mechanical Engineering, Univ. Ottawa, Ottawa, ON, 2013.

[47] D. Simon. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches.* Hoboken, N.J. : Wiley-Interscience, 2006.

[48] P. Newman. "On the Structure and Solution of the Simultaneous Localisation and Map Building Problem." PhD dissertation, Univ. of Sydney, 2000.

[49] K. Murphy. Bayesian map learning in dynamic environments. *NIPS-99*.

[50] A. Doucet, S. Godsill, and C. Andrieu. On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, 1999.

[51] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte Carlo localization: Efficient position estimation for mobile robots. In *AAAI*, 1999.

[52] D. Fox, W. Burgard, and S. Thrun. Active Markov localization for mobile robots. *Robotics and Autonomous Systems*, 1998.

[53] Z. Ghahramani and M. Jordan. Factorial Hidden Markov Models. *Machine Learning*, 29:245-273, 1997.

[54] S. Koenig and R. Simmons. Unsupervised learning of probabilistic models for robot navigation. In *ICRA*, 1996.

[55] J. Liu and R. Chen. Sequential Monte Carlo methods for dynamic systems. *JASA*, 93:1032-1044, 1998.

[56] A. Doucet, N. de Freitas, K. Murphy, and S. Russell. Rao-Blackwellised particle filtering for dynamic Bayesian networks. *UAI-2000*.

[57] K. Murphy and S. Russell. Rao-Blackwellized particle filtering for dynamic bayesian networks. In *Sequential Monte Carlo Methods in Practice*, Springer, 2001.

[58] F. Dellaert, D. Fox, W. Burgard, and S. Thrun. Monte Carlo localization for mobile robots. *ICRA-99*.

[59] A. Doucet, J.F.G. de Freitas, and N.J. Gordon, editors. *Sequential Monte Carlo Methods*

In Practice. Springer, 2001.

[60] Arulampalam, M.S., Maskell, S., Gordon, N. & Clapp, T. "A tutorial on particle filters for online nonlinear/non-gaussian Bayesian tracking," *IEEE Transactions on Signal Processing*, 50(2):174-188, 2002.

[61] H.M. Choset. *Principles of robot motion: theory, algorithms, and implementations*. MIT press, 2005.

[62] http://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping. [Accessed: 12-Dec-2014].

[63] P. Mountney, Stoyanov, D. Davison, A. Yang, G-Z. "Simultaneous Stereoscope Localization and Soft-Tissue Mapping for Minimal Invasive Surgery". *MICCAI* 1: 347–354, 2006.

[64] G. Dissanayake, P. Newman, S. Clark, H.F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localization and map building (SLAM) problem. *IEEE Transactions of Robotics and Automation*, 2001.

Appendix A

MATLAB Simulation: Robot Navigation Using the FastSLAM Approach with Two Obstacles Avoidance

A.1 Main Function

```
% Clear some data
clc
clf
clear all

% Setting the random seed, so the same example can be run
several times
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);

% Initial Robot Values
robot_pose=[10,-30,pi/2];

% Limit the initial angle to certain field
if (robot_pose(3)>pi)
    robot_pose(3)=robot_pose(3)-2*pi;
elseif (robot_pose(3)<=-pi)
    robot_pose(3)=robot_pose(3)+2*pi;
end

% Initial Robot Parameter
sensor_radius=10;
max_vel=4;
max_ang_vel=.5;

% Map Dimensions
map_max=30;
```

```
map_min=0;

% Goal Position on the map
goal=[10,40];
goal_radius=sensor_radius;

% Plot goal and the circle of interest around the goal
scatter(goal(1),goal(2),50,'d','m','filled');
rectangle('Position',[goal(1)-goal_radius, goal(2)-
goal_radius,...
        2*goal_radius, 2*goal_radius],
'Curvature',[1,1],'EdgeColor','m');

% Axis properties
axis([-50,60,-40,55]);
axis equal;
axis manual;
hold on;

% Give some notes to the goal and obstacles
%text(10,40,'\leftarrow sin(\pi)','FontSize',18)
text(11,40,'G','FontSize',18)
text(3.75,9,'O_1','FontSize',14,'FontWeight','bold')

text(15,10,'O_2','FontSize',14,'FontWeight','bold')

% Concave obstacle 1 dimensions
xlimit=[3,6,6,3];
ylimit=[8,8,11,11];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);
fill(concaveX1, concaveY1,'y','edgealpha',0);
alpha(.5); % Set the transparency

% Concave obstacle 2 dimensions
xlimit=[11,21,21,11];
ylimit=[5,5,15,15];
concaveX2=xlimit([4 1 2 3 4]);
concaveY2=ylimit([4 1 2 3 4]);
fill(concaveX2, concaveY2,'y','edgealpha',0);
alpha(.5);

% Set two function handles
sensorcircle=sensor_view(robot_pose,sensor_radius);
robotshape=moving_arrows(robot_pose);

% Some unceratinty parameters
```

```
measurementNoiseStdev = 0.1; speedStdev = 1;

% Some parameters for plotting the particles
m = 1000; k = 0.01;

% Number of particles
N = 500;

% Initialize particle weights
w = 1/N*ones(N,1);
w_goal = 1/N*ones(N,1);

% Create normally distribution particles around the know
initial
% position of the robot
p0=zeros(N,2);
p0(:,1)=robot_pose(1)*ones(N,1)+randn(N,1);
p0(:,2)=robot_pose(2)*ones(N,1)+randn(N,1);

% Time settings
t_sim=20;
T=0.1;
t_steps=floor(t_sim/T);

% Initial parameters
j=0;px=0;py=0;

% Output some values to display
plot_robot_position=[robot_pose(1)*ones(231,1),robot_pose(2)*o
nes(231,1)];
plot_estimation_obstacles=ones(231,2);
plot_estimation_goal=ones(231,2);

% Initial parameters
px_goal=0;py_goal=0;z=0;

% Main loop
for t=1:230

    % Choose turning direction if the obstacles enter into
safety radius
    [turning_direction]=turningdirection(robot_pose,
sensor_radius);
    e=turning_direction(1);

    % Plot the concave or convex obstacle, and check if search
sensor
```

```
    % intersect with obstacles. If yes, b or d >0, then the
particle filter
    % work afterwards. However, the coordinates of the
reference points for
    % particle filter are not obtained on this function. This
function also
    % gets the obst_dist and obst_angle for obstacle avoidance
algorithm.
    [concave_convex]=concave(robot_pose, sensor_radius);
    obst_dist=concave_convex(1);
    obst_angle=concave_convex(2);
    dim1=concave_convex(3);
    dim2=concave_convex(4);

    % Get the coordinates of intersection points for particle
filters
    [intersection_points]=getintersectionpoints(robot_pose,
sensor_radius);
    c=intersection_points(1); % distance measurements for PF
    PF_angle=intersection_points(2); % angle measurements for
PF
    x1=intersection_points(3);
    y1=intersection_points(4);

    % Calculate the new robot pose based on potential velocity
field obstacle avoidance algorithm
    robot=[robot_pose(1), robot_pose(2), robot_pose(3)];

    % Obtain distance and angle wrt the goal
    d=sqrt((robot_pose(1)-goal(1))^2+(robot_pose(2)-
goal(2))^2);
    PF_angle_goal=atan((goal(2)-robot_pose(2))/(goal(1)-
robot_pose(1)));

    % Plot updated robot pose
    delete(robotshape);
    delete(sensorcircle);
    sensorcircle=sensor_view(robot_pose,sensor_radius);
    robotshape=moving_arrows(robot_pose);

    % Based on which time step we want, choose the imbedding
position of ploting
    % robot poses

    [new_pose]= obstacle_controller(robot, goal, max_vel,
max_ang_vel,...
    goal_radius, sensor_radius, obst_dist, obst_angle, T, e);
```

```
    robot_pose=[new_pose(1),new_pose(2),new_pose(3)];
    delta_d=[new_pose(4),new_pose(5)];

    % Output some values
    plot_robot_position(t+1,1)=robot_pose(1);
    plot_robot_position(t+1,2)=robot_pose(2);

    zzzz=zeros(t,2);
    for i=1:t
        zzzz(i,1)=plot_robot_position(i,1);
        zzzz(i,2)=plot_robot_position(i,2);
    end

    % Plot estimated robot positions
    plot(zzzz(:,1),zzzz(:,2),'-r^','MarkerFaceColor','r');

    % Choose estimations wrt obstacles or the goal
    if (d>30)
      if (dim1>0)||(dim2>0)

        %%%%%%%%%%%%%%%% Particle Filter Robot Path
Estimation %%%%%%%%%%%%%%%%

        % Output particle clouds after time propagation
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);

        % Generate bearing and distance measurements (with
gaussian measurement noise)

measurebearing=PF_angle+randn*measurementNoiseStdev/30;
        measuredistance=c+randn*measurementNoiseStdev;

        % Calculate corresponding parameters for all particles
        predBrg = TrueBearing(p0,[x1*ones(N,1),y1*ones(N,1)]);
        predBrg1 =
TrueDistance(p0,[x1*ones(N,1),y1*ones(N,1)]);

        % Evaluate measurements (i.e., create weights) using
the pdf for the normal distribution
        brgWts =
w.*(1/(sqrt(2*pi)*measurementNoiseStdev/30)*exp(-(predBrg-
measurebearing).^2/(2*(measurementNoiseStdev/30)^2)));
```

```
        rngWts =
w.*(1/(sqrt(2*pi)*measurementNoiseStdev)*exp(-(predBrg1-
measuredistance).^2/(2*measurementNoiseStdev^2)));

        % Combine range and bearing weights
        f_Wts=brgWts.* rngWts;

        % Normalize particle weigths and plot particles
        w = f_Wts/sum(f_Wts);

        % Output particle clouds after measurement update
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);

        % Resample the particles
        u = rand(N,1); wc = cumsum(w);
        [~,ind1] = sort([u;wc]); ind=find(ind1<=N)-(0:N-1)';
        p0=p0(ind,:); w=ones(N,1)./N;

        % Plot the particles after resampling (may have many
overlapping particles)
        %scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');

        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);
        %}

        % Get the mean of particles
        for i=1:N
            px=p0(i,1)+px;
        end
        px=px/N;

        for i=1:N
            py=p0(i,2)+py;
        end
        py=py/N;

        % Plot estimated robot positions

plot(px,py,':bs','MarkerFaceColor','b','MarkerSize',10);
```

```
        plot_estimation_obstacles(t,1)=px;
        plot_estimation_obstacles(t,2)=py;

         % Time propagation
        speedNoisex=speedStdev*rand(N,1)*1;
        speedNoisey=speedStdev*randn(N,1);
        p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
        p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;



        %%%%%%%%%%%%%%%%%%%%%%%% Landmark Location
Estimation %%%%%%%%%%%

        % Get estimated landmark positions
        px1=px+c*cos(PF_angle);
        py1=py+c*sin(PF_angle);

        % Plot estimated landmark positions and virtual
measurement lines
        plot([px,px1],[py,py1],'--g');

plot(px1,py1,'b*','MarkerFaceColor','b','MarkerSize',6);

      else

        % Time propagation
        speedNoisex=speedStdev*randn(N,1);
        speedNoisey=speedStdev*randn(N,1);
        p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
        p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;
      end

    else

    %%%%%%%%%%%%%%%%%%%%%%%% Estimation wrt the
goal %%%%%%%%%%%%%%%%%

        %%%%%%%%%%%%%%%%% Particle Filtering wrt the
goal %%%%%%%%%%%%%%%%%

        % Output particle clouds after time propagation
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        %pause(.5)
```

```
        delete(particleHandle);

        % Generate bearing and distance measurements wrt the
goal (with gaussian measurement noise)

measurebearing_goal=PF_angle_goal+randn*measurementNoiseStdev/
10;
        measuredistance_goal=d+randn*measurementNoiseStdev;

        % Calculate corresponding parameters for all particles
        predBrg_goal =
TrueBearing(p0,[goal(1)*ones(N,1),goal(2)*ones(N,1)]);
        predBrg1_goal =
TrueDistance(p0,[goal(1)*ones(N,1),goal(2)*ones(N,1)]);

        % Evaluate measurements (i.e., create weights) using
the pdf for the normal distribution
        brgWts_goal =
w_goal.*(1/(sqrt(2*pi)*measurementNoiseStdev/10)*exp(-
(predBrg_goal-
measurebearing_goal).^2/(2*(measurementNoiseStdev/10)^2)));
        rngWts_goal =
w_goal.*(1/(sqrt(2*pi)*measurementNoiseStdev)*exp(-
(predBrg1_goal-
measuredistance_goal).^2/(2*measurementNoiseStdev^2)));

        % Combine range and bearing weights
        f_Wts_goal=brgWts_goal.* rngWts_goal;

        % Normalize particle weigths and plot particles
        w_goal = f_Wts_goal/sum(f_Wts_goal);

        % Output particle clouds after measurement update
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);

        %scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');

        % Resample the particles
        u_goal = rand(N,1); wc_goal = cumsum(w_goal);
        [~,ind1] = sort([u_goal;wc_goal]); ind=find(ind1<=N)-
(0:N-1)';
        p0=p0(ind,:); w_goal=ones(N,1)./N;
```

```
        % Output particle clouds after resampling
        %disp('particles after resampling...press return to
step....');t
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);

        % Get the mean of particles
        for i=1:N
            px_goal=p0(i,1)+px_goal;
        end
        px_goal=px_goal/N;

        for i=1:N
            py_goal=p0(i,2)+py_goal;
        end
        py_goal=py_goal/N;

        % Plot the estimated robot positions wrt the goal
        plot(px_goal,py_goal,'--
bp','MarkerFaceColor','b','MarkerSize',10);

        plot_estimation_goal(t+1,1)=px_goal;
        plot_estimation_goal(t+1,2)=py_goal;

        % Time propagation
        speedNoisex=speedStdev*randn(N,1);
        speedNoisey=speedStdev*randn(N,1);
        p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
        p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;

        %%%%%%%%%%%%%%%% Plot the measurements wrt the
goal %%%%%%%%%%%%

        % Plot the measurement line wrt the goal
        plot([px_goal,goal(1)],[py_goal,goal(2)],'--
g*','MarkerEdgeColor','b','MarkerSize',2);

    end

    % Reset some parameters
    px=0;
    py=0;
    px_goal=0;py_goal=0;
```

```
    % Save pictures
    if (mod(t,2)==0)
      nomeaning=strcat('pics/',int2str(t),'.png');
        saveas(gcf,nomeaning)
    end

    % Capture frames
    j=j+1;
    M(j)=getframe();

end
```

A.2 Choosing Turning Direction Function

```
function [turning_direction]=turningdirection(robot_pose,
sensor_radius)

% Obstacle Parameters
obst_radius=sensor_radius;
searchradius=30;

% Concave obstacle 1 (smaller) dimensions
xlimit=[3,6,6,3];
ylimit=[8,8,11,11];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);

% Concave obstacle 2 (bigger) dimensions
xlimit=[11,21,21,11];
ylimit=[5,5,15,15];
concaveX2=xlimit([4 1 2 3 4]); % close the shape
concaveY2=ylimit([4 1 2 3 4]);

% Check to see if the robot sensors intersect with an obstacle
dim1_r=0;dim2_r=0;
dim1_l=0;dim2_l=0;

%%%%%%%% Consider the right half part of the robot searching
range
for beam_angle=(robot_pose(3)-7*pi/12):pi/24:robot_pose(3);

    % Define sensing beams

x_ob=[robot_pose(1),robot_pose(1)+searchradius*cos(beam_angle)
];
```

```
y_ob=[robot_pose(2),robot_pose(2)+searchradius*sin(beam_angle)
];

    % Check intersections with obstacle 1
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX1, concaveY1);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for i=1:length(x1)
        dist1(i,:)= sqrt((x1(i)-robot_pose(1)).^2+(y1(i)-
robot_pose(2)).^2);
        dim1_r=dim1_r+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
      %scatter(x1,y1,50,'r','filled');
    end

    % Check intersections with obstacle 2
    [x2,y2]=polyxpoly(x_ob, y_ob, concaveX2, concaveY2);

    % If there are two intersections, caculate all the
distances
    dist2=zeros(length(x2),1);
    if (isempty(dist2)==false)
      for i=1:length(x2)
        dist2(i,:)= sqrt((x2(i)-robot_pose(1)).^2+(y2(i)-
robot_pose(2)).^2);
        dim2_r=dim2_r+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist2);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x2=x2(ind,:);
      y2=y2(ind,:);
```

```
      %scatter(x2,y2,50,'r','filled');
    end
end

%%%%%%%% Consider the left half part of the robot searching
range
for beam_angle=robot_pose(3):pi/24:(robot_pose(3)+7*pi/12);


x_ob=[robot_pose(1),robot_pose(1)+searchradius*cos(beam_angle)
];

y_ob=[robot_pose(2),robot_pose(2)+searchradius*sin(beam_angle)
];

    % Check intersections with obstacle 1
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX1, concaveY1);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for i=1:length(x1)
        dist1(i,:)= sqrt((x1(i)-robot_pose(1)).^2+(y1(i)-
robot_pose(2)).^2);
        dim1_l=dim1_l+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
      %scatter(x1,y1,50,'k','filled');
    end

    % Check intersections with obstacle 2
    [x2,y2]=polyxpoly(x_ob, y_ob, concaveX2, concaveY2);

    % If there are two intersections, caculate all the
distances
    dist2=zeros(length(x2),1);
    if (isempty(dist2)==false)
      for i=1:length(x2)
```

```
        dist2(i,:)= sqrt((x2(i)-robot_pose(1)).^2+(y2(i)-
robot_pose(2)).^2);
        dim2_l=dim2_l+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist2);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x2=x2(ind,:);
      y2=y2(ind,:);
      %scatter(x2,y2,50,'k','filled');
    end
end

%%%%%%%% Get the right part dimensions of the robot
dim_r=dim1_r+dim2_r;

%%%%%%%% Get the left part dimensions of the robot
dim_l=dim1_l+dim2_l;

if (dim_r>=dim_l)
   e=2; % robot turns left
else
   e=1; % robot turns right
end

% Return turning directions
turning_direction=e;
```

A.3 Robot Sensing Obstacles Function

```
function [concave_convex]=concave(robot_pose, sensor_radius)

% Obstacle Parameters
obst_radius=sensor_radius;
searchradius=30;

% Concave obstacle 1 (smaller) dimensions
xlimit=[3,6,6,3];
ylimit=[8,8,11,11];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);
%fill(concaveX1, concaveY1,'y','edgealpha',0);

% Concave obstacle 2 (bigger) dimensions
xlimit=[11,21,21,11];
ylimit=[5,5,15,15];
concaveX2=xlimit([4 1 2 3 4]); % close the shape
concaveY2=ylimit([4 1 2 3 4]);
%fill(concaveX2, concaveY2,'y','edgealpha',0);

% Check to see if the robot sensors intersect with an obstacle
dim1=0;dim2=0;
obst_dist=0;
obst_angle=0;

for beam_angle=(robot_pose(3)-
7*pi/12):pi/24:(robot_pose(3)+7*pi/12);


x_ob=[robot_pose(1),robot_pose(1)+searchradius*cos(beam_angle)
];

y_ob=[robot_pose(2),robot_pose(2)+searchradius*sin(beam_angle)
];

    % Check intersections with obstacle 1
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX1, concaveY1);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for i=1:length(x1)
        dist1(i,:)= sqrt((x1(i)-robot_pose(1)).^2+(y1(i)-
robot_pose(2)).^2);
```

```
        dim1=dim1+1; % calculate all intersection points in
the searchsensor range
      end
      dist1_min=min(dist1);
    else
      dist1_min=100;
    end

    % Check intersections with obstacle 2
    [x2,y2]=polyxpoly(x_ob, y_ob, concaveX2, concaveY2);

    % If there are two intersections, caculate all the
distances
    dist2=zeros(length(x2),1);
    if (isempty(dist2)==false)
      for i=1:length(x2)
        dist2(i,:)= sqrt((x2(i)-robot_pose(1)).^2+(y2(i)-
robot_pose(2)).^2);
        dim2=dim2+1; % calculate all intersection points in
the searchsensor range
      end
      dist2_min=min(dist2);
    else
      dist2_min=100;
    end

    if (isempty(dist1)==false)||(isempty(dist2)==false)
      % Only obstacles enters into the safety radius, robot
turns
       if (dist1_min<=obst_radius)
            obst_dist=dist1_min;
            obst_angle=beam_angle;
            break
       elseif (dist2_min<=obst_radius)
            obst_dist=dist2_min;
            obst_angle=beam_angle;
            break
       end
    else
          obst_dist=0;
    end
end

% Return obst_dist and obst_angle values to the main
concave_convex=[obst_dist,obst_angle,dim1,dim2];
```

```
% It gives the coordinate of the most counterclockwise
intersection point.
% Only one obst_dist and one obst_angle.
```

A.4 Obtaining Reference Point for Particle Filter and Real Measurement Values Function

```
function
[intersection_points]=getintersectionpoints(robot_pose,
sensor_radius)

% Obstacle Parameters
obst_radius=sensor_radius;
searchradius=30;

% Concave obstacle 1 (smaller) dimensions
xlimit=[3,6,6,3];
ylimit=[8,8,11,11];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);

% Concave obstacle 2 (bigger) dimensions
xlimit=[11,21,21,11];
ylimit=[5,5,15,15];
concaveX2=xlimit([4 1 2 3 4]); % close the shape
concaveY2=ylimit([4 1 2 3 4]);

% Check to see if the robot sensors intersect with an obstacle

PF_angle=0;
for beam_angle=(robot_pose(3)-
7*pi/12):pi/24:(robot_pose(3)+7*pi/12);


x_ob=[robot_pose(1),robot_pose(1)+searchradius*cos(beam_angle)
];

y_ob=[robot_pose(2),robot_pose(2)+searchradius*sin(beam_angle)
];

    %%%%% Check intersections with obstacle 1
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX1, concaveY1);

    % If there are two intersections, caculate all the
distances
```

```
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for j=1:length(x1)
        dist1(j,:)= sqrt((x1(j)-robot_pose(1)).^2+(y1(j)-
robot_pose(2)).^2);
      end
      % Obtain PF distance
      c=min(dist1);

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);

      if (length(x1)>1)
         x1=x1(1);
         y1=y1(1);
      end
      % Obtain PF angle
      PF_angle=beam_angle;
      break
    else
        c=100;
        x1=0;
        y1=0;
    end

    %%%%% Check intersections with obstacle 2
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX2, concaveY2);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for j=1:length(x1)
        dist1(j,:)= sqrt((x1(j)-robot_pose(1)).^2+(y1(j)-
robot_pose(2)).^2);
      end
      % Obtain PF distance
      c=min(dist1);

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
```

```
        x1=x1(ind,:);
        y1=y1(ind,:);

        if (length(x1)>1)
           x1=x1(1);
           y1=y1(1);
        end
        PF_angle=beam_angle;
        break
    else
        c=100;
        x1=0;
        y1=0;
    end
end

% Return obst_dist and obst_angle values to the main
intersection_points=[c,PF_angle,x1,y1];
```

A.5 Obstacle Avoidance Controller

```
function [new_position]= obstacle_controller(robot, goal,
max_vel, max_ang_vel,...
          goal_radius, obst_radius, obst_dist, obst_angle, T,
e)

% Constants
k_a=2;
k_n=0.5;
k_t=0.5;

% Calculate the x and y components of the distance to goal
delta_x=goal(1)-robot(1);
delta_y=goal(2)-robot(2);

% Calculate the distance to the goal
dist_goal=sqrt(delta_x^2+delta_y^2);

% Calculate angle to goal
theta=atan(delta_y/delta_x);

% Calculate the change in the angle between the current and
goal positon
delta_theta=theta-robot(3);

% Calculate the distance relation
dist_relation=exp(-dist_goal/goal_radius);

% Calculate attractive velocity
u_a=k_a*max_vel*(1-dist_relation);

% Find new theta value
close_to_zero=0.02;
if(theta<=0 && (pi+theta-robot(3))< close_to_zero)
    theta=pi+theta;
elseif (theta>0 && abs(delta_theta)< close_to_zero)
    theta=theta;
elseif (theta<=0)
    theta=robot(3)+max_ang_vel*T;
elseif (theta>0)
    theta=robot(3)-max_ang_vel*T;
end

% Calculate repulsive velocity
% If no obstacle is in view
% set normal and tangent values to zero
```

```
% If an obstacle is in view
% calculate normal and tangent values

if (obst_dist==0)
    normal=0;
    tangent=0;
    obst_relation=0;
    normal_angle=0;
    tangent_angle=0;
    theta_obst=0;

else
    normal=1/obst_dist;

    tangent=1/obst_dist;

    obst_relation=exp(-obst_dist/obst_radius);

    % Calculate normal angle
    normal_angle=obst_angle+pi;

    if (e==1)
        theta_obst=robot(3)-max_ang_vel*T;
        tangent_angle=normal_angle+pi/2;

    elseif (e==2)
        theta_obst=robot(3)+max_ang_vel*T;
        tangent_angle=normal_angle-pi/2;
    end

    % Ignore goal while obstacle is in view
    u_a=0;
    theta=0;
end

% Calculate normal and tangential velocity components
u_n=k_n*normal*obst_relation;
u_t=k_t*tangent*obst_relation;

% Calculate the sum of all velocity components
u_totX=u_a*cos(theta)+u_n*cos(normal_angle)+u_t*cos(tangent_an
gle);
u_totY=u_a*sin(theta)+u_n*sin(normal_angle)+u_t*sin(tangent_an
gle);

% Calculate the change in x and y distance for the current
time step
```

```
delta_d=[u_totX*T, u_totY*T];
delta_theta=theta+theta_obst;

% Limit the angle to a certain field
if (delta_theta>=2*pi)
    delta_theta=delta_theta-2*pi;
elseif (delta_theta<=-2*pi)
    delta_theta=delta_theta+2*pi;
end

% Send the robot's next position to the main function
new_position=[robot(1)+delta_d(1), robot(2)+delta_d(2),...
    delta_theta, delta_d(1), delta_d(2)];
```

A.6 Obtaining Particles Bearings Function

```
%Returns brg in radians from P2 (vector of x and y)
%to P1 (single x y element)

function [brg] = TrueBearing(x,y)
brg = atan2(y(:,2)-x(:,2),y(:,1)-x(:,1));
```

A.7 Obtaining Particles Distances Function

```
function [brg1] = TrueDistance(x,y)
brg1=zeros(length(x),1);
%N=length(x);
for i=1:length(x)
  brg1(i,:)=sqrt((y(i,1)-x(i,1)).^2+(y(i,2)-x(i,2)).^2);
end
```

A.8 Moving Arrow Function

```
function [h]= moving_arrows(Q)
x=Q(1);
y=Q(2);
theta=Q(3);

l = 0.1;
X = [x,x+l*cos(theta-2*pi/3) ,
x+l*cos(theta),x+l*cos(theta+2*pi/3),x];
Y = [y,y+l*sin(theta-2*pi/3) ,
y+l*sin(theta),y+l*sin(theta+2*pi/3),y];
h= fill(X,Y,'r');
```

A.9 Sensor Configuration Function

```
function [h]=sensor_view(Q,radius)
x=Q(1);
y=Q(2);
theta=Q(3);
searchradius=30;
% Plot beam
i=0;
beamX=zeros(29,2);
beamY=zeros(29,2);

% Plot beam
for beam_angle=(theta-7*pi/12):pi/24:(theta+7*pi/12);
    i=i+1;
    beamX(i,:)=[x,x+searchradius*cos(beam_angle)];
    beamY(i,:)=[y,y+searchradius*sin(beam_angle)];

end
%b=[beamX,beamY]
h1=plot(beamX(1,:),beamY(1,:),'c',beamX(2,:),beamY(2,:),'c',be
amX(3,:),beamY(3,:),'c',beamX(4,:),beamY(4,:),...

'c',beamX(5,:),beamY(5,:),'c',beamX(6,:),beamY(6,:),'c',beamX(
7,:),beamY(7,:),'c',beamX(8,:),beamY(8,:),...

'c',beamX(9,:),beamY(9,:),'c',beamX(10,:),beamY(10,:),'c',beam
X(11,:),beamY(11,:),'c',...

beamX(12,:),beamY(12,:),'c',beamX(13,:),beamY(13,:),'c',beamX(
14,:),beamY(14,:),'c'...
    ,beamX(15,:),beamY(15,:),'c',beamX(16,:),beamY(16,:),'c',b
eamX(17,:),beamY(17,:),'c'...
    ,beamX(18,:),beamY(18,:),'c',beamX(19,:),beamY(19,:),'c',b
eamX(20,:),beamY(20,:),'c'...
    ,beamX(21,:),beamY(21,:),'c',beamX(22,:),beamY(22,:),'c',b
eamX(23,:),beamY(23,:),'c'...
    ,beamX(24,:),beamY(24,:),'c',beamX(25,:),beamY(25,:),'c',b
eamX(26,:),beamY(26,:),'c'...
    ,beamX(27,:),beamY(27,:),'c',beamX(28,:),beamY(28,:),'c',b
eamX(29,:),beamY(29,:),'c');


% Plot searching range
circle=rectangle('Position',[x-searchradius,y-
searchradius,2*searchradius,2*searchradius],...
```

```
'Curvature',[1,1],'EdgeColor','b','LineStyle','-.');

% Plot search radius
circle1=rectangle('Position',[x-radius,y-
radius,2*radius,2*radius],...
'Curvature',[1,1],'EdgeColor','m');

h=[h1;circle;circle1];
```

Appendix B

MATLAB Simulation: Robot Navigation Using the FastSLAM Approach Escaping a Concave Obstacle

B.1 Main Function

```
clc
clf
clear all

% Setting the random seed, so the same example can be run
several times
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);

% Initial Robot Values
robot_pose=[10,-45,pi/2];

if (robot_pose(3)>pi)
    robot_pose(3)=robot_pose(3)-2*pi;
elseif (robot_pose(3)<=-pi)
    robot_pose(3)=robot_pose(3)+2*pi;
end

sensor_radius=10;
max_vel=4;
max_ang_vel=.5;

% Map Dimensions
map_max=30;
map_min=0;

% Goal Position on the map
```

```
goal=[10,35];
goal_radius=sensor_radius;

% Plot goal and the circle of interest around the goal
scatter(goal(1),goal(2),50,'d','m','filled');
rectangle('Position',[goal(1)-goal_radius, goal(2)-
goal_radius,...
        2*goal_radius, 2*goal_radius],
'Curvature',[1,1],'EdgeColor','m');

% Axis properties
axis([-60,60,-55,55]);
axis equal;
axis manual;
hold on;

text(11,40,'G','FontSize',14)
text(-1.45,4.55,'O_1','FontSize',10,'FontWeight','bold')
text(3.7,8.6,'O_2','FontSize',10,'FontWeight','bold')

text(8.65,9.5,'O_3','FontSize',10,'FontWeight','bold')
text(13.65,4.55,'O_4','FontSize',10,'FontWeight','bold')
text(18.5,-10.5,'O_5','FontSize',10,'FontWeight','bold')

% Concave obstacle 1 dimensions
xlimit=[-2,1,1,-2];
ylimit=[3,3,6,6];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);
fill(concaveX1, concaveY1,'y','edgealpha',0);
alpha(.5); % Set the transparency

% Concave obstacle 2 dimensions
xlimit=[3,6,6,3];
ylimit=[7,7,10,10];
concaveX2=xlimit([4 1 2 3 4]);
concaveY2=ylimit([4 1 2 3 4]);
fill(concaveX2, concaveY2,'y','edgealpha',0);
alpha(.5); % Set the transparency

% Concave obstacle 3 dimensions
xlimit=[8,11,11,8];
ylimit=[8,8,11,11];
concaveX3=xlimit([4 1 2 3 4]);
concaveY3=ylimit([4 1 2 3 4]);
fill(concaveX3, concaveY3,'y','edgealpha',0);
alpha(.5); % Set the transparency
```

```
% Concave obstacle 4 dimensions
xlimit=[13,16,16,13];
ylimit=[3,3,6,6];
concaveX4=xlimit([4 1 2 3 4]);
concaveY4=ylimit([4 1 2 3 4]);
fill(concaveX4, concaveY4,'y','edgealpha',0);
alpha(.5); % Set the transparency

% Concave obstacle 5 dimensions
xlimit=[18,21,21,18];
ylimit=[-12,-12,-9,-9];
concaveX5=xlimit([4 1 2 3 4]);
concaveY5=ylimit([4 1 2 3 4]);
fill(concaveX5, concaveY5,'y','edgealpha',0);
alpha(.5); % Set the transparency

% Set two function handles
sensorcircle=sensor_view(robot_pose,sensor_radius);
robotshape=moving_arrows(robot_pose);

% Some unceratinty parameters
measurementNoiseStdev = 0.1; speedStdev = 1;

% Some parameters for plotting the particles
m = 1000; k = 0.01;

% Number of particles
N = 500;

% Initialize particle weights
w = 1/N*ones(N,1);
w_goal = 1/N*ones(N,1);

% Creat normally distribution particles around the know
initial
% position of the robot
p0=zeros(N,2);
p0(:,1)=robot_pose(1)*ones(N,1)+randn(N,1);
p0(:,2)=robot_pose(2)*ones(N,1)+randn(N,1);

% Time settings
t_sim=20;
T=0.1;
t_steps=floor(t_sim/T);
```

```
plot_robot_position=[robot_pose(1)*ones(301,1),robot_pose(2)*o
nes(301,1)];
plot_estimation_obstacles=ones(301,2);
plot_estimation_goal=ones(301,2);

% Movie
j=0;px=0;py=0;p=zeros(300,2);
for t=1:300

    % Choose turning direction (if the obstacles enter into
safety radius)
    [turning_direction]=turningdirection(robot_pose,
sensor_radius);
    e=turning_direction(1);

    % Plot the concave or convex obstacle, and check if search
sensor
    % intersect with obstacles. If yes, dim1,dim2,dim3 or
dim4 >0, then the particle filter
    % work afterwards. However, the coordinates of the
reference points for
    % particle filter are not obtained on this function. This
function also
    % gets the obst_dist and obst_angle for obstacle avoidance
algorithm.

    [concave_convex]=concave(robot_pose, sensor_radius);
    obst_dist=concave_convex(1);
    obst_angle=concave_convex(2);
    dim1=concave_convex(3);
    dim2=concave_convex(4);
    dim3=concave_convex(5);
    dim4=concave_convex(6);
    dim5=concave_convex(7);

    % Get the coordinates of intersection points for particle
filters
    [intersection_points]=getintersectionpoints(robot_pose,
sensor_radius);
    c=intersection_points(1); % distance measurements for PF
    PF_angle=intersection_points(2);
    x1=intersection_points(3);
    y1=intersection_points(4);

    % Calculate the new robot pose based on potential velocity
field obstacle avoidance algorithm
    robot=[robot_pose(1), robot_pose(2), robot_pose(3)];
```

```
    d=sqrt((robot_pose(1)-goal(1))^2+(robot_pose(2)-
goal(2))^2);
    PF_angle_goal=atan((goal(2)-robot_pose(2))/(goal(1)-
robot_pose(1)));

    % Plot updated robot pose
    delete(robotshape);
    delete(sensorcircle);
    sensorcircle=sensor_view(robot_pose,sensor_radius);
    robotshape=moving_arrows(robot_pose);


    % Based on which time step we want, choose the imbedding
position of ploting
    % robot poses

    [new_pose]= obstacle_controller(robot, goal, max_vel,
max_ang_vel,...
    goal_radius, sensor_radius, obst_dist, obst_angle, T, e);
    robot_pose=[new_pose(1),new_pose(2),new_pose(3)];
    delta_d=[new_pose(4),new_pose(5)];

    %plot(robot_pose(1),robot_pose(2),'r.');

    plot_robot_position(t+1,1)=robot_pose(1);
    plot_robot_position(t+1,2)=robot_pose(2);

    zzzz=zeros(t,2);
    for i=1:t
        zzzz(i,1)=plot_robot_position(i,1);
        zzzz(i,2)=plot_robot_position(i,2);
    end
    plot(zzzz(:,1),zzzz(:,2),'-r^','MarkerFaceColor','r');

    if (d>30)
    if (dim1>0)||(dim2>0)||(dim3>0)||(dim4>0)||(dim5>0)

        %%%%%%%% Filters begin %%%%%%%%%%

        %%%%%%%%%%%%%%%% Particle Filter Robot Path
Estimation %%%%%%%%%%%%%%%%

        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
        pause;
        %pause(.5)
```

```
        delete(particleHandle);

        % Generate bearing and distance measurements (with
gaussian measurement noise)

measurebearing=PF_angle+randn*measurementNoiseStdev/30;
        measuredistance=c+randn*measurementNoiseStdev;

        % Calculate corresponding parameters for all particles
        predBrg = TrueBearing(p0,[x1*ones(N,1),y1*ones(N,1)]);
        predBrg1 =
TrueDistance(p0,[x1*ones(N,1),y1*ones(N,1)]);

        % Evaluate measurements (i.e., create weights) using
the pdf for the normal distribution
        brgWts =
w.*(1/(sqrt(2*pi)*measurementNoiseStdev/30)*exp(-(predBrg-
measurebearing).^2/(2*(measurementNoiseStdev/30)^2)));
        rngWts =
w.*(1/(sqrt(2*pi)*measurementNoiseStdev)*exp(-(predBrg1-
measuredistance).^2/(2*measurementNoiseStdev^2)));

        % Combine range and bearing weights
        f_Wts=brgWts.* rngWts;

        % Normalize particle weigths and plot particles
        w = f_Wts/sum(f_Wts);

        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);

        %scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');

        % Resample the particles
        u = rand(N,1); wc = cumsum(w);
        [~,ind1] = sort([u;wc]); ind=find(ind1<=N)-(0:N-1)';
        p0=p0(ind,:); w=ones(N,1)./N;

        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);
```

```
        % Get the mean
        for i=1:N
            px=p0(i,1)+px;
        end
        px=px/N;

        for i=1:N
            py=p0(i,2)+py;
        end
        py=py/N;


plot(px,py,':bs','MarkerFaceColor','b','MarkerSize',6);
        plot_estimation_obstacles(t,1)=px;
        plot_estimation_obstacles(t,2)=py;

        % Time propagation
        speedNoisex=speedStdev*abs(randn(N,1))/5;
        speedNoisey=speedStdev*randn(N,1);
        p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
        p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;

        %%%%%%%%%%%%%%%%%%%%%%%% Landmark Location
Estimation %%%%%%%%%%%

        px1=px+c*cos(PF_angle);
        py1=py+c*sin(PF_angle);
        plot([px,px1],[py,py1],'--g');

plot(px1,py1,'b*','MarkerFaceColor','b','MarkerSize',6);
    else
        % Time propagation
        speedNoisex=speedStdev*randn(N,1);
        speedNoisey=speedStdev*randn(N,1);
        p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
        p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;
    end

    else

        %%%%%%%%%%%%%%%%%%%%%%%% Estimation wrt the
goal %%%%%%%%%%%%%%%%%

        %%%%%%%%%%%%%%%% Particle Filtering wrt the
goal %%%%%%%%%%%%%%%%%
```

```
        %disp('time propagation after resampling...press
return to step....');t
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);

        % Generate bearing and distance measurements wrt the
goal (with gaussian measurement noise)

measurebearing_goal=PF_angle_goal+randn*measurementNoiseStdev/
20;
        measuredistance_goal=d+randn*measurementNoiseStdev;

        % Calculate corresponding parameters for all particles
        predBrg_goal =
TrueBearing(p0,[goal(1)*ones(N,1),goal(2)*ones(N,1)]);
        predBrg1_goal =
TrueDistance(p0,[goal(1)*ones(N,1),goal(2)*ones(N,1)]);

        % Evaluate measurements (i.e., create weights) using
the pdf for the normal distribution
        brgWts_goal =
w_goal.*(1/(sqrt(2*pi)*measurementNoiseStdev/20)*exp(-
(predBrg_goal-
measurebearing_goal).^2/(2*(measurementNoiseStdev/20)^2)));
        rngWts_goal =
w_goal.*(1/(sqrt(2*pi)*measurementNoiseStdev)*exp(-
(predBrg1_goal-
measuredistance_goal).^2/(2*measurementNoiseStdev^2)));

        % Combine range and bearing weights
        f_Wts_goal=brgWts_goal.* rngWts_goal;

        % Normalize particle weigths and plot particles
        w_goal = f_Wts_goal/sum(f_Wts_goal);

        %disp('obtain weights for particles...press return to
step....');t
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);

        %scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
```

```
        % Resample the particles
        u_goal = rand(N,1); wc_goal = cumsum(w_goal);
        [~,ind1] = sort([u_goal;wc_goal]); ind=find(ind1<=N)-
(0:N-1)';
        p0=p0(ind,:); w_goal=ones(N,1)./N;

        %disp('particles after resampling...press return to
step....');t
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        %pause(.5)
        delete(particleHandle);

        for i=1:N
            px_goal=p0(i,1)+px_goal;
        end
        px_goal=px_goal/N;

        for i=1:N
            py_goal=p0(i,2)+py_goal;
        end
        py_goal=py_goal/N;

        plot(px_goal,py_goal,'--
bp','MarkerFaceColor','b','MarkerSize',10);
        plot_estimation_goal(t+1,1)=px_goal;
        plot_estimation_goal(t+1,2)=py_goal;

        % Time propagation
        speedNoisex=speedStdev*randn(N,1);
        speedNoisey=speedStdev*randn(N,1);
        p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
        p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;

        %%%%%%%%%%%%%%%% Plot the measurements wrt the
goal %%%%%%%%%%%%

        plot([px_goal,goal(1)],[py_goal,goal(2)],'--
g*','MarkerEdgeColor','b','MarkerSize',2);
    end

    px=0;
    py=0;
    px_goal=0;py_goal=0;
```

```
    % Save pictures
    if (mod(t,2)==0)
      nomeaning=strcat('pics/',int2str(t),'.png');
        saveas(gcf,nomeaning)
    end

    j=j+1;
    M(j)=getframe();

end
```

B.2 Choosing Turning Direction Function

```
function [turning_direction]=turningdirection(robot_pose,
sensor_radius)

% Obstacle Parameters
obst_radius=sensor_radius;
searchradius=30;

% Concave obstacle 1 dimensions
xlimit=[-2,1,1,-2];
ylimit=[3,3,6,6];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);

% Concave obstacle 2 dimensions
xlimit=[3,6,6,3];
ylimit=[7,7,10,10];
concaveX2=xlimit([4 1 2 3 4]);
concaveY2=ylimit([4 1 2 3 4]);

% Concave obstacle 3 dimensions
xlimit=[8,11,11,8];
ylimit=[8,8,11,11];
concaveX3=xlimit([4 1 2 3 4]);
concaveY3=ylimit([4 1 2 3 4]);

% Concave obstacle 4 dimensions
xlimit=[13,16,16,13];
ylimit=[3,3,6,6];
concaveX4=xlimit([4 1 2 3 4]);
concaveY4=ylimit([4 1 2 3 4]);

% Concave obstacle 5 dimensions
```

```
xlimit=[18,21,21,18];
ylimit=[-12,-12,-9,-9];
concaveX5=xlimit([4 1 2 3 4]);
concaveY5=ylimit([4 1 2 3 4]);

% Check to see if the robot sensors intersect with an obstacle
dim1_r=0;dim2_r=0;dim3_r=0;dim4_r=0;dim5_r=0;
dim1_l=0;dim2_l=0;dim3_l=0;dim4_l=0;dim5_l=0;

%%%%%%%% Consider the right half part of the robot searching
range
for beam_angle=(robot_pose(3)-7*pi/12):pi/24:robot_pose(3);


x_ob=[robot_pose(1),robot_pose(1)+searchradius*cos(beam_angle)
];

y_ob=[robot_pose(2),robot_pose(2)+searchradius*sin(beam_angle)
];

    % Check intersections with obstacle 1
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX1, concaveY1);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for i=1:length(x1)
        dist1(i,:)= sqrt((x1(i)-robot_pose(1)).^2+(y1(i)-
robot_pose(2)).^2);
        dim1_r=dim1_r+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
    end

    % Check intersections with obstacle 2
    [x2,y2]=polyxpoly(x_ob, y_ob, concaveX2, concaveY2);

    % If there are two intersections, caculate all the
distances
```

```
    dist2=zeros(length(x2),1);
    if (isempty(dist2)==false)
      for i=1:length(x2)
        dist2(i,:)= sqrt((x2(i)-robot_pose(1)).^2+(y2(i)-
robot_pose(2)).^2);
        dim2_r=dim2_r+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist2);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x2=x2(ind,:);
      y2=y2(ind,:);
    end

    % Check intersections with obstacle 3
    [x3,y3]=polyxpoly(x_ob, y_ob, concaveX3, concaveY3);

    % If there are two intersections, caculate all the
distances
    dist3=zeros(length(x3),1);
    if (isempty(dist3)==false)
      for i=1:length(x3)
        dist3(i,:)= sqrt((x3(i)-robot_pose(1)).^2+(y3(i)-
robot_pose(2)).^2);
        dim3_r=dim3_r+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist3);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x3=x3(ind,:);
      y3=y3(ind,:);
    end

    % Check intersections with obstacle 4
    [x4,y4]=polyxpoly(x_ob, y_ob, concaveX4, concaveY4);

    % If there are two intersections, caculate all the
distances
    dist4=zeros(length(x4),1);
    if (isempty(dist4)==false)
      for i=1:length(x4)
```

```
        dist4(i,:)= sqrt((x4(i)-robot_pose(1)).^2+(y4(i)-
robot_pose(2)).^2);
        dim4_r=dim4_r+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist4);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x4=x4(ind,:);
      y4=y4(ind,:);
    end

    % Check intersections with obstacle 5
    [x5,y5]=polyxpoly(x_ob, y_ob, concaveX5, concaveY5);

    % If there are two intersections, caculate all the
distances
    dist5=zeros(length(x5),1);
    if (isempty(dist5)==false)
      for i=1:length(x5)
        dist5(i,:)= sqrt((x5(i)-robot_pose(1)).^2+(y5(i)-
robot_pose(2)).^2);
        dim5_r=dim5_r+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist5);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x5=x5(ind,:);
      y5=y5(ind,:);
    end
end

%%%%%%%% Consider the left half part of the robot searching
range
for beam_angle=robot_pose(3):pi/24:(robot_pose(3)+7*pi/12);


x_ob=[robot_pose(1),robot_pose(1)+searchradius*cos(beam_angle)
];

y_ob=[robot_pose(2),robot_pose(2)+searchradius*sin(beam_angle)
];
```

```
    % Check intersections with obstacle 1
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX1, concaveY1);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for i=1:length(x1)
        dist1(i,:)= sqrt((x1(i)-robot_pose(1)).^2+(y1(i)-
robot_pose(2)).^2);
        dim1_l=dim1_l+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
    end

    % Check intersections with obstacle 2
    [x2,y2]=polyxpoly(x_ob, y_ob, concaveX2, concaveY2);

    % If there are two intersections, caculate all the
distances
    dist2=zeros(length(x2),1);
    if (isempty(dist2)==false)
      for i=1:length(x2)
        dist2(i,:)= sqrt((x2(i)-robot_pose(1)).^2+(y2(i)-
robot_pose(2)).^2);
        dim2_l=dim2_l+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist2);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x2=x2(ind,:);
      y2=y2(ind,:);
    end


    % Check intersections with obstacle 3
```

```
    [x3,y3]=polyxpoly(x_ob, y_ob, concaveX3, concaveY3);

    % If there are two intersections, caculate all the
distances
    dist3=zeros(length(x3),1);
    if (isempty(dist3)==false)
      for i=1:length(x3)
        dist3(i,:)= sqrt((x3(i)-robot_pose(1)).^2+(y3(i)-
robot_pose(2)).^2);
        dim3_l=dim3_l+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist3);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x3=x3(ind,:);
      y3=y3(ind,:);
    end

    % Check intersections with obstacle 4
    [x4,y4]=polyxpoly(x_ob, y_ob, concaveX4, concaveY4);

    % If there are two intersections, caculate all the
distances
    dist4=zeros(length(x4),1);
    if (isempty(dist4)==false)
      for i=1:length(x4)
        dist4(i,:)= sqrt((x4(i)-robot_pose(1)).^2+(y4(i)-
robot_pose(2)).^2);
        dim4_l=dim4_l+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist4);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x4=x4(ind,:);
      y4=y4(ind,:);
    end

    % Check intersections with obstacle 5
    [x5,y5]=polyxpoly(x_ob, y_ob, concaveX5, concaveY5);
```

```
    % If there are two intersections, caculate all the
distances
    dist5=zeros(length(x5),1);
    if (isempty(dist5)==false)
      for i=1:length(x5)
        dist5(i,:)= sqrt((x5(i)-robot_pose(1)).^2+(y5(i)-
robot_pose(2)).^2);
        dim5_l=dim5_l+1; % calculate all intersection points
in the searchsensor range
      end

      % Just point the closest one
      [~,ind] = sort(dist5);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x5=x5(ind,:);
      y5=y5(ind,:);
    end
end

%%%%%%%% Get the right part dimensions of the robot
dim_r=dim1_r+dim2_r+dim3_r+dim4_r+dim5_r;

%%%%%%%% Get the left part dimensions of the robot
dim_l=dim1_l+dim2_l+dim3_l+dim4_l+dim5_l;

if (dim_r>=dim_l)
   e=2; % robot turns left
else
   e=1; % robot turns right
end

% Return turning directions
turning_direction=e;
```

B.3 Robot Sensing Obstacles Function

```
function [concave_convex]=concave(robot_pose, sensor_radius)

% Obstacle Parameters
obst_radius=sensor_radius;
searchradius=30;

% Concave obstacle 1 dimensions
xlimit=[-2,1,1,-2];
ylimit=[3,3,6,6];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);

% Concave obstacle 2 dimensions
xlimit=[3,6,6,3];
ylimit=[7,7,10,10];
concaveX2=xlimit([4 1 2 3 4]);
concaveY2=ylimit([4 1 2 3 4]);

% Concave obstacle 3 dimensions
xlimit=[8,11,11,8];
ylimit=[8,8,11,11];
concaveX3=xlimit([4 1 2 3 4]);
concaveY3=ylimit([4 1 2 3 4]);

% Concave obstacle 4 dimensions
xlimit=[13,16,16,13];
ylimit=[3,3,6,6];
concaveX4=xlimit([4 1 2 3 4]);
concaveY4=ylimit([4 1 2 3 4]);

% Concave obstacle 5 dimensions
xlimit=[18,21,21,18];
ylimit=[-12,-12,-9,-9];
concaveX5=xlimit([4 1 2 3 4]);
concaveY5=ylimit([4 1 2 3 4]);

% Check to see if the robot sensors intersect with an obstacle
dim1=0;dim2=0;dim3=0;dim4=0;dim5=0;
obst_dist=0;
obst_angle=0;
for beam_angle=(robot_pose(3)-
7*pi/12):pi/24:(robot_pose(3)+7*pi/12);
```

```
x_ob=[robot_pose(1),robot_pose(1)+searchradius*cos(beam_angle)
];

y_ob=[robot_pose(2),robot_pose(2)+searchradius*sin(beam_angle)
];

    % Check intersections with obstacle 1
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX1, concaveY1);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for i=1:length(x1)
        dist1(i,:)= sqrt((x1(i)-robot_pose(1)).^2+(y1(i)-
robot_pose(2)).^2);
        dim1=dim1+1; % calculate all intersection points in
the searchsensor range
      end
      dist1_min=min(dist1);
    else
      dist1_min=100;
    end

    % Check intersections with obstacle 2
    [x2,y2]=polyxpoly(x_ob, y_ob, concaveX2, concaveY2);

    % If there are two intersections, caculate all the
distances
    dist2=zeros(length(x2),1);
    if (isempty(dist2)==false)
      for i=1:length(x2)
        dist2(i,:)= sqrt((x2(i)-robot_pose(1)).^2+(y2(i)-
robot_pose(2)).^2);
        dim2=dim2+1; % calculate all intersection points in
the searchsensor range
      end
      dist2_min=min(dist2);
    else
      dist2_min=100;
    end

    % Check intersections with obstacle 3
    [x3,y3]=polyxpoly(x_ob, y_ob, concaveX3, concaveY3);
```

```
    % If there are two intersections, caculate all the
distances
    dist3=zeros(length(x3),1);
    if (isempty(dist3)==false)
      for i=1:length(x3)
        dist3(i,:)= sqrt((x3(i)-robot_pose(1)).^2+(y3(i)-
robot_pose(2)).^2);
        dim3=dim3+1; % calculate all intersection points in
the searchsensor range
      end
      dist3_min=min(dist3);
    else
      dist3_min=100;
    end

    % Check intersections with obstacle 4
    [x4,y4]=polyxpoly(x_ob, y_ob, concaveX4, concaveY4);

    % If there are two intersections, caculate all the
distances
    dist4=zeros(length(x4),1);
    if (isempty(dist4)==false)
      for i=1:length(x4)
        dist4(i,:)= sqrt((x4(i)-robot_pose(1)).^2+(y4(i)-
robot_pose(2)).^2);
        dim4=dim4+1; % calculate all intersection points in
the searchsensor range
      end
      dist4_min=min(dist4);
    else
      dist4_min=100;
    end

    % Check intersections with obstacle 5
    [x5,y5]=polyxpoly(x_ob, y_ob, concaveX5, concaveY5);

    % If there are two intersections, caculate all the
distances
    dist5=zeros(length(x5),1);
    if (isempty(dist5)==false)
      for i=1:length(x5)
        dist5(i,:)= sqrt((x5(i)-robot_pose(1)).^2+(y5(i)-
robot_pose(2)).^2);
        dim5=dim5+1; % calculate all intersection points in
the searchsensor range
      end
      dist5_min=min(dist5);
```

```
    else
      dist5_min=100;
    end

    if
(isempty(dist1)==false)||(isempty(dist2)==false)||(isempty(dis
t3)==false)||(isempty(dist4)==false)||(isempty(dist5)==false)
      % Only obstacles enters into the safety radius, robot
turns
       if (dist1_min<=obst_radius)
            obst_dist=dist1_min;
            obst_angle=beam_angle;
            break
       elseif (dist2_min<=obst_radius)
            obst_dist=dist2_min;
            obst_angle=beam_angle;
            break
       elseif (dist3_min<=obst_radius)
            obst_dist=dist3_min;
            obst_angle=beam_angle;
            break
       elseif (dist4_min<=obst_radius)
            obst_dist=dist4_min;
            obst_angle=beam_angle;
            break
       elseif (dist5_min<=obst_radius)
            obst_dist=dist5_min;
            obst_angle=beam_angle;
            break
       end
    else
          obst_dist=0;
    end
end

% Return obst_dist and obst_angle values to the main
concave_convex=[obst_dist,obst_angle,dim1,dim2,dim3,dim4,dim5]
;

% It gives the coordinate of the most counterclockwise
intersection point.
% Only one obst_dist and one obst_angle.
```

B.4 Obtaining Reference Point for Particle Filter and Real Measurement Values Function

```
function
[intersection_points]=getintersectionpoints(robot_pose,
sensor_radius)

% Obstacle Parameters
obst_radius=sensor_radius;
searchradius=30;

% Concave obstacle 1 dimensions
xlimit=[-2,1,1,-2];
ylimit=[3,3,6,6];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);

% Concave obstacle 2 dimensions
xlimit=[3,6,6,3];
ylimit=[7,7,10,10];
concaveX2=xlimit([4 1 2 3 4]);
concaveY2=ylimit([4 1 2 3 4]);

% Concave obstacle 3 dimensions
xlimit=[8,11,11,8];
ylimit=[8,8,11,11];
concaveX3=xlimit([4 1 2 3 4]);
concaveY3=ylimit([4 1 2 3 4]);

% Concave obstacle 4 dimensions
xlimit=[13,16,16,13];
ylimit=[3,3,6,6];
concaveX4=xlimit([4 1 2 3 4]);
concaveY4=ylimit([4 1 2 3 4]);

% Concave obstacle 5 dimensions
xlimit=[18,21,21,18];
ylimit=[-12,-12,-9,-9];
concaveX5=xlimit([4 1 2 3 4]);
concaveY5=ylimit([4 1 2 3 4]);

% Check to see if the robot sensors intersect with an obstacle

PF_angle=0;
```

```
for beam_angle=(robot_pose(3)-
7*pi/12):pi/24:(robot_pose(3)+7*pi/12);


x_ob=[robot_pose(1),robot_pose(1)+searchradius*cos(beam_angle)
];

y_ob=[robot_pose(2),robot_pose(2)+searchradius*sin(beam_angle)
];

    %%%%% Check intersections with obstacle 1
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX1, concaveY1);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for j=1:length(x1)
        dist1(j,:)= sqrt((x1(j)-robot_pose(1)).^2+(y1(j)-
robot_pose(2)).^2);
      end
      c=min(dist1);

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
      if (length(x1)>1)
         x1=x1(1);
         y1=y1(1);
      end
      PF_angle=beam_angle;
      break
    else
        c=100;
        x1=0;
        y1=0;
    end

    %%%%% Check intersections with obstacle 2
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX2, concaveY2);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
```

```
    if (isempty(dist1)==false)
      for j=1:length(x1)
        dist1(j,:)= sqrt((x1(j)-robot_pose(1)).^2+(y1(j)-
robot_pose(2)).^2);
      end
      c=min(dist1);

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
      if (length(x1)>1)
         x1=x1(1);
         y1=y1(1);
      end
      PF_angle=beam_angle;
      break
    else
        c=100;
        x1=0;
        y1=0;
    end

    %%%%% Check intersections with obstacle 3
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX3, concaveY3);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for j=1:length(x1)
        dist1(j,:)= sqrt((x1(j)-robot_pose(1)).^2+(y1(j)-
robot_pose(2)).^2);
      end
      c=min(dist1);

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
      if (length(x1)>1)
         x1=x1(1);
         y1=y1(1);
```

```
      end
      PF_angle=beam_angle;
      break
    else
        c=100;
        x1=0;
        y1=0;
    end

    %%%%% Check intersections with obstacle 4
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX4, concaveY4);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
      for j=1:length(x1)
        dist1(j,:)= sqrt((x1(j)-robot_pose(1)).^2+(y1(j)-
robot_pose(2)).^2);
      end
      c=min(dist1);

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
      if (length(x1)>1)
         x1=x1(1);
         y1=y1(1);
      end
      PF_angle=beam_angle;
      break
    else
        c=100;
        x1=0;
        y1=0;
    end

    %%%%% Check intersections with obstacle 5
    [x1,y1]=polyxpoly(x_ob, y_ob, concaveX5, concaveY5);

    % If there are two intersections, caculate all the
distances
    dist1=zeros(length(x1),1);
    if (isempty(dist1)==false)
```

```
      for j=1:length(x1)
        dist1(j,:)= sqrt((x1(j)-robot_pose(1)).^2+(y1(j)-
robot_pose(2)).^2);
      end
      c=min(dist1);

      % Just point the closest one
      [~,ind] = sort(dist1);
      ind(1)=ind(1);
      ind(2)=ind(1);
      x1=x1(ind,:);
      y1=y1(ind,:);
      if (length(x1)>1)
         x1=x1(1);
         y1=y1(1);
      end
      PF_angle=beam_angle;
      break
    else
        c=100;
        x1=0;
        y1=0;
    end
end

% Return obst_dist and obst_angle values to the main
intersection_points=[c,PF_angle,x1,y1];
```

Appendix C

LabVIEW for LEGO MINDSORMS NXT Robot

C.1 NI LabVIEW 2013

We will present a brief introduction to how to use LabVIEW 2013. Like what we have talked about in Section 7.1, LabVIEW from National Instruments (NI) is a development environment platform utilized for students, researchers and hobbyists. The G (graphical) language is the language used in LabVIEW. LabVIEW programs are called virtual instruments (VIs). Each VI has three components: a front panel, a block diagram and a connector pane.

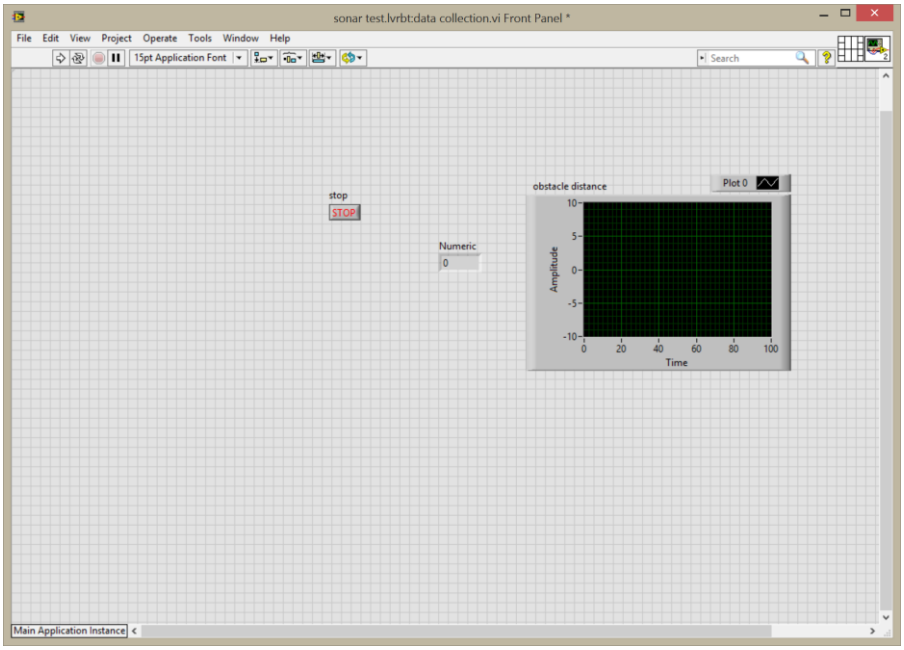
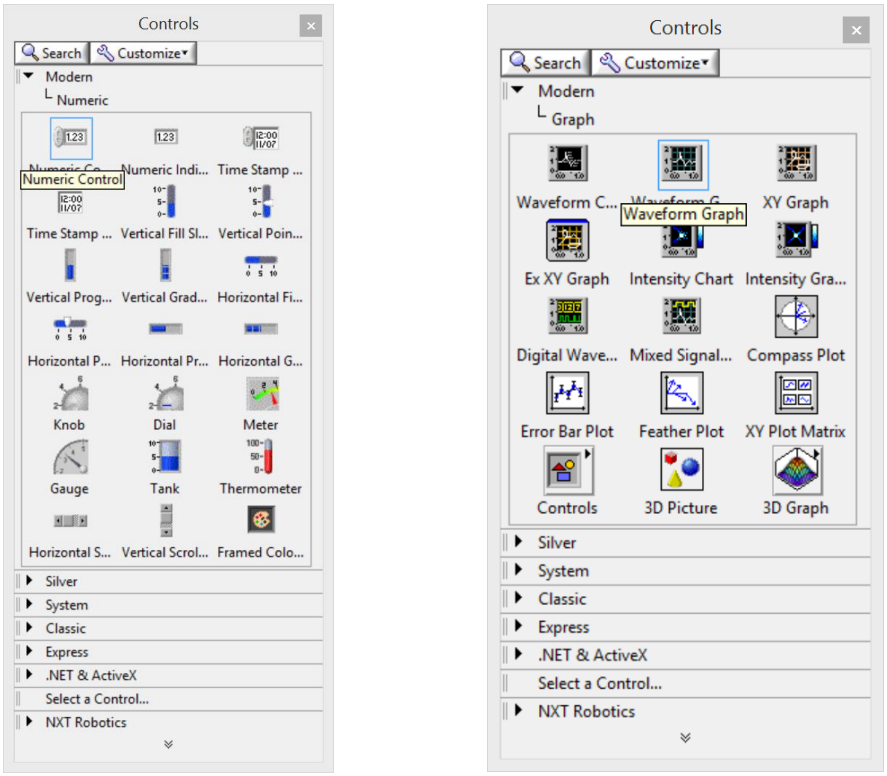


Figure C.1 Front panel of a VI.

The front panel is a user interface built using controls and indicators shown in Figure C.1. Controls are inputs allowing a user to supply information to the VI, and indicators are outputs indicating and displaying the results based on the inputs given. Frequently-used controls and indicators are illustrated in Figure C.2.



(a) (b)

Figure C.2 (a) Frequently-used controls; (b) Frequently-used indicators.

The back panel is a block diagram containing the graphical source code which is our main program. All the controls and indicators placed on the front panel will appear on the back panel as terminals. The controls and indicators are connected by various structures and functions user defined in the back panel. The back panel is like the main program working in the background, and the inputs and the outputs are shown in the front panel working like a GUI interface. Figure C.3 illustrates an example back panel of a VI in LabVIEW 2013. Frequently-used structures and elementary functions palettes are illustrated in Figure C.4 (a) and (b). For loop, while loop and case structures are structures we often used; trigonometric and exponential functions are frequently-used functions along with some elementary numeric

functions such as addition, subtracting, multiplying and dividing.

There are certain nodes for each block in LabVIEW, and we connect each block through wires connecting with nodes. For example, in Figure C.3, we connected an ultrasonic sensor to an indicator in order to obtain the distance of obstacle the ultrasonic sensor detected. In the back panel, the output node (the distance) of an ultrasonic sensor was wired to the input node of a numeric indicator.

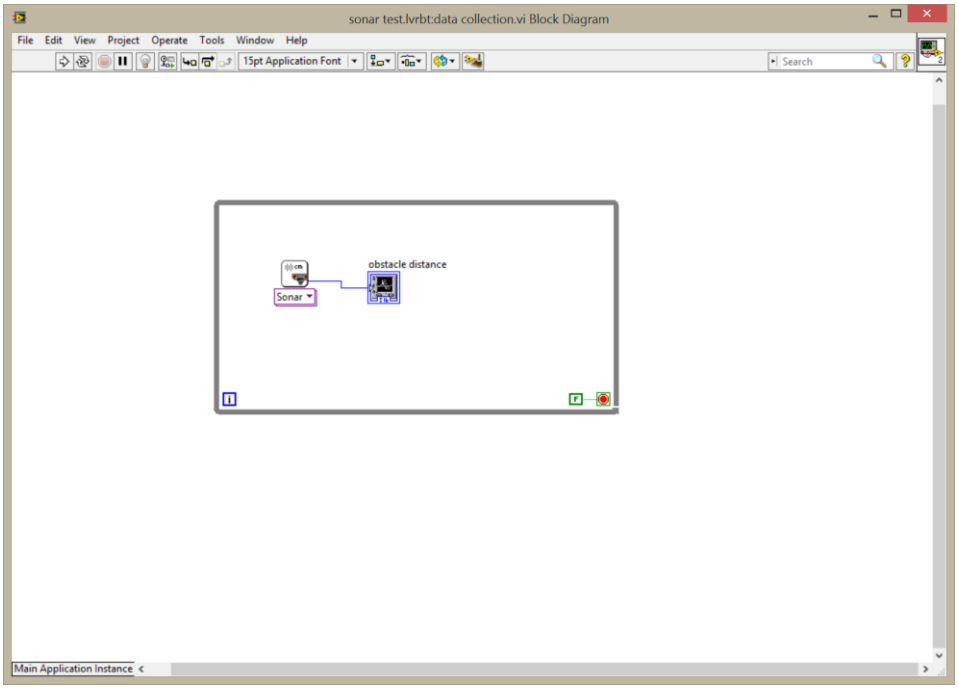
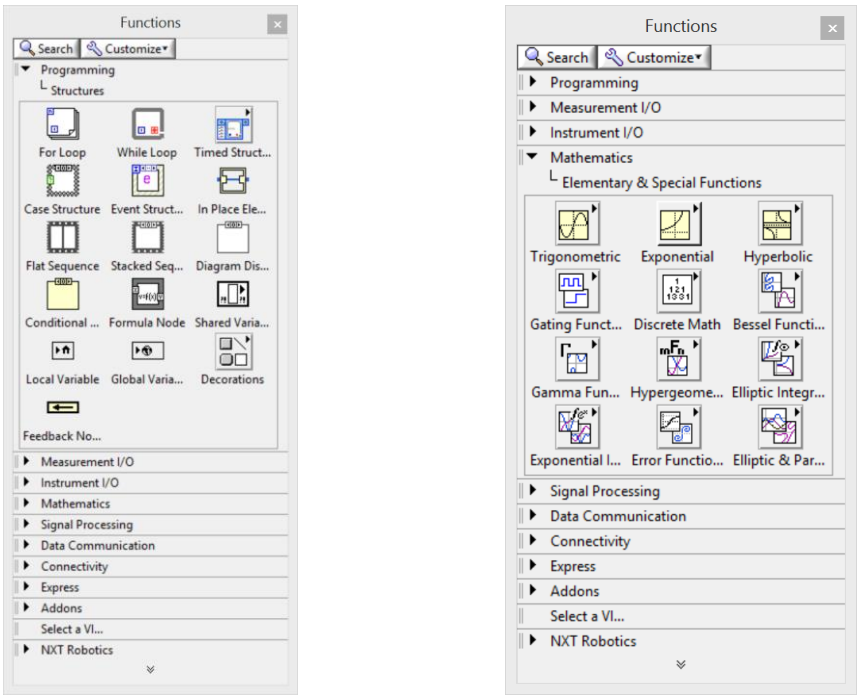


Figure C.3 Back panel of a VI.



(a) (b)

Figure C.4 (a) Frequently-used structures; (b) Frequently-used elementary functions.

The connector pane is used in a subVI (similar to a sub function) in order to define how the inputs and outputs of the subVI appear on a VI and how the inputs and outputs relate to the controls and indicators.

A sub VI is another LabVIEW VI that we can create and place anywhere on the block diagram just like with the functions and Vis available in the Functions palette. The subVI has three main parts: the actual code (the front panel and the block diagram), the icon and the connector pane. The actual code is the LabVIEW code that we used with the typical wires, terminals, controls and indicators. For example, Figure C.5 illustrates an example LabVIEW block diagram code in back panel. The control in the left side indicates the obstacle distance, and the indicator denotes the dimension of obstacle.

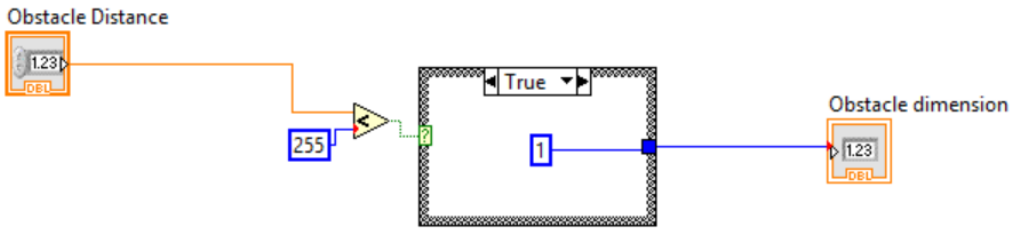


Figure C.5 An example LabVIEW block diagram code.

We will create a subVI for code in Figure C.5 because we will use the case structure in Figure C.5 plenty of times. To create a subVI, firstly, we need highlight the code we want to turn into a subVI like Figure C.6. Then click the *Edit* menu and select *Create SubVI*. After that, we can see all the code we highlighted condensed into one icon which means we have successfully create a subVI, shown in Figure C.7.

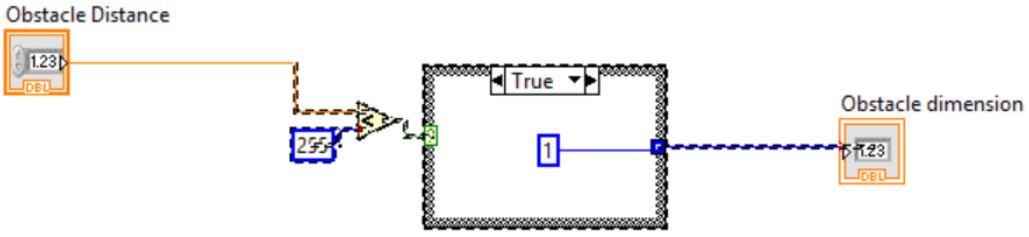


Figure C.6 Highlight the code we want to create a subVI.

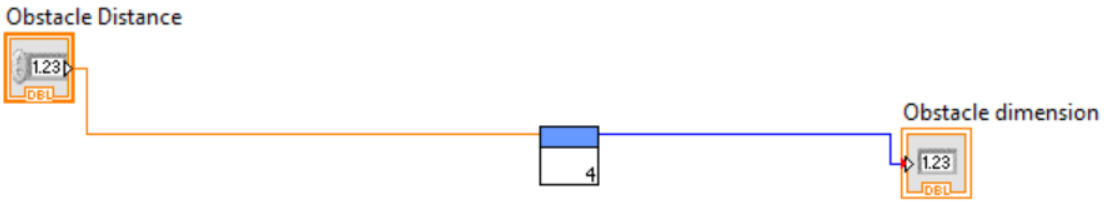


Figure C.7 Successfully create a subVI.

Mention that we can create our own icon based on our custom. A well designed icon helps us easily identify the function of the subVI. We can access the *Icon Editor* window when

opening the subVI we created and double clicking the icon in the upper right corner of the block diagram. Figure C.8 shows the *Icon Editor* window.

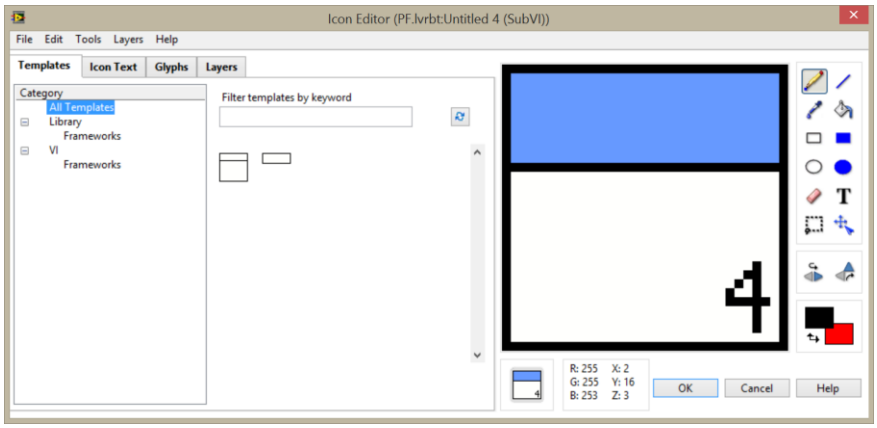


Figure C.8 *Icon Editor* window.

Since this subVI is used to detect whether or not there is an obstacle in front, we created our own icon in Figure C.9 in order to better identify the function of this subVI. Save our icon design, our code in the block diagram is shown in Figure C.10 with our costumed icon design.

The last step to successfully create a subVI is to use the connector pane to ensure that the input and output terminals are in the right place on the subVI. There are a default set of patterns that we can use, and the connector pane is next to the VI icon at the top right of the window shown in Figure C.11.

Usually we do not need change the connector pane since LabVIEW finds and wires a pattern for the controls and indicators by default when we created a subVI. There patterns represent preset terminals where we can set inputs and outputs.

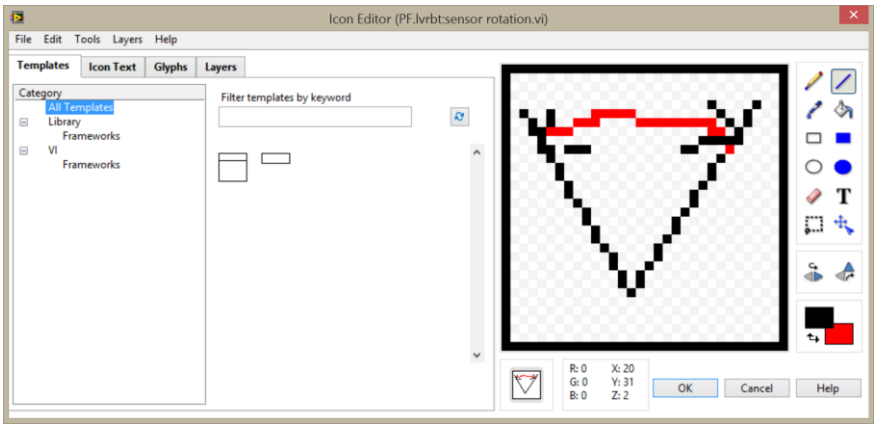


Figure C.9 Our own icon design.

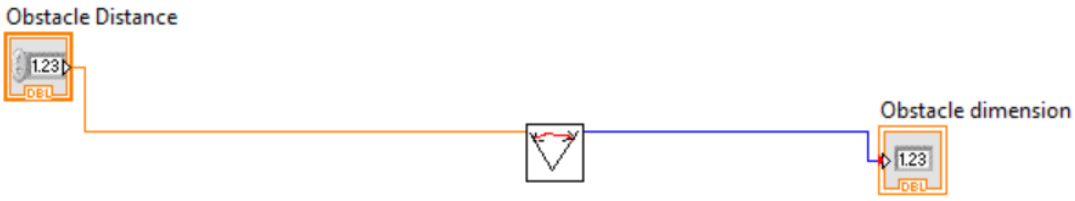


Figure C.10 A subVI with costumed icon design.

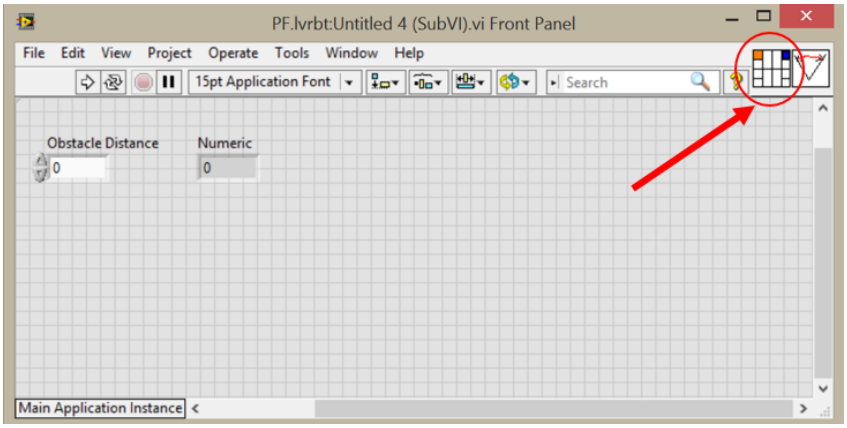


Figure C.11 Illustration of connector pane.

To change the pattern used in the connector pane, just right-click the connector pane, select *Patterns*, and select the pattern to be used. Figure C.12 shows all patterns users can choose.

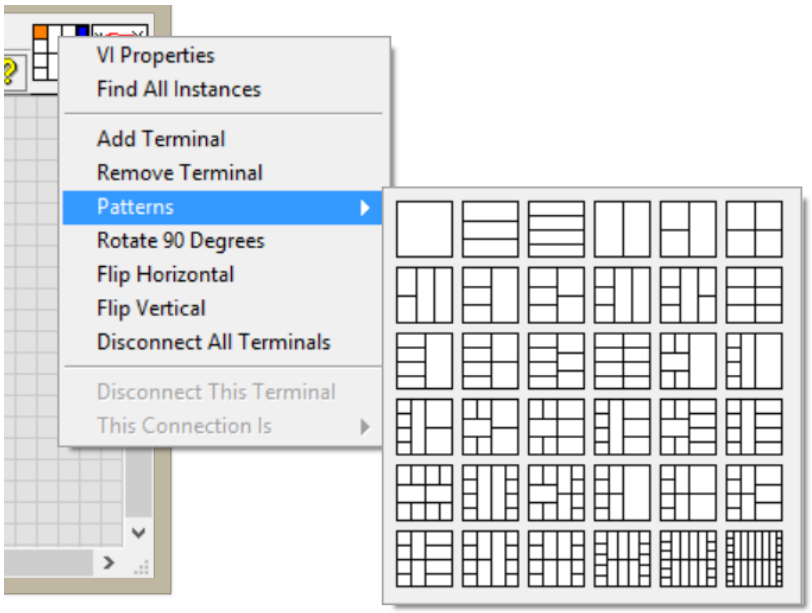


Figure C.12 All patterns users can choose from.

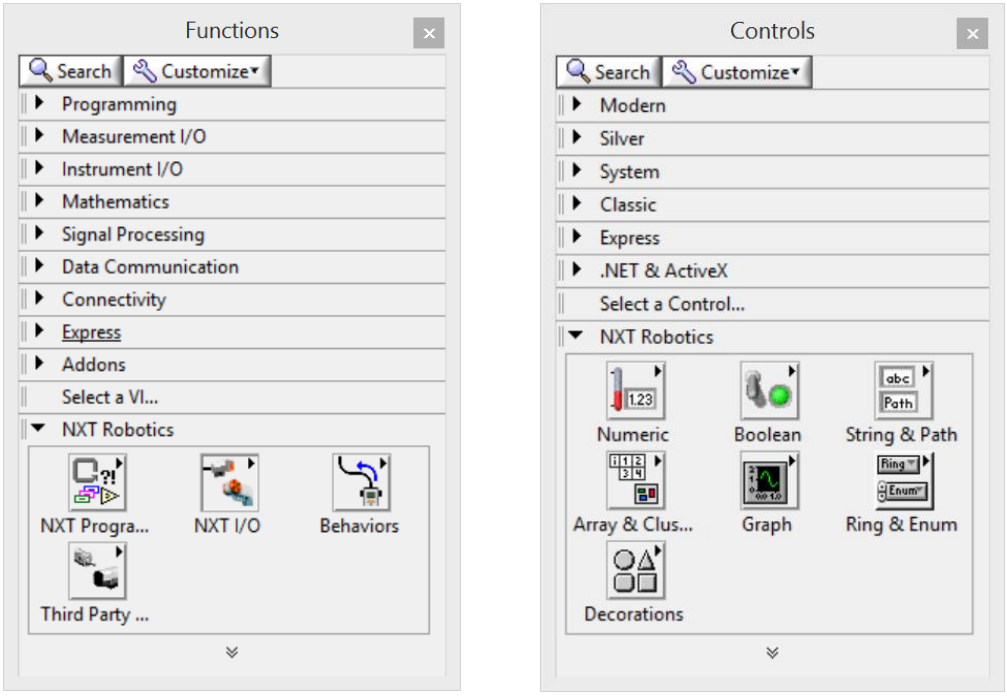
C.2 NI LabVIEW 2013 with LEGO MINDSTORMS NXT Module

We have introduced NI LabVIEW 2013 in Section C.1. In order to control the NXT robot, LEGO MINDSTORMS NXT Module patch needs to be installed into NI LabVIEW 2013. This patch provides a series of LEGO NXT controllers and indicators shown in Figure C.13 (a) and (b).

In NXT Programming functions there are certain typical functions such as structure functions, numeric functions and comparison functions. In NXT I/O functions, there are some frequently-used functions to control the motor, the sensors and so on.

After installing the NXT Module, we can create a robot project in the drop-down menu of File menu in the start menu of LabVIEW 2013 shown in Figure 7.1. We need name our robot project, then a pop-up menu Robot Project Center in Figure C.14 shows up. In the left column of the menu, *Choose NXT* indicates we need find our NXT robot by either USB or Bluetooth; *Schematic* denotes we need configure our robot sensors and motors to specific input/output ports. In the middle column is our robot files, the controller is often written in a VI subject to

the folder Programs.



(a) (b)

Figure C.13 (a) LEGO NXT controllers; (b) LEGO NXT indicators.

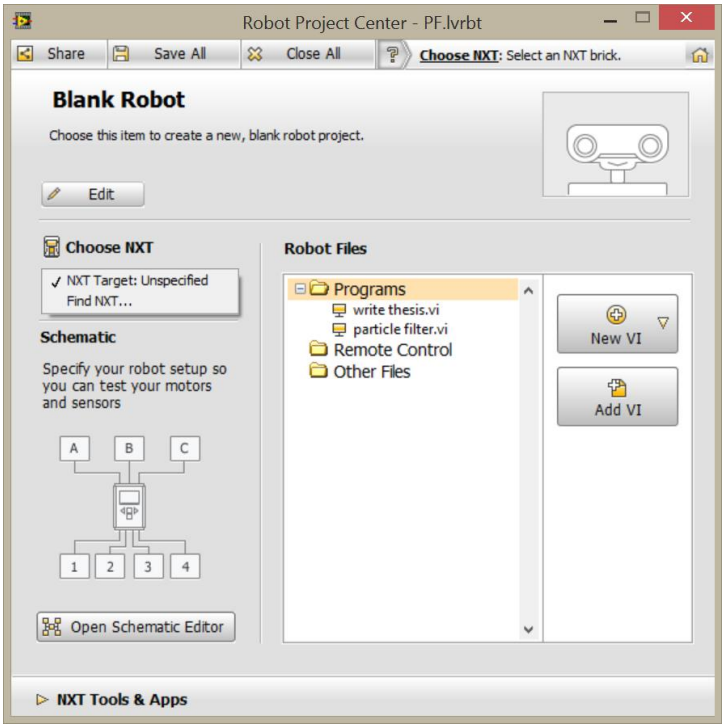


Figure C.14 Robot Project Center.

Appendix D

LabVIEW Code: Obstacle Avoidance Algorithm

D.1 Velocity Potential Field Approach

In our experiment, obstacle avoidance algorithm is based on our proposed velocity potential field approach, the normal repulsive velocity is defined in Equation (3.22), and the tangential repulsive velocity is defined in Equation (3.23). The overall velocity of the robot when it is too close to obstacles is a function of the distance of robot to obstacle. Figure D.1 illustrates the function given to the robot motor in LabVIEW.

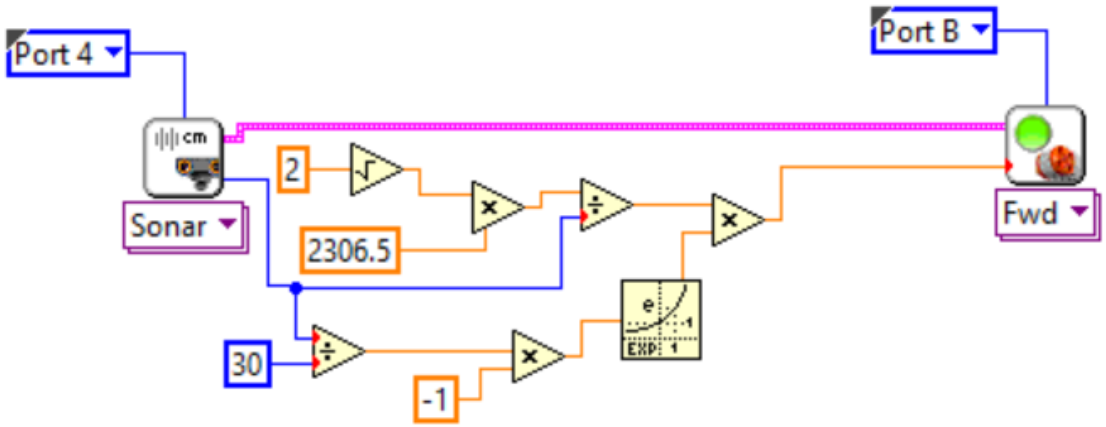


Figure D.1 The function given to the robot motor in LabVIEW.

D.2 Obstacle Avoidance Algorithm

In this section, LabVIEW obstacle avoidance algorithm will be presented. The flowchart of algorithm has been already explained in Chap.3. Also, we have introduced some basic programming procedure about LabVIEW and some meanings of LabVIEW blocks for the NXT robot in Table 7.1. So we can directly go into the source code.

In Appendix C.1, we have shown one example of how to build a subVI, while this subVI is going to be utilized in our obstacle avoidance algorithm. Mention that the function of that subVI is to check if the ultrasonic sensor detects obstacles. If yes (which means we have the value of the distance less than 255cm), return the number 1; if no obstacles detected, return the number 0.

Figure D.2 illustrates the code regarding to obstacles are not too close to the robot, which means robot does not need turn away from obstacles even if robot already detected obstacles.

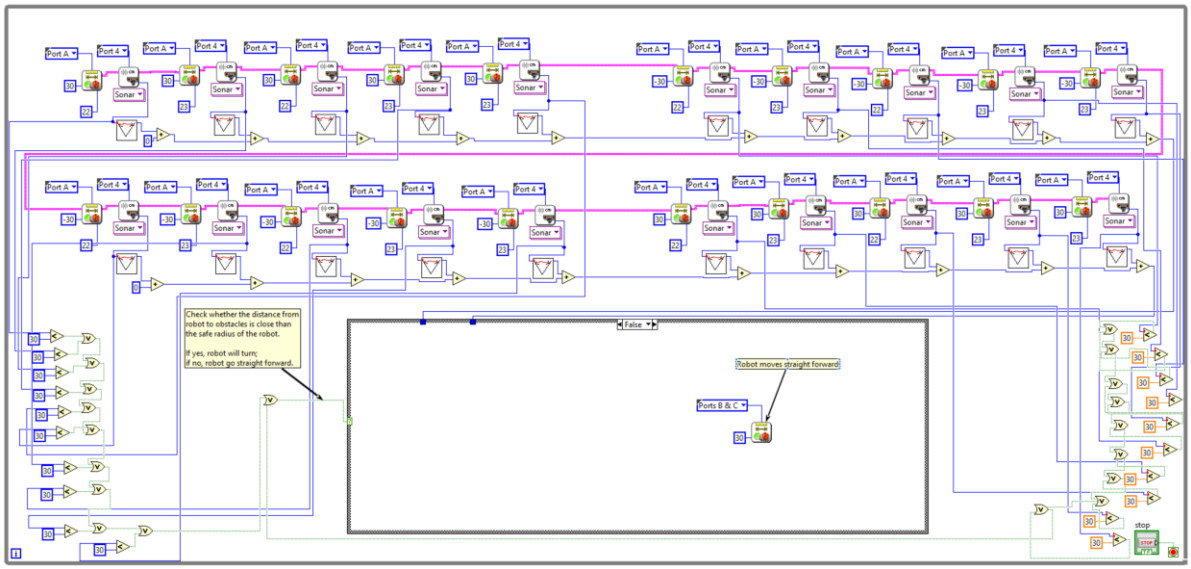


Figure D.2 Obstacles not too close to robot.

In our experiment, we turned the ultrasonic sensor π/8 per step, and our searching range of our sensor is from - π/8 to 9*π/8. We divided this range into ten sections, and we took 10 measurements per sensing cycle. In Figure D.2, the first 10 measurements are shown in the first row of the code, and all these 10 measurements are regarding to the right part of sensing section of robot. Hence, we can obtain the right obstacles dimension through adding all the output of our subVIs.

In the same way, we can obtain the left obstacles dimension carried out by the second row of the code. Mention that in the left bottom and right bottom of the code lies some codes corresponding to checking if obstacles enter into the safety radius of the robot. If yes, robot will be applied obstacle avoidance algorithm in order to turn away from obstacles; if no, robot

will go straight forward. Figure D.2 shows the case that robot goes forward since the distance of robot to obstacles is larger than the safety radius of robot.

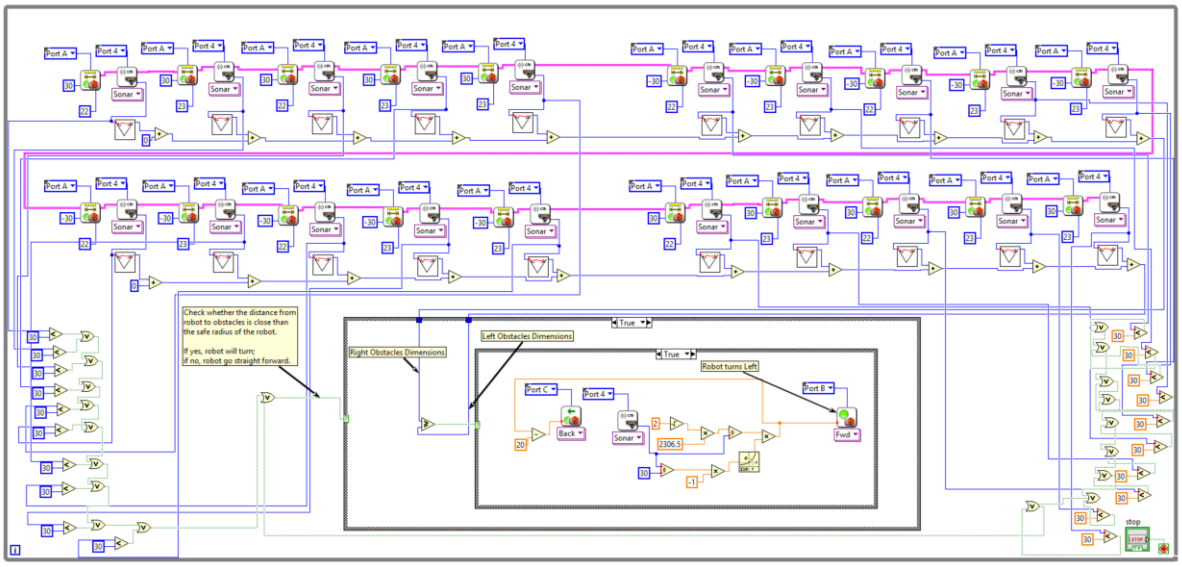


Figure D.3 Robot turns left case.

Figure D.3 illustrates the case where robot turns left. That is because one: robot stands too close to obstacles; the second: right obstacles dimension is larger than left one. We used two case structures in our code, the outermost case structure decides if robot will turn; and the innermost one chooses in which direction robot will turn.

Similarly, robot turns right case is demonstrated in Figure D.4. Mention that we drove robot turning by setting two servo motor different turning speed (including the value and direction). That is why we can see the code in innermost case structure for both robot turns left case and robot turns right case looks similar. We just switched the servo motor ports from port B to C so that robot can change its turning direction.

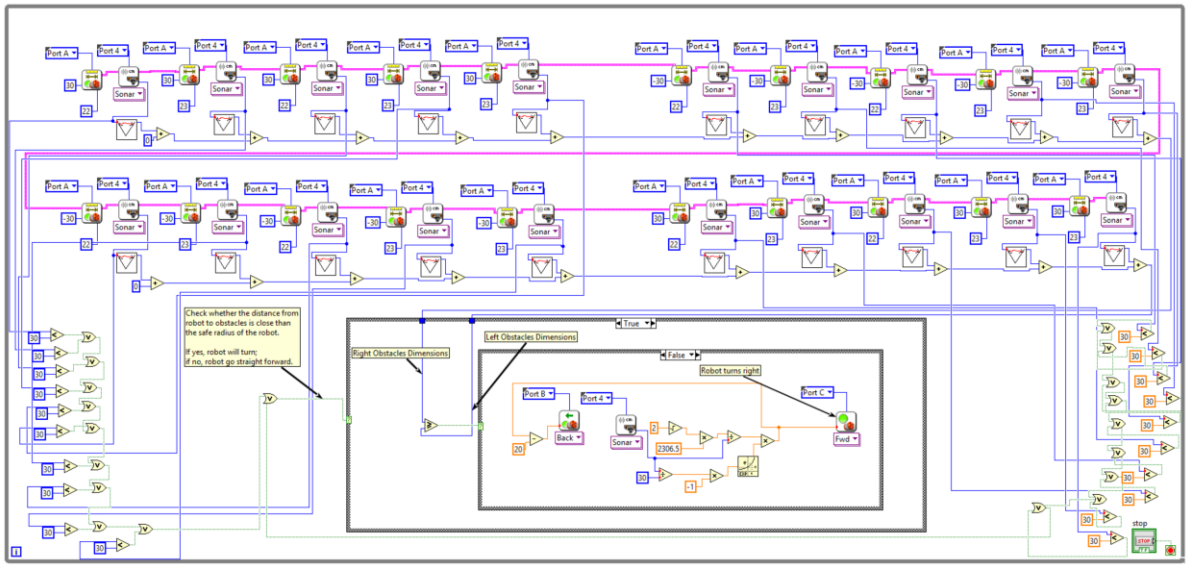


Figure D.4 Robot turns right case.

Appendix E

MATLAB Code for Experiment: Robot Navigation Using the FastSLAM Approach

E.1 Robot Navigation Using the FastSLAM Approach with Two Obstacles Avoidance

E.1.1 Main Function

```
clc
clf
clear all

% Setting the random seed, so the same example can be run
several times
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);

% Robot path
robot_pose=[10,-30,pi/2;10,-27.5,pi/2;10,-25,pi/2;10,-
25,3*pi/4;...
    10,-25,pi;7.835,-23.75,5*pi/6;6.585,-21.585,2*pi/3;6.585,-
19.085,pi/2;...
    7.232,-16.67,5*pi/12;8.482,-14.5,pi/3;9.732,-
12.335,pi/3;10.982,-10.17,pi/3];

% Measurement values, the first value denotes the angle of the
measurement beam
% in the local coordinate of robot, the second value indicates
the distance
% from robot to the obstacle (in decimeters)
measurement_data=[67.5*pi/180,7.7;67.5*pi/180,5.0;67.5*pi/180,
2.5;...
    0*pi/180,2.9;-22.5*pi/180,2.2;-22.5*pi/180,2.9;-
22.5*pi/180,1.8;...
```

```
    -22.5*pi/180,2.1;-
22.5*pi/180,5;90*pi/180,5.9;90*pi/180,2.6;90*pi/180,0.06];

% Transfer the local angles of robot to global angles
global_angle=[67.5*pi/180;67.5*pi/180;67.5*pi/180;pi/4;3*pi/8;
5*pi/24;...
    pi/24;-pi/8;-5*pi/24;pi/3;pi/3;pi/3];

% Goal Position on the map
goal=[10.985,-10.173];

% Plot goal and the circle of interest around the goal
scatter(goal(1),goal(2),1000,'o','m','filled');

% Axis properties
axis([-5,35,-50,0]);
axis equal;
axis manual;
hold on;

% Log the goal and obstacles
text(11.5,-10,'G','FontSize',18)
text(9,-21,'O_1','FontSize',14,'FontWeight','bold')
text(12,-21,'O_2','FontSize',14,'FontWeight','bold')

% Concave obstacle 1 dimensions
xlimit=[9,9.5,9.5,9];
ylimit=[-22,-22,-20,-20];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);
fill(concaveX1, concaveY1,'y','edgealpha',0);
alpha(.5); % Set the transparency

% Concave obstacle 2 dimensions
xlimit=[10.75,14,14,10.75];
ylimit=[-22.5,-22.5,-19.5,-19.5];
concaveX2=xlimit([4 1 2 3 4]);
concaveY2=ylimit([4 1 2 3 4]);
fill(concaveX2, concaveY2,'y','edgealpha',0);
alpha(.5);

% Some unceratinty parameters
measurementNoiseStdev = 0.1; speedStdev = 1;

% Some parameters for plotting the particles
m = 1000; k = 0.01;
```

```
% Number of particles
N = 500;

% Initialize particle weights
w = 1/N*ones(N,1);
w_goal = 1/N*ones(N,1);

% Creat normally distribution particles around the the know
initial
% position of the robot

p0=zeros(N,2);
p0(:,1)=robot_pose(1,1)*ones(N,1)+randn(N,1);
p0(:,2)=robot_pose(1,2)*ones(N,1)+randn(N,1);

plot_virtual_obstacle_position=zeros(12,2);
% Do some initialization
px_goal=0;py_goal=0;
px=0;py=0;

for t=1:12


    % Plot robot poses
    zzzz=zeros(t,2);
    for i=1:t
        zzzz(i,1)=robot_pose(i,1);
        zzzz(i,2)=robot_pose(i,2);
    end

    plot(zzzz(:,1),zzzz(:,2),'-r^','MarkerFaceColor','r');

    % Set measurement noises
    measurebearing_error=randn*measurementNoiseStdev/30;
    measuredistance_error=randn*measurementNoiseStdev;

    % Obtain virtual obstacle position as the reference point

virtual_obstacle_position=[robot_pose(t,1)+(measurement_data(t
,2)-measuredistance_error)...
        *cos(global_angle(t)-
measurebearing_error),robot_pose(t,2)+(measurement_data(t,2)-
measuredistance_error)...
        *sin(global_angle(t)-measurebearing_error)];
```

```
plot_virtual_obstacle_position(t,1)=virtual_obstacle_position(
1);

plot_virtual_obstacle_position(t,2)=virtual_obstacle_position(
2);



    if t<10
    %%%%%%%%%%%%%%%%%%%%% Particle
filtering %%%%%%%%%%%%%%%%%%%%%%%%%

    %%%%%%%%%%%%%%%%%%%%% Robot Position
Estimation %%%%%%%%%%%%%%%%

    particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
    pause(.5)
    delete(particleHandle);


    % Generate bearing and distance measurements (with
gaussian measurement noise)
    measurebearing=global_angle(t);
    measuredistance=measurement_data(t,2);

    % Calculate corresponding parameters for all particles
    predBrg =
TrueBearing(p0,[virtual_obstacle_position(1)*ones(N,1),virtual
_obstacle_position(2)*ones(N,1)]);
    predBrg1 =
TrueDistance(p0,[virtual_obstacle_position(1)*ones(N,1),virtua
l_obstacle_position(2)*ones(N,1)]);

    % Evaluate measurements (i.e., create weights) using the
pdf for the normal distribution
    brgWts = w.*(1/(sqrt(2*pi)*measurementNoiseStdev/30)*exp(-
(predBrg-
measurebearing).^2/(2*(measurementNoiseStdev/30)^2)));
    rngWts = w.*(1/(sqrt(2*pi)*measurementNoiseStdev)*exp(-
(predBrg1-measuredistance).^2/(2*measurementNoiseStdev^2)));

    % Combine range and bearing weights
    f_Wts=brgWts.* rngWts;

    % Normalize particle weigths and plot particles
```

```
    w = f_Wts/sum(f_Wts);

    particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
    pause(.5)
    delete(particleHandle);

    scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');

    % Resample the particles
    u = rand(N,1); wc = cumsum(w);
    [~,ind1] = sort([u;wc]); ind=find(ind1<=N)-(0:N-1)';
    p0=p0(ind,:); w=ones(N,1)./N;

    % Plot the particles after resampling (may have many
overlapping particles)
    particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
    pause(.5)
    delete(particleHandle);

    % Get the mean
    for i=1:N
        px=p0(i,1)+px;
    end
    px=px/N;

    for i=1:N
        py=p0(i,2)+py;
    end
    py=py/N;

    plot(px,py,':bs','MarkerFaceColor','g');


     % Time propagation
     delta_d=[robot_pose(t+1,1)-
robot_pose(t,1),robot_pose(t+1,2)-robot_pose(t,2)];
     speedNoisex=speedStdev*rand(N,1);
     speedNoisey=speedStdev*randn(N,1);
     p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
     p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;

    %%%%%%%%%%%%%%%%%%%%%%% Landmark Location
Estimation %%%%%%%%%%%
```

```
    px1=px+(measurement_data(t,2)-
measuredistance_error)*cos(global_angle(t)-
measurebearing_error);
    py1=py+(measurement_data(t,2)-
measuredistance_error)*sin(global_angle(t)-
measurebearing_error);
    plot([px,px1],[py,py1],'--
b*','MarkerEdgeColor','g','MarkerSize',10);

    else

        %%%%%%%%%%%%%%%% Particle Filtering wrt the
goal %%%%%%%%%%%%%%%%

        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause(.5)
        delete(particleHandle);

        % Generate bearing and distance measurements wrt the
goal (with gaussian measurement noise)
        measurebearing_goal=global_angle(t);
        measuredistance_goal=measurement_data(t,2);

        % Calculate corresponding parameters for all particles
        predBrg_goal =
TrueBearing(p0,[goal(1)*ones(N,1),goal(2)*ones(N,1)]);
        predBrg1_goal =
TrueDistance(p0,[goal(1)*ones(N,1),goal(2)*ones(N,1)]);

        % Evaluate measurements (i.e., create weights) using
the pdf for the normal distribution
        brgWts_goal =
w_goal.*(1/(sqrt(2*pi)*measurementNoiseStdev/10)*exp(-
(predBrg_goal-
measurebearing_goal).^2/(2*(measurementNoiseStdev/10)^2)));
        rngWts_goal =
w_goal.*(1/(sqrt(2*pi)*measurementNoiseStdev/1)*exp(-
(predBrg1_goal-
measuredistance_goal).^2/(2*(measurementNoiseStdev/1)^2)));
        % Combine range and bearing weights
        f_Wts_goal=brgWts_goal.* rngWts_goal;

        % Normalize particle weigths and plot particles
        w_goal = f_Wts_goal/sum(f_Wts_goal);
```

```
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause(.5)
        delete(particleHandle);

        scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');

        % Resample the particles
        u_goal = rand(N,1); wc_goal = cumsum(w_goal);
        [~,ind1] = sort([u_goal;wc_goal]); ind=find(ind1<=N)-
(0:N-1)';
        p0=p0(ind,:); w_goal=ones(N,1)./N;

        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause(.5)
        delete(particleHandle);
        for i=1:N
            px_goal=p0(i,1)+px_goal;
        end
        px_goal=px_goal/N;

        for i=1:N
            py_goal=p0(i,2)+py_goal;
        end
        py_goal=py_goal/N;

        plot(px_goal,py_goal,'--bd','MarkerFaceColor','g');

        if t<12

            % Time propagation
            delta_d=[robot_pose(t+1,1)-
robot_pose(t,1),robot_pose(t+1,2)-robot_pose(t,2)];
            speedNoisex=speedStdev*randn(N,1);
            speedNoisey=speedStdev*randn(N,1);
            p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
            p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;
        end

        %%%%%%%%%%%%%%%% Plot the measurements wrt the
goal %%%%%%%%%%%%

        plot([px_goal,goal(1)],[py_goal,goal(2)],'--
b*','MarkerEdgeColor','g','MarkerSize',10);
```

```
    end
end
```

E.2 Robot Navigation Using the FastSLAM Approach with a Concave Obstacle Avoidance

E.2.1 Main Function

```
clc
clf
clear all

% Setting the random seed, so the same example can be run
several times
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);

% Robot path
robot_pose=[10,-30,pi/2;10,-27.5,pi/2;10,-25,pi/2;10,-
25,3*pi/4;...
    10,-25,pi;7.835,-23.75,5*pi/6;6.585,-21.585,2*pi/3;6.585,-
19.085,pi/2;...
    7.232,-16.67,5*pi/12;8.482,-14.5,pi/3;9.732,-
12.335,pi/3;10.982,-10.17,pi/3];

% Measurement values, the first value denotes the angle of the
measurement beam
% in the local coordinate of robot, the second value indicates
the distance
% from robot to the obstacle (in decimeters)
measurement_data=[22.5*pi/180,5.3;0*pi/180,4.9;0*pi/180,5.3;..
.
    0*pi/180,4.2;-22.5*pi/180,4.1;0*pi/180,1.2;-
22.5*pi/180,1.8;...
    -22.5*pi/180,3.5;-
22.5*pi/180,4.2;90*pi/180,5.5;90*pi/180,2.5;90*pi/180,0.05];

% Transfer the local angles of robot to global angles
global_angle=[22.5*pi/180;0*pi/180;0*pi/180;pi/4;3*pi/8;pi/3;.
..
    pi/24;-pi/8;-5*pi/24;pi/3;pi/3;pi/3];

% Goal Position on the map
goal=[10.985,-10.173];


% Plot goal and the circle of interest around the goal
```

```
scatter(goal(1),goal(2),1000,'o','m','filled');

% Axis properties
axis([-5,35,-50,0]);
axis equal;
axis manual;
hold on;

text(11.5,-10,'G','FontSize',18)
text(9,-22,'O_1','FontSize',14,'FontWeight','bold')
text(11.75,-20.75,'O_2','FontSize',14,'FontWeight','bold')
text(13.5,-22,'O_3','FontSize',14,'FontWeight','bold')
text(15.5,-25.5,'O_4','FontSize',14,'FontWeight','bold')
text(15.75,-29,'O_5','FontSize',14,'FontWeight','bold')

% Concave obstacle 1 dimensions
xlimit=[8.5,10.5,10.5,8.5];
ylimit=[-22.5,-22.5,-21.5,-21.5];
concaveX1=xlimit([4 1 2 3 4]);
concaveY1=ylimit([4 1 2 3 4]);
fill(concaveX1, concaveY1,'y','edgealpha',0);
alpha(.5); % Set the transparency

% Concave obstacle 2 dimensions
xlimit=[10.75,12.75,12.75,10.75];
ylimit=[-21,-21,-20,-20];
concaveX2=xlimit([4 1 2 3 4]);
concaveY2=ylimit([4 1 2 3 4]);
fill(concaveX2, concaveY2,'y','edgealpha',0);
alpha(.5);

% Concave obstacle 3 dimensions
xlimit=[13,15,15,13];
ylimit=[-22.5,-22.5,-21.5,-21.5];
concaveX3=xlimit([4 1 2 3 4]);
concaveY3=ylimit([4 1 2 3 4]);
fill(concaveX3, concaveY3,'y','edgealpha',0);
alpha(.5);

% Concave obstacle 4 dimensions
xlimit=[15.25,16.25,16.25,15.25];
ylimit=[-26,-26,-24,-24];
concaveX4=xlimit([4 1 2 3 4]);
concaveY4=ylimit([4 1 2 3 4]);
fill(concaveX4, concaveY4,'y','edgealpha',0);
alpha(.5);
```

```
% Concave obstacle 5 dimensions
xlimit=[15,16.75,16.75,15];
ylimit=[-30,-30,-27,-27];
concaveX5=xlimit([4 1 2 3 4]);
concaveY5=ylimit([4 1 2 3 4]);
fill(concaveX5, concaveY5,'y','edgealpha',0);
alpha(.5);

% Some unceratinty parameters
measurementNoiseStdev = 0.1; speedStdev = 1;

% Some parameters for plotting the particles
m = 1000; k = 0.01;

% Number of particles
N = 500;

% Initialize particle weights
w = 1/N*ones(N,1);
w_goal = 1/N*ones(N,1);

% Creat normally distribution particles around the the know
initial
% position of the robot
p0=zeros(N,2);
p0(:,1)=robot_pose(1,1)*ones(N,1)+randn(N,1);
p0(:,2)=robot_pose(1,2)*ones(N,1)+randn(N,1);

plot_virtual_obstacle_position=zeros(12,2);

px_goal=0;py_goal=0;
px=0;py=0;

for t=1:12

    zzzz=zeros(t,2);
    for i=1:t
        zzzz(i,1)=robot_pose(i,1);
        zzzz(i,2)=robot_pose(i,2);
    end

    plot(zzzz(:,1),zzzz(:,2),'-r^','MarkerFaceColor','r');

    measurebearing_error=randn*measurementNoiseStdev/30;
    measuredistance_error=randn*measurementNoiseStdev;
```

```
virtual_obstacle_position=[robot_pose(t,1)+(measurement_data(t
,2)-measuredistance_error)...
        *cos(global_angle(t)-
measurebearing_error),robot_pose(t,2)+(measurement_data(t,2)-
measuredistance_error)...
        *sin(global_angle(t)-measurebearing_error)];


plot_virtual_obstacle_position(t,1)=virtual_obstacle_position(
1);

plot_virtual_obstacle_position(t,2)=virtual_obstacle_position(
2);

    if t<10
    %%%%%%%%%%%%%%%%%%%%%% Particle
filtering %%%%%%%%%%%%%%%%%%%%%%%%%%
    particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
    pause(.5)
    delete(particleHandle);

    % Generate bearing and distance measurements (with
gaussian measurement noise)
    measurebearing=global_angle(t);
    measuredistance=measurement_data(t,2);

    % Calculate corresponding parameters for all particles
    predBrg =
TrueBearing(p0,[virtual_obstacle_position(1)*ones(N,1),virtual
_obstacle_position(2)*ones(N,1)]);
    predBrg1 =
TrueDistance(p0,[virtual_obstacle_position(1)*ones(N,1),virtua
l_obstacle_position(2)*ones(N,1)]);

    % Evaluate measurements (i.e., create weights) using the
pdf for the normal distribution
    brgWts = w.*(1/(sqrt(2*pi)*measurementNoiseStdev/30)*exp(-
(predBrg-
measurebearing).^2/(2*(measurementNoiseStdev/30)^2)));
    rngWts = w.*(1/(sqrt(2*pi)*measurementNoiseStdev)*exp(-
(predBrg1-measuredistance).^2/(2*measurementNoiseStdev^2)));

    % Combine range and bearing weights
    f_Wts=brgWts.* rngWts;
```

```
    % Normalize particle weigths and plot particles
    w = f_Wts/sum(f_Wts);

    particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
    pause(.5)
    delete(particleHandle);

    scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');

    % Resample the particles
    u = rand(N,1); wc = cumsum(w);
    [~,ind1] = sort([u;wc]); ind=find(ind1<=N)-(0:N-1)';
    p0=p0(ind,:); w=ones(N,1)./N;

    % Plot the particles after resampling (may have many
overlapping particles)
    particleHandle =
scatter(p0(:,1),p0(:,2),m*(w+k),'k','filled');
    pause(.5)
    delete(particleHandle);


    % Get the mean
    for i=1:N
        px=p0(i,1)+px;
    end
    px=px/N;

    for i=1:N
        py=p0(i,2)+py;
    end
    py=py/N;

    plot(px,py,':bs','MarkerFaceColor','g');


     % Time propagation
     delta_d=[robot_pose(t+1,1)-
robot_pose(t,1),robot_pose(t+1,2)-robot_pose(t,2)];
     speedNoisex=speedStdev*rand(N,1);
     speedNoisey=speedStdev*randn(N,1);
     p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
     p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;

    %%%%%%%%%%%%%%%%%%%%%%% Landmark Location
Estimation %%%%%%%%%%%
```

```
    px1=px+(measurement_data(t,2)-
measuredistance_error)*cos(global_angle(t)-
measurebearing_error);
    py1=py+(measurement_data(t,2)-
measuredistance_error)*sin(global_angle(t)-
measurebearing_error);
    plot([px,px1],[py,py1],'--
b*','MarkerEdgeColor','g','MarkerSize',10);

    else

        %%%%%%%%%%%%%%%% Particle Filtering wrt the
goal %%%%%%%%%%%%%%%%
        particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        delete(particleHandle);

        % Generate bearing and distance measurements wrt the
goal (with gaussian measurement noise)
        measurebearing_goal=global_angle(t);
        measuredistance_goal=measurement_data(t,2);

        % Calculate corresponding parameters for all particles
        predBrg_goal =
TrueBearing(p0,[goal(1)*ones(N,1),goal(2)*ones(N,1)]);
        predBrg1_goal =
TrueDistance(p0,[goal(1)*ones(N,1),goal(2)*ones(N,1)]);

        % Evaluate measurements (i.e., create weights) using
the pdf for the normal distribution
        brgWts_goal =
w_goal.*(1/(sqrt(2*pi)*measurementNoiseStdev/10)*exp(-
(predBrg_goal-
measurebearing_goal).^2/(2*(measurementNoiseStdev/10)^2)));
        rngWts_goal =
w_goal.*(1/(sqrt(2*pi)*measurementNoiseStdev/1)*exp(-
(predBrg1_goal-
measuredistance_goal).^2/(2*(measurementNoiseStdev/1)^2)));
        % Combine range and bearing weights
        f_Wts_goal=brgWts_goal.* rngWts_goal;

        % Normalize particle weigths and plot particles
        w_goal = f_Wts_goal/sum(f_Wts_goal);
```

```
particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        delete(particleHandle);

        scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');

        % Resample the particles
        u_goal = rand(N,1); wc_goal = cumsum(w_goal);
        [~,ind1] = sort([u_goal;wc_goal]); ind=find(ind1<=N)-
(0:N-1)';
        p0=p0(ind,:); w_goal=ones(N,1)./N;

particleHandle =
scatter(p0(:,1),p0(:,2),m*(w_goal+k),'k','filled');
        pause;
        delete(particleHandle);

        for i=1:N
            px_goal=p0(i,1)+px_goal;
        end
        px_goal=px_goal/N;

        for i=1:N
            py_goal=p0(i,2)+py_goal;
        end
        py_goal=py_goal/N;

        plot(px_goal,py_goal,'--bd','MarkerFaceColor','g');

        if t<12

            % Time propagation
            delta_d=[robot_pose(t+1,1)-
robot_pose(t,1),robot_pose(t+1,2)-robot_pose(t,2)];
            speedNoisex=speedStdev*randn(N,1);
            speedNoisey=speedStdev*randn(N,1);
            p0(:,1)=p0(:,1)+delta_d(1)*ones(N,1)+speedNoisex;
            p0(:,2)=p0(:,2)+delta_d(2)*ones(N,1)+speedNoisey;
        end

        %%%%%%%%%%%%%%%% Plot the measurements wrt the
goal %%%%%%%%%%%%

        plot([px_goal,goal(1)],[py_goal,goal(2)],'--
b*','MarkerEdgeColor','g','MarkerSize',10);
```

```
    end
end
```