8-2014

A COMPARATIVE STUDY OF UNDERWATER ROBOT PATH PLANNING ALGORITHMS FOR ADAPTIVE SAMPLING IN A NETWORK OF SENSORS

Sreeja Banerjee
University of Nebraska - Lincoln, sreeja.b.2010@gmail.com

A COMPARATIVE STUDY OF UNDERWATER ROBOT PATH PLANNING ALGORITHMS FOR ADAPTIVE SAMPLING IN A NETWORK OF SENSORS

by

Sreeja Banerjee

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Carrick Detweiler

Lincoln, Nebraska

August, 2014

A COMPARATIVE STUDY OF UNDERWATER ROBOT PATH PLANNING ALGORITHMS FOR ADAPTIVE SAMPLING IN A NETWORK OF SENSORS

Sreeja Banerjee, M. S.

University of Nebraska, 2014

Adviser: Carrick Detweiler

Monitoring lakes, rivers, and oceans is critical to improving our understanding of complex large-scale ecosystems. We introduce a method of underwater monitoring using semi-mobile underwater sensor networks and mobile underwater robots in this thesis. The underwater robots can move freely in all dimension while the sensor nodes are anchored to the bottom of the water column and can move only up and down along the depth of the water column. We develop three different algorithms to optimize the path of the underwater robot and the positions of the sensors to improve the overall quality of sensing of an area of water.

The algorithms fall into three categories based on knowledge of the environment: global knowledge, local knowledge, and a decentralized approach. The first algorithm, VoronoiPath, is a global path planning algorithm that uses the concept of *Voronoi Tessellation*. The second algorithm, TanBugPath, is a local path planning algorithm, inspired from the *Tangent Bug method* for obstacle avoidance. Finally, the third path planning algorithm, AdaptivePath, optimizes the path by balancing the distance covered by the underwater robot and maximizing the sensing efficiency of both the sensor and the robot. It is based on an adaptive decentralized algorithm and plans the path of the underwater robot by assigning robot waypoints along the depth of the water column, and then adapting them alongside the sensor nodes to obtain the path of the robot. It uses a stable gradient-descent based controller which, we show, converges to a local minimum.

We verify the algorithms through simulations and experiments. The VoronoiPath algorithm, generally, results in more efficient sensing paths. However, it is difficult to implement in real world as it needs global information and results in longer robot paths.

The TanBugPath algorithm, on the other hand, has good sensing and it plans paths which are a usually shorter under varying conditions. However, all the processing takes place on-board the mobile robot, hence, this approach needs a more advanced robot than other algorithms. Finally, in case of the AdaptivePath algorithm, the in-network sensors calculate the path of the mobile robot in a decentralized manner. A major advantage of this approach is that the the positions of the sensors in the water column also get optimized depending on the path of the mobile robot. However, this algorithm can get stuck in a local minima, and is also dependent on the starting positions of the robot waypoints. For each of the algorithms we perform a detailed analysis and comparison. We identify limitations of each, and provide framework for future improvements.

DEDICATION

To my parents, Sipra Banerjee and Satya Priya Banerjee, for their unwavering support throughout my life. To my first pet as a solo-owner, Ms. Squigglum, the droopy eyed chinkster a.k.a. Jr. Montmorency a.k.a. Montu, who rendered her life, as well as her death, to my graduate years.

ACKNOWLEDGMENTS

I am very grateful for the invaluable guidance and support given by Dr. Carrick J. Detweiler for the last three years in the NIMBUS Lab. My experience here and the lessons I have learned will be beneficial to me for the rest of my life.

I would also like to thank Dr. Jitender S. Deogun and Dr. Lisong Xu for serving on my committee. Their feedback and suggestions helped me improve this thesis. They also have given me valuable guidance throughout my academic career and advice to help me in my career.

I also sincerely thank my parents, Satya Priya Banerjee & Sipra Banerjee, and my dearest sister, Sreya Banerjee, for their love, patience and constant encouragement throughout my life and graduate studies.

My international education would not be complete without mentioning my wonderful friends in Lincoln, NE who have loved me and supported me during all my crests and troughs. They are my family away from my home and I have learned so much about life, priorities and career from each and everyone of them.

GRANT INFORMATION

This work was partially supported by NSF IIS-1116221.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A wireless sensor network (WSN) is a set of autonomous sensors which are spatially distributed to monitor physical or environmental conditions such as temperature, pressure, etc. The WSN can be built of a handful to a few thousands of nodes where each node is connected to one (or sometimes several) sensors. These sensors can vary widely in size as well. A WSN has several important applications. Area monitoring, air pollution monitoring, forest fire detection, landslide detection, water quality monitoring are some of them. Terrestrial WSN have been widely studied and numerous workshops and conferences are arranged each year for this active research area. At the same time, there is drive to develop underwater sensor networks to sense the underwater environment.

Water is crucial for supporting life on earth, so it is important to develop tools to monitor the water bodies. New technologies have enabled the exploration of the vast unexplored aquatic environment. This includes underwater modeling, mapping, and resource monitoring. One such example is the study of Chromophoric Dissolved Organic Matter (CDOM) which is the optically active component of the total dissolved organic matter in the oceans. An understanding of CDOM dynamics in coastal waters and its resulting distribution is important for remote sensing and for estimating light penetration in the ocean. However, the majority of exploration is currently done manually or by using expensive, large, and hard-to-maneuver underwater vehicles. As such, new solutions which consider the unique features of underwater environments are in high demand. In this thesis, we introduce a method of underwater monitoring using semi-mobile underwater sensor networks [20, 25]. We introduce three different algorithms that use a mobile underwater robot to perform the following functions:

- Collect data effectively from the underwater environment in presence of semi-mobile sensors; and
- Plan efficient paths for the mobile robot based on global, local and decentralized algorithms.

Underwater sensor network has some unique characteristics that differ from terrestrial WSNs such as:

- Large communication propagation delay,
- Low communication bandwidth,
- Limited node mobility,
- High error rate,
- Harsh underwater environment

Due to these, the existing solutions of terrestrial sensor networks cannot be applied directly to underwater sensor networks.

We introduce a method of underwater monitoring using semi-mobile underwater sensor networks and mobile underwater robots. The sensors are called AquaNodes and the underwater robot is called Amour. The AquaNode sensors are anchored at the bottom of the water column and floats mid-water column. The depth adjustment system within each sensor node allows the length of anchor line to alter in order to adjust its depth. These nodes are able to dynamically adjust their depths by using a decentralized, gradient-descent-based algorithm [20]. This dynamic depth adjustment algorithm runs online which enables the nodes to adapt to changing conditions (e.g., tidal front) and does not require a priori decisions about node placement in the water. In this work, we consider a two dimensional slice of the water and introduce a mobile underwater robot with two degrees of freedom. We describe setup and algorithms that determine path of a mobile underwater robot through a underwater sensor network. In particular, we develop three different algorithms for planning the path of a mobile underwater robot, traveling through the sensor field, in presence of the underwater sensor nodes.

The first algorithm, VoronoiPath, is a global path planning algorithm using the *Voronoi Tessellation* method. In the second approach, TanBugPath, we propose a local path planning algorithm inspired from the *Tangent Bug* method for obstacle avoidance. The third algorithm, AdaptivePath, is based on an adaptive decentralized control algorithm [20, 25], which plans the path of the mobile robot by determining the positions in the water column where the mobile robot should stop and gather information. These strategic sensing positions are referred to as *robot waypoints*. The sensors and robot waypoints are together referred to as *nodes*. This thesis extends the control algorithm to include the operation and path planning for a mobile robot and includes additional simulation and experimental results.

In case of VoronoiPath method, the system has global knowledge of the position of each sensor. Thus before the mobile robot enters the water, the algorithm notifies it of the positions where it will need to sense information. In the case of TanBugPath method, the robot does not know the position of any sensor except the first one when it enters the water column. The underwater robot finds its path to this sensor while maintaining a minimum distance so as to not cover overlapping regions. This sensor then transmits information about the location of the next sensor to the robot and the process continues in this manner. Finally, in the case of the adaptive decentralized algorithm, AdaptivePath, the sensors inform the underwater robot about the position of the next robot waypoint it should go to for sensing. A covariance function is needed for this algorithm. This covariance function describes the relationship between the sensors' positions and all the other points in the region of interest.

We have assumed a fixed covariance model for AdaptivePath algorithm. However, it can be iterated with different covariance models to capture dynamic phenomena. For example, if the water column has a specific region which is more interesting to study, the user can specify a different covariance function for that region and tell the underwater nodes to explore that region in greater detail. The decentralized controller determines the position of the nodes so that it is able to collect data that is reflective of the performance of the entire system and not just the particular positions where there are nodes. We have modeled the covariance as a multivariate Gaussian, as is often used in objective analysis in underwater environments [51]. The algorithm uses the covariance model in a decentralized gradient descent algorithm. We prove that the controller algorithm converges to a local minimum. While planning the path of the underwater robot, the algorithm also readjusts the position of the sensors to adapt to the mobile robot path and to provide better sensing of the region.

In the three algorithms mentioned above, we assume that an acoustic modem is used for communication between sensors and with the underwater robot. All the algorithms have low memory requirements and thus can run locally on the sensor network. We perform simulation experiments to examine and compare the different algorithms and then, show our results.

In [20, 25] where Detweiler *et al.* applied the original adaptive decentralized algorithm to study the problem of monitoring CDOM in the Neponset River. We can use the path planning algorithms to study the same phenomenon. This is one of the many practical applications where we will be testing the developed system in our future work.

1.1 Thesis Contributions

This thesis makes a number of contributions to the field of underwater sensor networks and robotics. Specifically, we:

- Present a new model of using underwater sensor network and an underwater mobile robot in parallel for effective network coverage and node placement.

- Introduce and compare three different path planning algorithms for the traversal of the underwater mobile robot.
- Develop a decentralized controller that creates a path for the mobile underwater robot and also optimizes the depths of the underwater sensors for energy efficient and effective data collection.
- Prove that the proposed controller converges.
- Extensively analyze the performance of the algorithms in simulation.

1.2 Thesis Outline

The rest of this thesis is organized as follows. First, we discuss the related work in Chapter 2. We next give a brief overview of the background work in Decentralized Depth Adjustment algorithm in Chapter 3. This is followed by Chapter 4, in which we introduce and analyze three different path planning algorithms for planning the path of the underwater mobile robot. Chapter 5 explores results of simulations to test the performance of the algorithm and explores the sensitivity to different parameters. Finally, we discuss future work and conclude in Chapter 6.

Chapter 2

Related Work

Studying the underwater phenomenon is an increasingly interesting area of research. Many scientists have pursued this research in many different ways. Some have concentrated on studying under the water by effective placement of sensor networks, while some study that by planning an effective path of a mobile robot through regions of interest. Our research focuses on both these areas and combines them together in this thesis. Along with these two important areas, some other interesting topics are path optimization, and papers dealing specifically with underwater sensors. We have presented the various important research in these fields in separate section. Since two of our algorithms deal with path planning with the help of *Voronoi Tessellation* and *Tangent Bug Algorithm*, we have presented two different sections that discuss relevant work on these topics which might not be directly related to underwater sensors.

2.1 Prior Work in Sensor Placement

Many research [35, 44] on sensor placement use submodular optimization to address problems based on searching. Most of them uses Gaussian Processes to model spatial phenomena. As such we have used a Gaussian Covariance function to model the phenomenon for our research. Some of the important research in sensor placement is presented in this section.

In [35], Guestrin *et al.* propose placing of sensors for monitoring Gaussian spatial phenomena based on maximizing the mutual information. They chose mutual information over entropy which is typically more popular and used in our research. The authors propose a polynomial-time approximation that is within $(1 - \frac{1}{e})$ of the optimum since finding the configuration that maximizes mutual information is NP-complete. The entropy and mutual information methods are compared based on root mean square error and log likelihood on temperature data and the authors claim that the mutual information method performs better than entropy. Then they demonstrate the approach on two real-world data sets.

In [44], Krause *et al.* review the recent work on optimizing observation in sensor network using several submodular functions. The authors present several submodular theorems and propose how they can be considered to solve different problems.

The recent availability of low-cost Unmanned aerial vehicles (UAVs) have made it possible for it to be used in a wide range of applications such as mapping, surveillance, search, and tracking. To leverage the capabilities of a team of UAVs, efficient methods of decentralized sensing and cooperative path planning are necessary. In [62], Tisdale *et al.* developed decentralized, autonomous control strategies that can account for a wide variety of sensing missions. In this paper, the goal is to use a team of unmanned vehicles to search for and localize a stationary target. The sensing system is vision-based. The system allows target search and localization in the same software framework. A general control framework was developed by posing path planning as a trajectory optimization problem. Path planning is accomplished in a receding-horizon framework; an objective function that captures information gain is optimized at each time step, over some planning horizon. The UAVs cooperate by exchanging predicted sensing actions between vehicles. The authors discuss many receding-horizon strategies that plan only a single step into the future and claim that for systems with constrained sensor footprints, multi-step planning is important and in some cases necessary for a good performance.

In [20, 25], Detweiler *et al.* present a adaptive decentralized controller that optimizes sensing by adjusting the depth of a network of underwater sensors. The authors prove that the controller converges to a local minimum. Extensive simulations and experiments are performed to verify the functionality of the system. The body of work presented in this thesis is an extension of this work. An extensive background on this research is presented in Chapter 3.

In [39], Julian *et al.* present an entropy based approach to control robots equipped with sensors to improve the quality of sensing. The robots move following the gradient of mutual information. The performance of the system is demonstrated in a five quad-rotor flying robot experiment and 100 robot numerical simulation. This is similar to our thesis as a few robotic sensors are distributed autonomously to maximize the sensing. However, in this paper the authors assume a non-parametric system but we assume a Gaussian process for our system.

In [38] the authors present theorems and algorithms for using many collaborating robots equipped with sensors to acquire information from a large-scale environment. The authors assume a non-parametric distribution of data and achieve decentralized control by using a consensus-based algorithm which was specifically designed to approximate the required global quantities like mutual information gradient and sequential Bayesian filter with local estimates.

Finally they combine the work presented in [39] and [38] in [40] and then develop a fully decentralized system. They also carry out further experiments to test their system on a small-scale indoor experiment and a large-scale outdoor experiment using five quadrotor flying robots and then use the developed inference and coordination software to simulate a system of 100 robots.

2.2 Prior Work in Path Planning

Often only one or a few mobile robots need to gather information from a large body of water. In such cases we need to plan their trajectory depending on various constraints such as presence of obstacles on the path or the energy capability of the robot itself, etc. Several approaches have been developed for addressing these problems, however, they have limitations like discretization of state, efficiency vs. accuracy trade offs, or the difficulty of adding interleaved execution. These existing methods are successful in planning path for the robot to move from an initial to a final position by a minimum distance or pertaining to some other optimizing constraint. But most of them do not focus on mobile robots whose primary objective is to gather information with an on-board sensor from different target points. In our thesis we present a system which combines a semi-mobile sensor network and a mobile underwater robot which together focus on gathering maximum information from the entire region of interest. In this section we present some prior research in path planning.

One of the earliest problems regarding path planning is discussed by Brooks *et al.* in [4] where the main focus is on a good representation of free space. The authors present a fast algorithm to find good collision-free paths for convex polygonal bodies through space littered with polygonal obstacle. The algorithm is based on characterizing the volume swept by a body as it translates and rotates as a generalized cone. Then it determines under the conditions in which one generalized cone is a subset of another. An important feature of this work is that the paths found out by the algorithm tends to be equidistant from all objects, thus it does not lead to any failure of mechanical devices in the robot due to mechanical imperfections. The major drawback of the algorithm presented in this paper is that it typically does not work well in tightly constrained spaces as there are insufficient generalized cones to provide a rich choice of paths.

A large portion of research on path planning is focused on how a robotic manipulator can reach a certain goal point while avoiding other static obstacles in its path. In [34], Gilbert *et al.* describe an approach where an obstacle is avoided in terms of the mathematical properties of the distance functions between potentially colliding parts. The authors then apply the numerical methods on a three-degree-of-freedom Cartesian manipulator.

Some researchers have explored the problem of path planning for autonomous robots in the presence of mobile obstacles. In [29], Fujimura *et al.* use time as one of the dimensions of the model world due to which the moving obstacles can be regarded as being stationary in the extended world. The obstacles are represented by quadtree-type hierarchical structure. According to the authors, speed, acceleration, and centrifugal force are the three most essential factors in navigation. If the the robot does not collide with any other moving obstacles and is able to navigate without exceeding the predetermined range of velocity, acceleration, and centrifugal force, a solution is feasible. The major drawback of this paper is that the performance suffers if the search space size increases or if the number of obstacles in space increases.

A number of the earlier path planning problems deal with how robotic manipulators can operate in an environment with static obstacles. For example, in [42], Kavraki *et al.* propose a two stage algorithm for path planning for robots with many Degrees of Freedom (DOF). The first or preprocessing stage takes place only once for a given environment and in this stage the algorithm generates a network of random collision-free configurations. In the next planning stage, the algorithm connects any given initial and final configurations of the robot to two nodes of the network and then computes a path through the network between these two nodes. The preprocessing stage takes a large amount of time but the planning stage is extremely fast. This approach is specially suitable for many-DOF robots which have to perform many successive point-to-point motions in the same environment. The authors implement the method with many 6 to 10 DOF robots and then analyze their performance.

Many papers on robotic path planning deal with how a single autonomous robot will navigate through a cluttered environment. In [61], Thrun *et al.* integrate the grid-based and topological paradigms. Topological maps are generated on top of the grid-based maps which are learned using artificial neural networks and Bayesian integration which is then used to autonomously operate a mobile robot equipped with sonar sensors in populated multi-room environments.

For unmanned missions in the real world, longer-range path planning is required. In most path planning algorithms it is assumed that all known, a priori information is correct and the environment is fully known. However, in real word, the system should be able to automatically plan a path from the vehicle's current position to its goal position, using only partial information about the environment. In [58], Stentz introduced a D* path planning algorithm for generating optimal paths for a robot operating with a sensor and a map of the environment. The map can be complete, empty, or contain partial information about the environment. For regions of the environment that are unknown, the map may contain approximate information, stochastic models for occupancy, or even a heuristic estimates. The method is the used to plan a path for Unmanned Ground Vehicles (UGVs).

A sizable amount of work has been done in path planning with the help of the *Potential Field Method*. For these method, global knowledge of the system is needed and mostly static obstacles are assumed. Often, the objective is to go from one point to other in the shortest time or by covering the shortest distance. No emphasis is given on the physical properties of the system, so the characteristic of the system does not govern the path. Some of the prior research is discussed in this section.

Warren *et al.* discuss planning path of robotic manipulators or mobile robots around stationary obstacles in [68]. Potential Field Method is used as it is relatively fast. A trial path is chosen and then improved under the influence of a potential field method as it helps to avoid a scenario in which the robot gets stuck in a local minima. The drawback to this method is that the global workspace should be known at the time of the planning.

Some research on path planning with *Potential Field Method* combine both the global and local path planning together to achieve best results. For example, in [37], Hwang *et al.* define a two level path planner where a potential function similar to the electrostatic potential is assigned to each obstacle and free space is determined in terms of minimum potential valleys. In the first stage, a global planner selects the path of the robot from the minimum potential valleys and its orientations along the path that minimize a heuristic estimate of the path length and the chance of collision. In the next stage, a local planner modifies the orientations of the robot and the local path and to derive the final collision-free path. If the local planner fails at any stage, a new path and orientations of the robot are selected by the global planner and then fed to the local planner for estimation. This process is continued until a solution is found or until there are no paths left to be examined. The authors claim that this algorithm is capable of solving a large set of problems in much shorter time than exact algorithms. The drawback of this paper is that it considers a point robot.

Some papers place emphasis on local path planning because global path planning, as discussed before, can often be impractical for the given situation and at the same time computationally and time-wise more expensive. In [2], Barraquand *et al.* present a collection of numerical potential field techniques for robot path planning which constructs a good potential field and effectively escapes their local minima. All of them apply the same approach of constructing a potential field over the configuration space of the mobile robot, then builds a graph connecting the local minima of this potential, and then searches this graph. The graph is built incrementally and searched as it is built. The authors propose four different techniques for constructing the local minima graph and then study the difference. Among the four techniques, the random motion technique has the best combination of time efficiency, generality and reliability. Also it is highly *parallelizable*. The authors claimed that the planner implementing these techniques was able to solve path planning problems, whose complexity (measured by the number of DOFs or the number of obstacles) is far beyond the capabilities of previously implemented planners and faster than most previous planners for simpler problems.

When path planning problems are solved with *Potential Field Method*, many times there are obstacles near the goal position and hence the mobile robots do not reach the goal. To address that problem, in [31], Ge *et al.* introduce repulsive potential functions that take the relative distance between the robot and the goal into consideration, ensuring the goal position is the global minimum of the total potential so that the robot can reach the goal while avoiding collision with obstacles. Then they use their method to solve the GNRON problem in simulation.

There are various off-beat papers which add to the family of artificial-potential-based path-planning methods. For example, in [66], the potential field is motivated by steady-state heat transfer. The authors, Wang *et al.,* define obstacles and free-space in terms of variable thermal conductivity. Thus the optimal path planning problem is reduce to the problem of heat flow in the direction of minimal thermal resistance. The advantages of this technique is that complex obstacles can be represented in a simple geometrical domain and it can handle changes in the environment. The authors propose a method for path planning of non-spherical robots, by reducing the problem into a sequential translation-rotation search.

One of the major drawbacks of path planning with the help of *Potential Field Method* is that the robot can get stuck in a local minima. Some researchers have studied different methods so that the robot can escape the local minima if they are stuck. Yun *et al.,* in [71], describe an algorithm which switches between two control modes - the overall algorithm follows the potential field guided control mode. However, when the robot falls into a local minimum the new algorithm switches to a wall-following control mode. It then switches back to the potential field guided control mode when a certain condition is met. The distance from the robots current position to the goal position is used to determine if the robot is caught in a local minima. The algorithm is implemented on a Nomad 200 mobile robot. And simulation and experimental results are presented to show that the algorithm is effective in escaping local minima in complex environments.

Many researchers have combined the *Potential Field Method* of solving path planning problem with other methods. In [18], Connolly *et al.* propose a method for planning smooth robot paths using of *Laplaces Equation* so that the functions will prevent the spontaneous creation of local minima over regions of the configuration space of the robot. The advantages of this method are that once the function is computed, paths can be solved very quickly and is well suited for running on massively parallel architectures. However, the process of finding the function is slow, hence, more suitable for parallelization.

In [73], Zhao *et al.* present a method for autonomous navigation of mobile robots using artificial potential field. It is shown that the controller can keep the robot away from obstacles, and can escape from the local minima. Simulation are performed which show that the method used has small memory requirements and there is no need for preprocessing. The algorithm also prevents the robot from falling into complicated environment where it can get trapped in a local minima.

As we can see most of these focus on finding a path from the source to the goal point for the mobile robot and differs from our work which is focused on decentralized control of the mobile robot to gather maximum information from different target locations depending on local knowledge.

Bruce *et al.,* in [5], develop a robot control system that uses Rapidly-Exploring Random Trees (RRTs) path planner that combines path planning and execution called ERRT (execution extended RRT). The authors introduced two extensions of previous work on RRTs, the waypoint cache and adaptive cost penalty search, which are shown to improve replanning efficiency and the quality of generated paths. Then the ERRT is applied to a real-time multi-robot system and the results are shown it performs more efficiently for replanning than a basic RRT planner.

Since most sensor-based path planning algorithms are evaluated by the length of the path from the source to the goal, many are evaluated based on the worst path length and finding out measures to shorten that. In [53], Noborio *et al.* argue that shortening average path length is more important than shortening the worst path length in the practical use and then present one such algorithm. The authors also compare all the sensor-based path-planning algorithms with respect to average path length.

A number of research papers on path planning can be categorized as Simultaneous Localization and Mapping (SLAM) where the robot does not have any information about the global environment but traverses along the environment while at the same time it is mapping it. In [43], Kollar *et al.* present a information-theoretic approach for representing the frames in the environment as a constrained optimization problem. In this algorithm, the authors converted the current environmental map to a graph of the map skeleton.

Sensing constraints are placed at the boundaries and frontiers of the environment which are denoted as the graph nodes. Then the algorithm searches for a minimum entropy tour through the graph. The authors describe that a specific factorization of the map covariance allows the Extended kalman Filter (EKF) updates to be reused during the optimization which gives an efficient gradient ascent search for the maximum information gain path through the environment. Finally, a learner is introduced which optimizes the local trajectory of the robot to predict a global path that results in a high quality map.

Often research in robot path planning for exploration and mapping has focused on sampling the hotspot fields of the environment. In [72], authors Zhang *et al.* present a method which is an information roadmap deployment (IRD) approach that combines information theory with probabilistic roadmap methods. The information roadmap is sampled from a normalized information theoretic function that favors samples with a high expected value of information in configuration space. The method is implemented in a simulated de-mining system to plan the path of a robotic ground-penetrating radar, based on prior remote measurements and other geo-spatial data. The simulations show that under a wide range of workspace conditions and geometric characteristics the system performs more efficiently when IRD is used compared to complete coverage and random search.

Low *et al.*, in [49], formalized the task of exploration in a sequential decision-theoretic planning under uncertainty framework called MASP for multi-robot systems. The time complexity of solving MASP depends on the map resolution, which limits its use in large-scale, high-resolution exploration and mapping. In [50], the authors extend their work and present an information-theoretic approach to MASP (iMASP) for efficient adaptive path planning for active exploration and mapping of hotspot fields. They reformulate the cost-minimizing iMASP as a reward-maximizing problem and show, both theoretically and empirically, that the time complexity becomes independent of map resolution and is less sensitive to increasing robot team size. The advantage of this method is that it is useful in large-scale, high-resolution exploration and mapping. The authors claim that the proposed approximation techniques can be generalized to solve iMASPs that utilize the full joint action space of the robot team, and thus it will allow the robots to move simultaneously at every stage.

A number of researchers have solved the problem of path planning for mobile robots using heuristic measurements for point robots. In [11, 12], Choi *et al.* plan continuous paths for mobile sensors to improve long-term information forecast performance. The environment of the mobile robot is represented as a linear time-varying system and the information gain is defined by the mutual information between the continuous measurement path and the future verification variables. Spatial interpolation is used for path representation and planning. Two different expressions for computing the information gain - the filter form and the smoother form - are compared. The smoother form is reported to be preferable. The proposed theoretical frameworks are tested on a numerical example for the simplified weather forecast. The key contribution of this work is to provide a framework for quantifying the information obtained by a continuous measurement path to reduce the uncertainty in the long-term forecast for a subset of state variables which differs from the work presented in this thesis in that it focuses on reducing uncertainty in current ongoing measurements.

Some mobile robots have constraints like a bounded field of view (FOV). Not many existing path planning techniques can be applied to find a trajectory for such robots. In [8], Cai *et al.* developed a methodology for planning the sensing strategy of a robotic sensor with a bounded FOV deployed for the purpose of classifying multiple fixed targets located in an obstacle-populated workspace. In this paper, obstacles, targets, sensors platform, and FOV are represented as closed and bounded subsets of an Euclidean workspace giving an unique cell decomposition. A connectivity graph is constructed with observation cells and then it is pruned and transformed into a decision tree that is used to compute an optimal sensing strategy, including the sensors motion, mode, and measurement sequence. The method is demonstrated through a mine-hunting application. The authors then perform numerical experiments which show that these strategies outperform shortest path, complete coverage, random, and grid search strategies, and are applicable to non-overpass capable robots that must avoid targets as well as obstacles.

2.3 Prior Work in Path Optimization

Another important criteria for path planning for mobile robot is to optimize the path based on some criteria like time taken, energy consumed, or information gained etc. Our algorithm is optimized for the dual parameters of maximizing information gain and minimizing the distance traveled. In this section we present some of the prior work in optimizing the path based on different criteria.

In [48], Liu *et al.* introduced a method of information-directed routing in which routing is formulated as a joint optimization of data transport and information aggregation. The routing objective is to minimize communication cost while maximizing information gain. In this paper, possible moving signal sources are located and tracked as an example of information generation processes. Two common information extraction patterns are considered - routing a user query from an arbitrary entry node to the vicinity of signal sources and back, or to a pre-specified exit node. The goal is to maximize the information accumulated along the path. The simulations performed with the proposed algorithm demonstrated that information-directed routing is a significant improvement over a previously reported greedy algorithm, as measured by sensing quality such as localization and tracking accuracy and communication quality such as success rate in routing around sensor holes.

Celeste *et al.,* in [10], presented a framework to solve the the problem of planning the path of an intelligent mobile robot in a real world environment described by a map composed of features representing natural landmarks in the environment. The vehicle is equipped with a sensor which allows it to obtain range and bearing measurements from observed landmarks during the execution. The problem was discretized and a Markov Decision Process with constraints on the mobile maneuver was used. Functionals of the Posterior Cramẽr-Rao Bound is used as the criterion of performance of the optimal trajectory planner and a Cross Entropy algorithm is used to solve the optimization.

In [59], Stranders *et al.* presented an on-line, decentralized coordination algorithm for monitoring and predicting the state of spatial phenomena by a team of mobile sensors. Since the sensors are applied for disaster response, there is strict time constraint which prohibits path planning in advance. In this algorithm, the sensors coordinate their movements with their direct neighbors to maximize the collective information gain, while predicting measurements at unobserved locations using a Gaussian process. The authors show how the max-sum message passing algorithm can be applied to this domain in order to coordinate the motion paths of the sensors along which the most informative samples are gathered. It presents two new generic pruning techniques that result in speed-up of up to 92% for 5 sensors. The proposed algorithm is evaluated empirically against several on-line adaptive coordination mechanisms, and up to 50% reduction in root mean squared error is reported compared to a greedy strategy.

As we can see, optimization in most of the research presented here is based on one major or two somewhat related parameters. But in real world, we might need to optimize a system based on two opposing parameters. In our research we present such a system and evaluate the different system parameters that affect the overall sensing of the environment.

2.4 Prior Work in Underwater Sensors

Since our research discusses the path optimization for underwater mobile robot in presence of an existing semi-mobile sensor network, another important line of research is the prior work that has been carried out in the domain of underwater sensor networks. In this section we present some of these research relevant to our thesis.

In [47], Leonard *et al.* design an effective and reliable mobile sensor network for collecting the richest data set in an uncertain environment given limited resources. Their main focus is on designing mobile sampling network to take measurements of scalar or vector fields and collect optimal data using Autonomous Underwater vehicles (AUV) for sensing. In response to measurements of their own state and measurements of the sampled environment, AUV can control their own motion using feedback control. This reactive approach to data gathering is adaptive sampling. Even though they use feedback control for their robots, which is of a different type than ours, they use a similar covariance model.

Smith *et al.* present a combination of two algorithms in [57] for monitoring under water with sensors and use it to study the occurrence and life-cycle of harmful algal blooms in ocean. The first algorithm finds a closed path which passes through regions of high sensory interest while avoiding areas that have large magnitude or highly variable ocean currents. Along this path, the second algorithm sets the pitch angle at which the glider moves along the path to ensure higher sample density is achieved in areas of higher scientific interest. These two algorithms are combined into a single, iterative low cost algorithm with the output of the path planning algorithm being used as the input for the angle optimization algorithm. These strategies are implemented on an autonomous underwater glider which goes through a region of interest. This is similar to our algorithm if we use just the mobile underwater robot. Also they focus mainly on a planar region of interest whereas our mobile robot mainly focuses on the depth of the water column.

There are a number of papers which deal with path planning for underwater sensors. For example, in [54], Petillot *et al.* describe a general framework for performing 2-D obstacle avoidance and path planning for underwater vehicles based on a multi-beam forward looking sonar sensor. There are two phases - planning and tracking. The feature extraction is performed on real-time data and consecutive frames are studied to obtain the dynamic characteristics of the obstacles. A representation of the vehicles' workspace of the obstacle is created based on these features which is a convex set of obstacles defining the workspace. Then a sequential quadratic programming, which is a non-linear search algorithm, is employed, where obstacles are expressed as constraints in the search space for obstacle avoidance and path planning in complex environments which include fast moving obstacles. The authors then show the results obtained on real sonar data. Compared to other methods, this system generates very smooth paths, can handle complex and changing workspaces and presents no local minima as they are using a convex representation for the obstacles.

An Autonomous Underwater Vehicles (AUV) is a robot which travels underwater without requiring input from an operator. The path planning for AUV is necessary as unforeseen events may violate constraints of a previously planned path and it may need to plan a new path subject to additional constraints. In [9], Carroll *et al.* proposes a suitable path planning algorithm for AUV that maintains a number of databases to facilitate the planning process for the Autonomous Underwater Vehicle Controller Project at Texas A&M University. An A^* algorithm is used to generate path. The path planner described in this paper tries to find a 3D corridor that does not intersect any non-entry zones. Additional factors affecting performance of the algorithm are evaluated and discussed in this paper.

Often researchers have to send an AUV multiple times to collect information. If there is a way to improve the quality of the path based on the information gained while the AUV traveled on an old path or if a path in a new region can be determined from the path followed by the AUV in a different but similar region, that might be useful to researchers. In [65], Vasudevan *et al.* propose a case-based path planning scheme in which the algorithm relies on past experience to solve new problems or generates a new solution by retrieving and adapting an old one which approximately matches the current situation. In this paper, the authors describe how the environment, including past routes and objects, and case frames of past route planning scenarios can be represented in the navigational space and then represent the navigational environment using an annotated map system which are useful in retrieving and adapting them to a new route. Whenever a matching route is not available, a new route is synthesized by the planner relying on past cases that describe similar navigational environments.

Researchers have studied the path planning for AUV under the circumstance where there are obstacles in the water column. In [67], the author, Warren, develops an artificial potential field technique for planning the path of an AUV. This method is less susceptible to local minima than other potential field methods. A trial path is chosen first. Then potential fields are applied around obstacles. The trial path is then modified under the influence of the potential field until an appropriate path is found. One disadvantage of this methods is that most of the global workspace should be known at the time of the planning to figure out the trial path. However, this method has the provision that when new obstacles are discovered, they can be included and a new route planned based on the updated obstacle field.

In the ocean it is not easy to predict what locations will be more useful to collect data from. Qualitatively, the region is defined by an objective function and the goal is then to optimize this objective under the constraints of available observing network. This is called adaptive sampling of the ocean. Yilmaz *et al.,* in [70], propose a new path planning scheme for adaptive sampling based on mixed integer linear programming (MILP), which is capable of handling multiple-vehicle and multiple-day cases. The mathematical goal is to plan the path that will maximize the line integral of the uncertainty of field estimates as sampling this path can improve the accuracy of the field estimates the most. The authors take into account several constraints while addressing the issue like motion constraints, vicinity constraint for multiple vehicles, communication constraint, obstacle avoidance etc. Implementation platform is XPress-MP optimization package from Dash Optimization. The problem is an NP-hard problem, therefore, as the size of the region increases, the solution time increases exponentially.

In [55], the authors Petres *et al.* introduce a novel Fast Marching (FM)-based approach that is designed to efficiently deal with wide continuous environments prone to currents under water. There are four steps: First, the authors develop an algorithm (FM^*) to extract a 2-D continuous path from a discrete representation of the environment; in the next step, a practical implementation of anisotropic FM is used so that the underwater currents adapted to the path-planning method; thirdly, the vehicle kinematics is taken into account for both isotropic and anisotropic media; finally, a multi-resolution method is introduced to speed up the overall path-planning process. The main drawback of this method is data reduction can produce a loss of information which in turn can affect the optimality of the resulting path. Also, the authors assumed only static environment and, so, can be defined a priori in a cost function.

In all these research the focus is on either avoiding obstacles to gain maximum information from a region or on traversing the water to gain maximum information. However, none of these researcher discuss how to plan the path of mobile robot when there are semi-mobile sensors already deployed in the region of interest. Our method specifically deals with this problem.

2.5 Prior Work in Path Planning with Voronoi Method

A number of researchers have used the Voronoi Tessellation method for planning the path of a robot. In our thesis we use an implementation of the Voronoi Tessellation method for global path planning of the mobile robot through an environment scattered with semi-mobile sensors. Hence, in this section, we present a short overview of the relevant prior research in this area.

Takahashi *et al.*, in [60], introduce a path planning algorithm based on *Generalized Voronoi Diagram (GVD)* for a rectangular object in a planar workspace populated with polygonal obstacles. After generating the GVD of the free space, it is converted into a equivalent graph with nodes and arcs. Then, graph theory algorithms are applied to find an optimal path from the source to the destination. In addition, they examined four heuristic techniques for planing the motion of the rectangular object through the workspace. This algorithm is fast, creates shorter path lengths, and can be applied to cluttered workspaces. This is similar to our research as it uses a similar type of algorithm. However, the criterion used to search the graph in this paper is simply the shortest path satisfying a minimum radius threshold whereas in our algorithm we determine the shortest path using *Djikstra's algorithm*.

Several other path planning algorithms have been proposed based on the *generalized Voronoi diagrams*. Choset *et al.*, in [14, 15, 13, 16] present the underlying idea that the boundaries of Voronoi diagrams can be used to calculate paths that have maximum clearance between the boundaries of the robot and the obstacles which we use in our research to represent the most efficient robot path.

The *Hierarchical Generalized Voronoi Graph* (HGVG) is a method in which the algorithm uses distance information to incrementally construct the HGVG edges. The numerical procedure uses raw sensor data to generate a small portion of an GVG edge. The robot then moves along this portion and the procedure is repeated to generate the next segment. Therefore HGVG interleaves sensing with motion [17].

In [36], Hoff *et al.* present several techniques that exploit the fast computation of a generalized Voronoi Diagram using graphics hardware for motion planning in three-dimensional workspace for rigid robots. This paper gives similar arguments for using Voronoi Diagram for path planning just like us. However, the path planning is done in a three dimensions compared to two dimensions in our case, and as such requires the use of graphics HD drive.

Some authors have applied the Voronoi method of path planning for non-holonomic robots. In [52], Nagatani *et al.* proposed a local path planning method for car-like mobile robot based on *Generalized Voronoi Graph (GVG)*.

In [30], a linear time two-step method for global path planning for a robot is described by Garrido *et al.* Here, at first the safest areas of environment are extracted by means of a *Voronoi* diagram and then in the second step a *Fast Marching Method* is applied to the extracted areas to obtain the shortest path. The advantages are speed, easy implementation and creates smooth trajectories that increases the quality (reliability). An interesting features is that the proposed algorithm dilates the robots and obstacles to make the path secure and to avoid collision. An interesting similarity between this algorithm with the one presented by us in this thesis is that the objects and walls are dilated in a security distance that ensures that the robot neither collides with obstacles and walls nor accepts passages narrower than the robot size.

Bhattacharya *et al.*, in [3], present an algorithm based on the Voronoi diagram for the calculation of optimal path between the source and the destination. The path obtained directly from the Voronoi diagram may not be always optimal. At regions where the obstacles are far apart, there may be many unnecessary turns which increases the path length. So, in this algorithm, users can specify the clearance between the robot path and the obstacles. Depending on this value a path is constructed that is a close approximation of the shortest path satisfying the required clearance value set by the user.

2.6 Prior Work in Path Planning with TanBug Method

We present a local path planning algorithm which uses a variation of the Tangent Bug algorithm. So in this section we present some prior research work which helped us formulate our algorithm. In [41], Kamon *et al.* introduce the *TangentBug* algorithm which is a range sensor based navigation algorithm for autonomous robots with two degrees of freedom. This algorithm is discussed in detail in Chapter 4.5. In brief, the *TangentBug* algorithm uses the range sensor data to compute a locally shortest path based on the local tangent graph, or LTG and from there chooses the locally optimal direction to move towards the target. This algorithm performs better than other classical bug algorithms. The advantage of this algorithm is that it requires global position information only for the detection of the target and for looping around an obstacle boundary.

Most of the prior work in robot path planning assumes indoor environments and omnidirectional motion and sensing for robots. However, in [46], Laubach *et al.* remove these non-realistic assumptions and implement the algorithm on the *Rocky 7 Mars Rover prototype*.

In real life, the autonomous robot have less than ideal sensing due to limitations in the range in which it can sense its surroundings. Therefore, the path planning of a robot cannot be based on complete global knowledge of the environment. Thus the robots need to continuously sense its environment using its on-board sensors in order to navigate the environment. In [33, 32], Ge *et al.* introduced concept of an *Instant Goal* approach which is a local target specially designed for action planning based on sensory input. The algorithm, like the Bug algorithms, combines the global information and local sensor data. The main contribution of this paper is integration of Collision Avoidance With Boundary

Following. Also the authors present a vector representation of the local environment in the paper. This paper helped us formulate the concept of instant goal, updated on each iteration, which is applicable to our system. However unlike this paper, our research does not incorporate the concept of collision avoidance.

Tangent Bug is one of the most frequently used algorithm for obstacle avoidance for sensor based mobile robots. In presence of static obstacles, the Tangent Bug algorithm can be used to plan a path from the start to the goal. In [69], the author *Wei* proposed a new complete sensor based path planning algorithm for autonomous mobile robot that uses a smooth version of the TangentBug algorithm. The advantages are that the algorithm has global convergence to the target and at the same time generates a smooth path.

In [7], Buniyamin *et al.* present an overview of the path planning algorithms for autonomous robots focusing mainly on the bug algorithm family. In addition, they introduce a new algorithm called the *PointBug* which minimizes the use of the outer perimeter of an object, which results in a shorter path length. Finally, this new algorithm is compared with existing ones. The main feature of this algorithm is that it can operate in a dynamic environment as it requires minimal amount of prior information. However, the authors assume that the point robot is equipped with ideal sensing (an infinite range sensor, odometer, and digital compass with ideal positioning.) which is rarely possible in a real-life environment. This research is similar to ours as it emphasizes planning with only local knowledge and the type of sensing. However, instead of obstacles in the region of interest we have static sensors with given sensing radius.

2.7 Summary

Increasingly, scientists have used static sensors and robots to study various underwater phenomenons. However, until now, most of this work mainly focused on planning path for mobile underwater AUVs or deploying sensors underwater effectively to gain maximum information. None of these work have combined the underwater sensor and robots together to effectively scan the environment to increase the information gain. The current work aims at demonstrating the possibility that using sensors and robots together can be used to gain more target specific information from a underwater region. The related work addresses challenges in sensor placement, path planning, path optimization etc. while working with underwater sensors and robots. In the following chapters we present our system and algorithms in detail and discuss how they fill the gaps present in current systems.

Chapter 3

Decentralized Depth Adjustment : Background

3.1 Introduction

In this chapter we introduce the background work on Decentralized Depth Adjustment algorithm by Detweiler *et al.* in [20, 25]. This information is important in the upcoming chapters. In Section 3.2 we briefly describe the hardware platform that will be used for the experiments. In Section 3.3 we introduce the theory, software and algorithm that will be used on the hardware platform described. Chapter 4.4 proposes the mobile underwater robot to increase the efficiency of sensing over the whole region of interest.

3.2 Underwater Sensor Network and Robot Platform

Detweiler *et al.* developed an inexpensive underwater sensor network system that incorporates the ability to dynamically adjust its depth. The base sensor node hardware is called the AquaNode platform and is described in detail in [21, 26, 64]. In [22], they have extended this basic underwater sensor network with autonomous depth adjustment ability and created a five node sensor network system, whose nodes move up and down in the water column under their own control. In addition, they have developed an underwater robot, called Amour, that can interact and acoustically communicate with the underwater sensor network [24, 27, 63]. Here we will briefly summarize the system.

Figure 3.1 shows a picture of the AquaNodes with the depth adjustment hardware and the underwater robot Amour. Both the underwater sensors and the robot have built-in pressure and temperature sensors and inputs for a variety of analog and digital sensors.

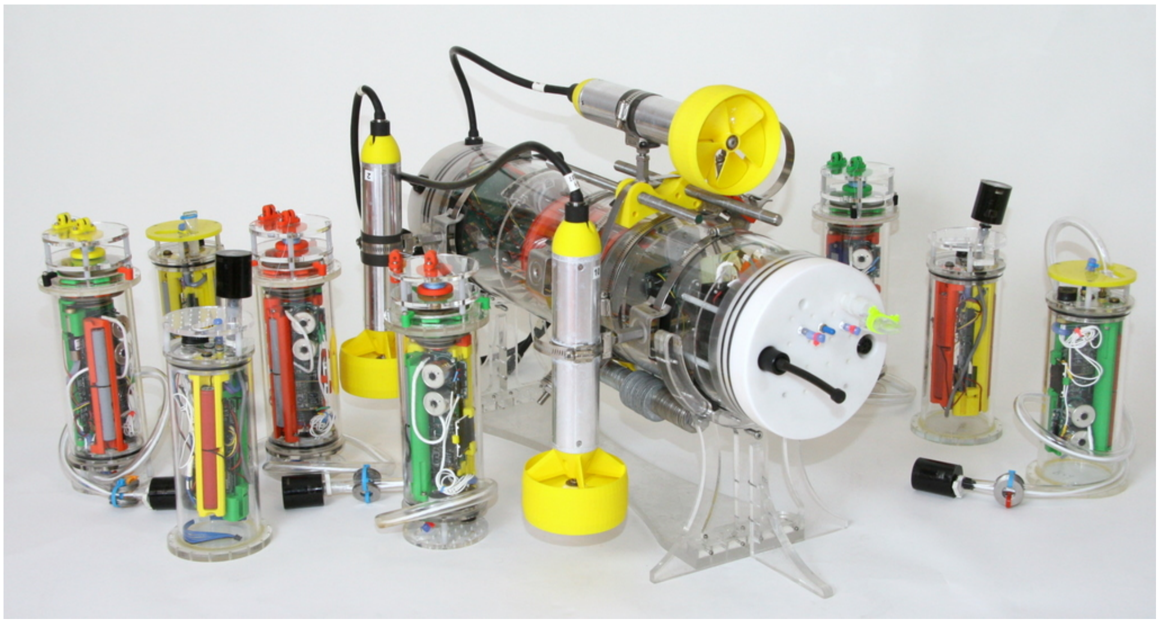


Figure 3.1: Depth adjustment system with AquaNode and underwater robot Amour [19].

The communication system used for communication between sensor nodes and the underwater robot is a custom developed 10 W acoustic modem [26]. The modem uses a frequency-shift keying (FSK) modulation with a 30 KHz carrier frequency and has a physical layer baud rate of 300 b/s. The acoustic modems are also able to measure distances between pairs of nodes. In previous work, it has been demonstrated how this can be used to perform relative localization between the sensor nodes and provide localization information to underwater robots [23]. We can use this capability to determine the positions of the nodes in our experiments and guide the underwater robot.

The AquaNodes is anchored at the bottom and floats mid-water column. The depth adjustment system allows the length of anchor line to be altered to adjust the depth in the water. The AquaNodes is cylindrically shaped with a diameter of 8.9 cm and a length of 25.4 cm without the winch mechanism and 30.5 cm with the winch attached. It weighs 1.8 kg and is 200 g buoyant with the depth adjustment system attached. The depth adjustment system allows the AquaNodes to adjust their depth (up to approximately 50 m) at a speed of up to 0.5 m/s and use approximately 1 W when in motion. With frequent motion and near continuous depth adjustment the nodes have power (60 Wh) for up to two days. In low power modes it can be deployed for about a year, but typical deployments are in the order of weeks. Additional details on the AquaNodes hardware can be found in [22].

3.3 Background in Decentralized Depth Adjustment

In this section we briefly discus the general decentralized controller, describe the Gaussian covariance function to be used, and define the controller in terms of the covariance function. This controller converges to a local minima [20, 25]. In Chapter 4 we extend this algorithm to encompass underwater mobile robots introduced to the system.

3.3.1 Assumptions

The decentralized depth control algorithm we develop in this chapter makes some assumptions about the system. These are:

- The nodes know their locations.
- The nodes can communicate with each other.
- The nodes can adjust their depths.
- The nodes know the covariance function.

3.3.2 Problem Formulation

Given N sensors at locations $\{p_1 \cdots p_N\}$, we want to optimize their positions for providing the most information about the change in the values of all other positions $q \in Q$, where Q is the set of all points in our region of interest. In this case, the underwater sensor nodes are constrained to move in one dimension along z-axis with fixed x, y axes.

The best positions to place the sensors are positions that tell us the most about other locations. If we have one sensor at position p_1, and one point of interest q_1, then we want to place p_1 at the location which is closest to q_1 because any changes in the sensory value at q_1 will be highly correlated to the changes that we measure at p_1. The covariance function captures the essence of this correlation. So, a sensor should be placed at the point of maximum covariance with the point of interest. Or more formally, sensor should be placed at position p_1 such that $Cov(p_1, q_1)$ is maximized.

More generally, if we have M points of interest in the region Q, to maximize the covariance between the point of interest q_j and all sensed points p_i by moving all p_i to maximize:

$$\arg\max_{p_i} \sum_{j=1}^{M} \sum_{i=1}^{N} Cov(p_i, q_j) \tag{3.1}$$

However, this objective function has the problem that some areas may be covered well while others might not be covered. To prevent that, we need to ensure the objective function penalizes regions that are already covered by other nodes. This is achieved by modifying the objective function to minimize:

$$\arg\min_{p_i} \sum_{j=1}^{M} \left(\sum_{i=1}^{N} Cov(p_i, q_j) \right)^{-1} \tag{3.2}$$

Instead of maximizing the double sum of the covariance, this objective function minimizes the sum of the inverse of the sum of covariance. This reduces the increase in the sensing quality achieved when additional nodes move to cover an already covered region.

For sensing every point in the region, we modify the objective function to integrate over all points q in the region Q of interest:

$$\int_Q \left(\sum_{i=1}^{N} Cov(p_i, q) \right)^{-1} dq \tag{3.3}$$

3.3.3 Objective Function

The objective function, $g(q, p_1, ..., p_N)$, is the cost of sensing at point q given sensors placed at positions $\{p_1, ..., p_N\}$. For N sensors, we define the sensing cost at a point q as:

$$g(q, p_1, ..., p_N) = \left(\sum_{i=1}^{N} f(p_i, q) \right)^{-1} \tag{3.4}$$

This is the inside of Equation 3.3, when $f(p_i, q) = Cov(p_i, q)$.

Integrating the objective function over the region of interest Q gives the total cost function. We call this function $\mathcal{H}(p_1, ..., p_N)$ and formally define it as:

$$\mathcal{H}(p_1, ..., p_N) = \int_Q g(q, p_1, ..., p_N)\, dq + \sum_{i=1}^{N} \phi(p_i) \tag{3.5}$$

The sum over the function $\phi(p_i)$ is a term added to prevent sensors from trying to move outside of the water column. This restriction on the node's movement is needed for the algorithm to converge. However, we can avoid this term in further discussions for simplicity of notation as this has little impact on the results.

3.3.4 General Decentralized Controller

A decentralized control algorithm [20, 25] is derived from the given objective function in Equation 3.5, that moves all nodes to optimal locations by making use of local information only. This is achieved by minimizing $\mathcal{H}(p_1, ..., p_N)$, which will be henceforth referred to as $\mathcal{H}$.

The gradient of $\mathcal{H}$ with respect to each of the z_is is calculated:

$$\frac{\partial \mathcal{H}}{\partial z_i} = \frac{\partial}{\partial z_i} \int_Q g(q, p_1, ..., p_N)\, dq \tag{3.6}$$

$$= \int_Q - \left(\sum_{j=1}^{N} f(p_j, q) \right)^{-2} \frac{\partial}{\partial z_i} f(p_i, q)\, dq \tag{3.7}$$

$$= \int_Q -g(q, p_1, ..., p_N)^2 \frac{\partial}{\partial z_i} f(p_i, q)\, dq \tag{3.8}$$

To minimize $\mathcal{H}$, each sensor should move in the direction of the negative gradient. Let $\dot{p}_i$ be the control input to sensor i. Then the control input for each sensor is:

$$\dot{p}_i = -k \frac{\partial \mathcal{H}}{\partial z_i} \tag{3.9}$$

where k is a scalar constant. This is a general controller which can be used for any sensing function, $f(p_i, q)$. In the next section a practical function for $f(p_i, q)$ is described so that this controller can be used.

3.3.5 Gaussian Sensing Function

We use the covariance between points p_i and q as the sensing function:

$$f(p_i, q) = Cov(p_i, q) \tag{3.10}$$

Ideally, the covariance between the i^{th} sensor and each point of interest, q should be known beforehand. As this is not possible, the authors chose to use a multivariate Gaussian as a first-approach approximation of the sensing quality function. Using a Gaussian to estimate the covariance between points in underwater systems is common in objective analysis [51].

Quantities of interest, like algae blooms etc., in the oceans or rivers tend to be stratified in layers with higher concentrations at certain depths. Thus, the sensor reading at a position p_i and depth d is likely to be similar to the reading at position q if it is also at depth d. And the sensor readings are less likely to be correlated between two points at different depths. So we model the covariance function as a three-dimensional Gaussian, which has one variance based on the surface distance (σ_s^2) and another based on the difference in the depth (σ_d^2) between the two points.

Let $f(p_i, q) = Cov(p_i, q)$ be the sensing function where the sensor is located at point $p_i = [x_i, y_i, z_i]$ and the point of interest is $q = [x_q, y_q, z_q]$. Define σ_d^2 to be the variance in the direction of depth and σ_s^2 to be the variance in the sensing quality based on the surface distance. We then write our sensing function as:

$$f(p_i, q) = Cov(p_i, q) = Ae^{-\left(\frac{(x_i-x_q)^2+(y_i-y_q)^2}{2\sigma_s^2}+\frac{(z_i-z_q)^2}{2\sigma_d^2}\right)} \tag{3.11}$$

where A is a constant related to the two variances, which can be set to 1 for simplicity.

3.3.6 Gaussian-Based Decentralized Controller

We take the partial derivative of the sensing function from Equation 3.11 to complete the gradient of our objective function shown in Equation 3.8. The gradient of the sensing function $\frac{\partial}{\partial z_i} f(p_i, q)$ is:

$$\begin{aligned}\frac{\partial}{\partial z_i} f(p_i, q) &= \frac{\partial}{\partial z_i} Ae^{-\left(\frac{(x_i-x_q)^2+(y_i-y_q)^2}{2\sigma_s^2}+\frac{(z_i-z_q)^2}{2\sigma_d^2}\right)} \\ &= -f(p_i, q)\frac{(z_i - z_q)}{\sigma_d^2}\end{aligned} \tag{3.12}$$

Substituting this into Equation 3.8, we get the objective function:

$$\frac{\partial \mathcal{H}}{\partial z_i} = -\int_Q \left(\sum_{j=1}^{N} f(p_j, q)\right)^{-2} \frac{\partial}{\partial z_i} f(p_i, q)^{-1}\, dq \tag{3.13}$$

$$= \int_Q g(q, p_1, ..., p_N)^2 f(p_i, q)\frac{(z_i - z_q)}{\sigma_d^2} dq \tag{3.14}$$

3.3.7 Controller Convergence

To prove that this gradient controller (Equation 3.9) converges to a critical point of $\mathcal{H}$, the following conditions must be satisfied [6, 45, 56]:

1. $\mathcal{H}$ must be differentiable;
2. $\frac{\partial \mathcal{H}}{\partial z_i}$ must be locally Lipschitz;
3. $\mathcal{H}$ must have a lower bound;
4. $\mathcal{H}$ must be radially unbounded or the trajectories of the system must be bounded.

In [20, 25] Detweiler *et al.* show that the gradient controller satisfies all the conditions for controller convergence and thus converges to a critical point of $\mathcal{H}$.

3.3.8 Algorithm Implementation

Algorithm 1 shows the implementation of the decentralized depth controller (Equation 3.7) in pseudo-code. The procedure receives, as input, the depths of all other nodes in communication range. The procedure requires two functions `F(p_i,x,y,z)` and `FDz(p_i,x,y,z)`. These functions take the sensor location p_i and the point $[x, y, z]$ that we want to cover. The first function, `F(p_i,x,y,z)`, computes the covariance between the sensor location and the point of interest. The second function, `FDz(p_i,x,y,z)`, computes the gradient of the covariance function with respect to z at the same pair of points.

After the procedure computes the numeric integral, it computes the change in the desired depth. This change is bounded by the maximum speed the node can travel. The algorithm then checks if the desired change is less than some threshold. If it is, the algorithm returns true to indicate that the algorithm has converged. If it has not converged, the procedure changes the node depth and returns false.

Algorithm 1 Decentralized Depth Controller

```
1: procedure UPDATEDEPTH($p_1 \cdots p_N$)
2:     $integral \leftarrow 0$
3:     for $x = x_{min}$ to $x_{max}$ do
4:         for $y = y_{min}$ to $y_{max}$ do
5:             for $z = z_{min}$ to $z_{max}$ do
6:                 $sum \leftarrow 0$
7:                 for $i = 1$ to $N$ do
8:                     $sum += $ F(p_i,x,y,z)
9:                 end for
10:                $integral += \frac{-1}{sum^2} *$ FDz(p_i,x,y,z)
11:            end for
12:        end for
13:    end for
14:    $delta = K * integral$
15:    if $delta > maxspeed$ then
16:        $delta = maxspeed$
17:    end if
18:    if $delta < -maxspeed$ then
19:        $delta = -maxspeed$
20:    end if
21:    changeDepth(delta)
22: end procedure
```

3.4 Summary

In this Section, we provide information on the hardware platform and the software interface which are important for understanding the background work of this thesis. In Chapter 4, we device path planning algorithms that will use sensor network and underwater robot presented in this chapter to increase overall sensing. We also present a modified version of Algorithm 1 as one of the proposed underwater robot path planning methods.

Chapter 4

Adaptive Sampling for Underwater Mobile Robots: Algorithms

4.1 Introduction

In this chapter we describe three different algorithms for planning the path of a mobile underwater robot in presence of semi-mobile sensors deployed in the system. The static underwater sensors, as well as the mobile underwater robot all communicate acoustically, so their range is limited.

First, in Section 4.2, we describe the path planning algorithm using the *Voronoi Tessellation* method, which is a global path planning method as it needs to know positions of all sensors. Depending on these positions the algorithm outputs the positions along the water column through which the path of the underwater robot should pass. Thus, the robot already knows the positions in the water column where it will sense information before entering the water. This method typically chooses the path points such that the influence of the neighboring two sensor nodes is the least on those points. This is the ideal case and should maximize the information gained from the water column. The path obtained this way is the Voronoi Robot Path.

In Section 4.3, we describe planning the path of the underwater robot using an approach inspired by the *Tangent Bug algorithm* which is a method used primarily for obstacle avoidance by a mobile robot. This is a purely local path planning algorithm because when the robot is released into the water, it has no idea of the location of the sensor nodes except the first one. The robot then locally finds its way to this sensor at the same time maintaining distance so that they are not covering overlapping regions. This sensor node then transmits information about the position of the next sensor node to the robot and the process continues in this manner. Since the underwater sensors tell the mobile robot which point it needs to go towards next, the robot has to be within the communication range of the sensor for this data to be transmitted. Hence, the Tangent Bug Robot path often looks as if it goes from a point very near to a sensor to a point very near the next sensor.

Finally in Section 4.4, we develop a path planning algorithm using a modified version of Decentralized Depth Adjustment algorithm which has been described in Chapter 3. To implement the path planning of the mobile robot we need to modify the algorithm presented in the previous chapter. Instead of just taking the location of the sensors as input, this algorithm also takes the tentative positions of different points in the robot path as input which are called *robot waypoints*. The algorithm then optimizes the positions of the sensors as well as the robot waypoints. Now, when the underwater robot is released in the water column, the sensor node nearest to the robot informs it about the next robot waypoint that it should travel to for sensing. In this chapter, we also show that this modified adaptive decentralized algorithm converges to a local minima.

The motivation behind using these techniques, the approach, and the implementation of the algorithms are discussed in details in the following sections.

4.2 Voronoi Path Algorithm

Voronoi Tessellation is a method of dividing a given space into a number of regions. Such a region is called a Voronoi cell and the set of all Voronoi cells for a given set of points is called a Voronoi diagram. Given a set of points, Voronoi Tessellation divides the space in such a way that all intermediate points lying closer to a particular point than other points in the set P lies in its Voronoi Cell. Mathematically, given a set of coplanar points $P = \{p_1, \cdots, p_n\}$ of points in a plane and a distance function $d(x, y)$ (the distance between two points x and y), a Voronoi Tessellation is a subdivision of the space into n different cells, one for each point in P such that a point q lies in the cell corresponding to a point p_i if and only if $d(p_i, q) < d(p_j, q)$ for $i \neq j$.

4.2.1 Problem Formulation

Let us assume that the coplanar points are the sensors in a plane of the water column. If we draw Voronoi cells (Fig 4.1) around each of the sensors, then the boundaries of the Voronoi Cell between any two sensors is a straight line which is equidistant from the two sensors on either side of it. So the influence of these two sensors is the least on this straight line. Now if we release a mobile robot to maximize the information gain from the water column then it should pass through this straight line. This is because the mobile robot is passing over points which are least influenced by the static sensors. This is the ideal case and should maximize the information gained from the water column. The path obtained this way is the Voronoi Robot Path and can be seen in Fig. 4.2. To apply the Voronoi Tessellation method to our problem, we need to do the following: Firstly, each sensor arranges itself on the basis of the Algorithm 1. These static sensor locations are sent as input to the Voronoi Tessellation Algorithm 2 . The algorithm finds the Voronoi path and outputs the path points at which the mobile robot should sense information. A mobile robot starts on the first point of interest and then moves along the path by joining all such points of interest along the path.

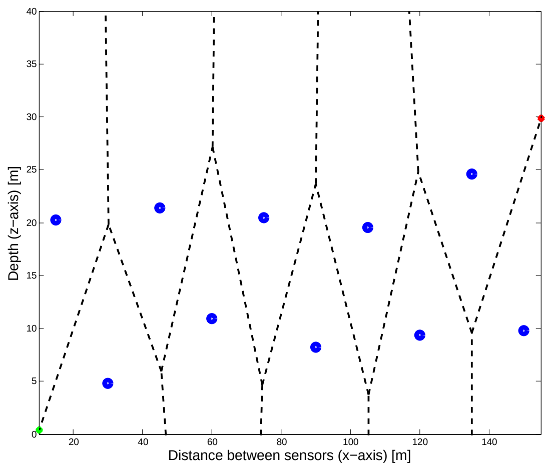


Figure 4.1: Voronoi Cell Diagram for Static Sensor Layout - Bounded on all sides

At times there can be very sparse Voronoi vertices for a given set of sensors. In our algorithm that will signify that even though the mobile robot is traversing the entire water column, it does not sense at all possible points of interests. Therefore, we need to explicitly add more robot way-points, or intermediate points, in between the Voronoi vertices that we obtain from this algorithm.

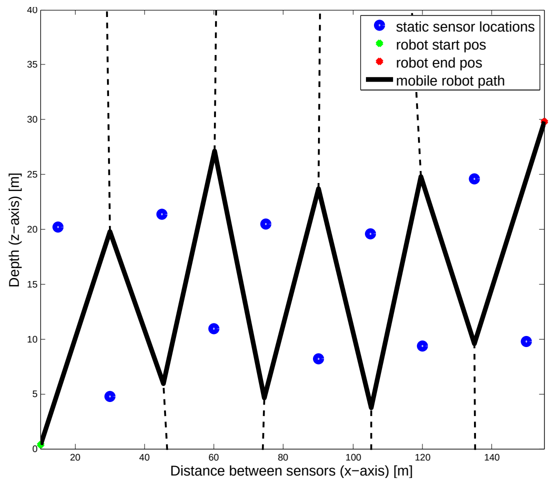


Figure 4.2: Mobile Robot path obtained from Fig. 4.1

For our problem we mostly define the water column to be a depth of 30m. For mathematical purposes, we can say that it lies on the positive quadrant of the Cartesian plane and the y-axis spans from 0 to 30 units. The sensors are always at a depth between 0m to 30m. However, often the Voronoi vertices can lie at great distances from the region of the sensors. For example, if all the sensors are colinear then the Voronoi vertices will lie at infinite distances and all the Voronoi edges will be parallel to each other passing in between each sensor. In such a scenario it is impossible to find a mobile robot path with the help of the Voronoi vertices. This is an extreme case. But even when all the sensors are not colinear, due to the layout, some Voronoi vertices can lie outside the region of the water column. In such a scenario, if we do not do any further processing then that would lead to discontinuities in the path of the mobile underwater robot. So in our algorithm, when we find a Voronoi vertex which lies outside the water column, we find the intersection point between the lines connecting this Voronoi vertex and the previous one and the upper or lower edge of the water column. This point of intersection is designated as the new sensing point and the the vertex lying outside the water column is discarded. We continue this process until we have eliminated all the Voronoi vertices that lie outside the water column.

4.2.2 Algorithm Implementation

Algorithm 2 VoronoiPath: Mobile Robot Path Planning using *Voronoi Tessellation*

```
Require: Sensor Locations, sensors, Robot Start position, $p_{start}$ (optional); Robot Destination, $p_{end}$ (optional)
Ensure: Path Points for Mobile Robot, $path_{points}$
procedure VoronoiPath(sensors)
    $[M_{adj}, V_{voro}] \leftarrow$ AdjacencyMatrix(sensors)
    $[p_{start}, p_{end}] \leftarrow$ ChoosePoints($V_{voro}$, sensors) ▷ First and last Voronoi vertices chosen as $p_{start}$ and $p_{end}$
    $[d, pred] \leftarrow$ Dijkstra($M_{adj}, p_{start}$)
    $path_{points} \leftarrow$ PathConstruct($V_{voro}, pred, p_{start}, p_{end}$)
    return $path_{points}$
end procedure

procedure AdjacencyMatrix(sensors)
    $V_{voro} \leftarrow$ voronoi(sensors)    ▷ $V_{voro}$ gets the $N$ $finite\_vertices\_of\_Voronoi\_Edges$
    for $i \leftarrow 1$ to $N$ do
        for $j \leftarrow 1$ to $N$ do
            $M_{adj}(i,j) = \text{dist}(V_{voro}(i), V_{voro}(j))$
        end for
    end for
    return $M_{adj}, V_{voro}$    ▷ Returns the Adjacency Matrix, $M_{adj}$ for $V_{voro}$.
end procedure

procedure PathConstruct($V_{voro}, pred, p_{start}, p_{end}$)
    $array \leftarrow V_{voro}(p_{end})$
    $i \leftarrow 2$
    while $p_{end} \neq p_{start}$ do
        $p_{prev} \leftarrow pred(p_{end})$
        $path_{points}(i) \leftarrow V_{voro}(p_{prev})$
        $p_{end} \leftarrow p_{prev}$
        $i \leftarrow i + 1$
    end while
    $path^{new}_{points} \subset (path_{points} \cup path^{intersection}_{points})$ s.t. all the points lie within the boundaries defined by user.
    return $path^{new}_{points}$    ▷ Returns all path_points
end procedure

```

Algorithm 2 shows the implementation of the path planning algorithm based on Voronoi Tessellation. The procedure receives the location of all the sensors in the water column as input. The main procedure requires three different functions such as: AdjacencyMatrix, DijkstraShortestPath and, PathConstruct.

The first function, AdjacencyMatrix, takes the location of the sensors *sensorArr* as an input and finds the vertices of the Voronoi cells around each sensor. It then connects each of the Voronoi vertices with the remaining vertices and forms a complete graph. It then creates an adjacency matrix M_{adj} which contains information on each of these Voronoi edges. The length of the edge, i.e., the distance between any two Voronoi vertices, is stored as the weight of the edge in the adjacency matrix.

The second function, DijkstraShortestPath, takes the the adjacency matrix M_{adj} as input and finds the shortest distance path d and the predecessor matrix *pred* that connects the source p_{start} and the destination p_{end} using the *Dijkstra's Shortest path algorithm*. p_{start} and p_{end} can be chosen by the user or by using another function choosePoints.

Lastly, function PathConstruct takes the predecessor matrix *pred* as an input and joins the vertices on the shortest path; adds intermediate points; and finally outputs the sensing path for the mobile robot. This function also limits the Voronoi edges within the region of the water column, by truncating the edges, if required, as discussed earlier.

The simulation experiments and analysis of mobile robot path obtained by this method is discussed in Section 5.7.

4.3 Tangent Bug Path Algorithm

To plan the path of a mobile robot we need to know the environment in which the path needs to be planned. Sometimes a global map of the environment is not available when the robot starts moving towards its destination. A class of algorithms called the *Bug Algorithms* have been proposed for path planning from a known fixed starting position to a known fixed destination. Bug algorithms operate by switching between two simple behaviors:

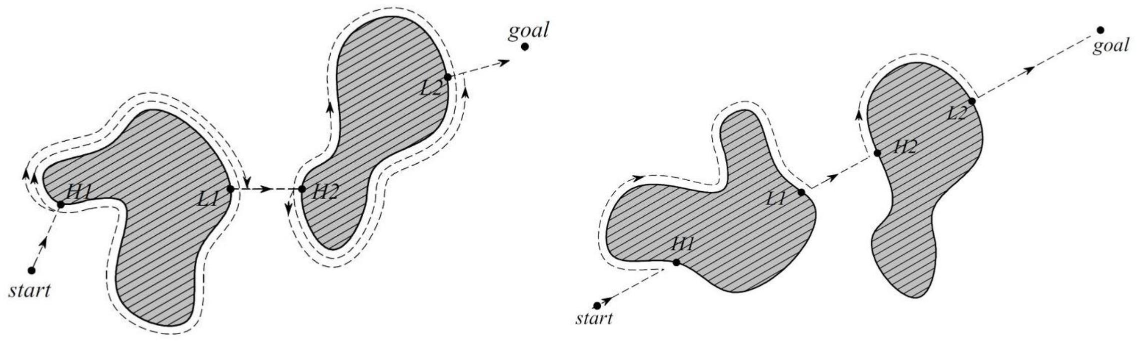
1. Moving directly towards the goal location.

2. Circumnavigating an obstacle.

The Bug algorithms are guaranteed to take the mobile robot from the start to the destination, given it is accessible. There are three popular Bug algorithms - Bug 1, Bug 2 and the Tangent Bug algorithm [28].

Generally for the Bug algorithms, user assumes a point robot with perfect sensors. For Bug 1 and Bug 2 algorithms, very simple sensors like contact sensors are sufficient. However, there should be a finite range sensor for Tangent Bug algorithm. The algorithm also assumes that the robot's position is accurately known and that using the sensors the robot is capable of measuring the distance between two points. We will discuss Bug 1 and Bug 2 briefly and then move on to discussing the Tangent Bug algorithm as it is the one which is relevant to this thesis.

4.3.1 Bug 1 and Bug 2 Algorithms



(a) Bug 1 (b) Bug 2

Figure 4.3: Path Planning with Bug Algorithms [1]

In the **Bug 1** Algorithm the robot follows the following steps:

- The robot heads towards the destination.
- If the robot encounters an obstacle, it circumnavigates it and stores/remembers at each point how close it was to the goal.
- After circumnavigating the obstacle, the robot returns to the point that is closest to the obstacle by wall-following and then continues on the path towards the destination.

Fig 4.3(a) shows how a point robot traverses an environment with two obstacles to reach towards the destination using the Bug 1 algorithm. The Bug 1 algorithm can be expensive. If the cost of the straight line path from the start to the destination is D and the perimeter of obstacle i is P_i, then the distance d traveled by the robot is bounded by $D \leq d \leq D + \frac{1}{2}\sum_i P_i$. Hence the Bug 2 algorithm was proposed in which circumnavigating the obstacle is avoided.

In the **Bug 2** Algorithm the robot follows the following steps:

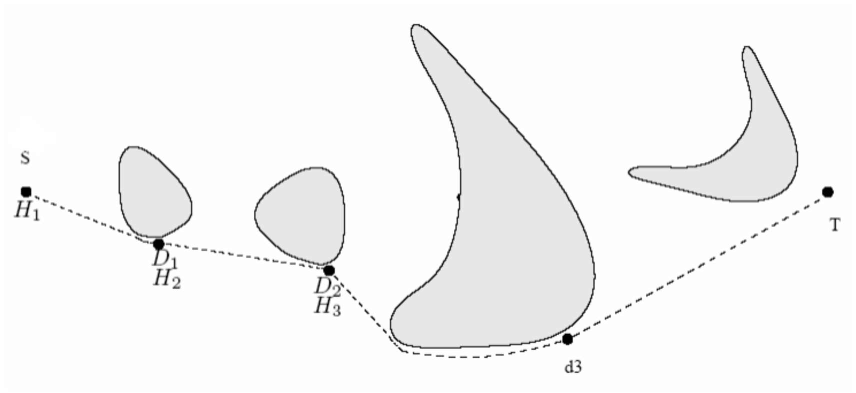
- The robot heads towards the destination on the m-line which is defined as the line joining the start location and the destination.
- If an obstacle is encountered, the robot follows it until the time it again encounters the m-line again closer to the goal.
- The robot then leaves the obstacle and again continues toward the goal on the m-line.

Fig 4.3(b) shows how a point robot traverses an environment with two obstacles to reach towards the destination using the Bug 2 algorithm. One of the major costs associated with the Bug 2 algorithm is due to the requirement that the robot returns to the m-line after circumnavigating an obstacle. It would be more cost effective if the robot could move towards the destination directly as soon as a straight line path from its current position to the destination is feasible.

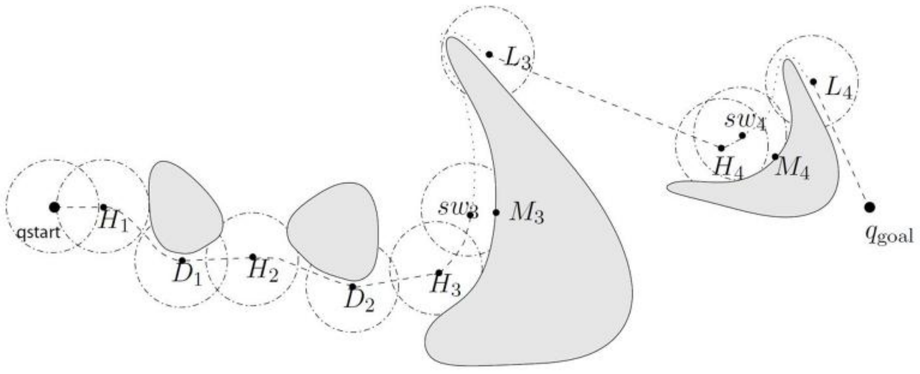
Thus Bug 1 is an exhaustive search algorithm as it looks at all choices but Bug 2 is a greedy algorithm. In many cases Bug 2 outperforms Bug 1, but Bug 1 has a more predictable performance. In real world, however, we have sensors like range sensors which are more powerful than a contact sensor and can look ahead to a finite range. Thus the Tangent Bug algorithm is introduced.

4.3.2 Tangent Bug Algorithm

The **Tangent Bug** algorithm, finds the endpoints of finite continuous segments by drawing a tangent from its current position to the surface of the obstacle. Let these points of intersection between the tangent and the obstacle boundary be denoted by a set $O = \{O_1, \cdots, O_n\}$. The algorithm chooses the point O_i such that the distance $d = dist(P_{curr}, O_i) + dist(O_i, P_{dest})$ (where $dist(x, y)$ is the distance function, P_{curr} is the current position of the robot, and P_{dest} is the destination) is minimized and then moves towards that point. Once it reaches O_i the robot again aims to go towards the destination and repeats the same process if it faces another obstacle. The Tangent Bug algorithm is a local path planning algorithm as it does not need complete knowledge of the environment before the robot starts moving towards the goal.



(a) Robot is equipped with infinite range sensor



(b) Robot is equipped with a finite range sensor

Figure 4.4: Path Planning with Tangent Bug Algorithm [1]

Ideally if the robot can sense infinite distances from its current position then the path planned by the tangent Bug algorithm will look like Fig 4.4(a). However, in most real problems, the robot is equipped with a finite range sensor which can look ahead only upto a finite distance. The path planned in such a case, will look like the one in Fig 4.4(b) or a different variation for the same set of obstacles. As we can see, the planned path depends on the distance till which the robot can look ahead.

4.3.3 Problem Formulation

In our implementation we use Tangent Bug algorithm in an innovative manner. We are inspired by the local look ahead property of the Tangent Bug algorithm. By the local look ahead property we mean that the robot, even though it has a distant goal to reach, can look ahead only upto a certain distance ahead of it. For example in Fig. 4.4(b), the robot can look ahead only upto the range of it's range sensor. Coming back to the proposed algorithm, both the underwater sensors and the underwater robot communicate acoustically, so their range is limited. Since the path of the mobile underwater robot depends on the location of the underwater sensors, one of the possible ways to implement a local path planning algorithm under water is if the sensors can tell the mobile robot the next point that the robot should be sensing at. This point can be determined by the underwater sensor by finding the point on its sensing perimeter which it has the least influence on. However, after a specific distance from the sensor is crossed, there will be many points where the information gained by the sensor is not good. To choose one point out of the possible ones to sense at is difficult for the robot. Instead, it is easier for the sensor to tell the robot in which direction it should go next. So we have designed this algorithm such that at any iteration, the mobile robot knows the position of the next sensor as its immediate destination. The mobile robot enters the water column with the knowledge of the location of the nearest sensor and travels towards it. Thereafter, every time the robot is close to one sensor, it tells the robot the location of the next sensor. The mobile robot has to be within the communication range of the sensor for this data to

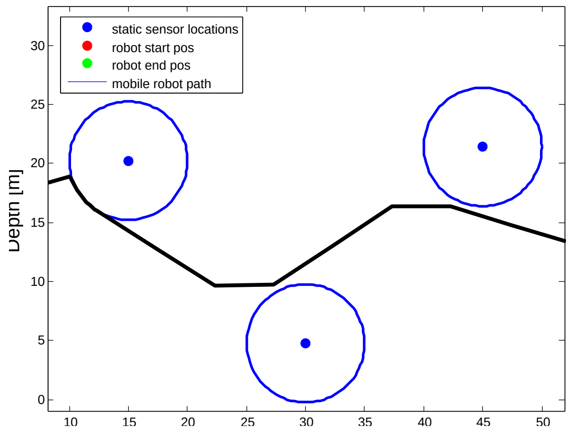


Figure 4.5: Plot showing a section of Sensors, Sensing Bubble surrounding the Sensors and Mobile Robot Path for path planning with modified Tangent Bug Algorithm. $(r_S = 5; r_R = 5)$

be transmitted.

Taking all these into account, we can apply the principles of the Tangent Bug algorithm to plan a path for a mobile underwater robot in a region distributed with sensors. This problem can be formulated using the following steps:

1. The finite radius of sensing and acoustic communication, r_S, forms a circular bubble around the underwater sensors (Fig 4.5). Together they are equivalent to the obstacles on the path of the mobile robot. Any point inside this bubble is well covered by the static sensor and any point outside needs better sensing.

2. The mobile robot does not want to enter the region inside any of these bubbles, but it has to move towards the boundary as it will be able to communicate these with the static sensor to know the next sensor location to determine the direction in which it should move next.

3. As in the case of the sensors, the mobile robot also has a circular sensing and communication bubble surrounding it, of radius r_R, within whose boundaries the sensing is maximum. The r_S and r_R can be same or different, depending on the real world characteristics of the two types of sensors.

4. The mobile robot starts moving in the direction of the next sensor. When it gets close enough to the sensor bubble, it determines the tangent intersection points between its current position and the sensor boundary, picks the intersection point O_i which is closer to final destination, and moves along the tangent to the boundary in the direction of O_i where it will be able to communicate with the sensor to get the required information.

5. Once it reaches O_i and receives the location information of the next sensor, it checks whether there is a m-line (or straight line path) that connects its current position to the center of the next sensor, that does not pass through the sensing bubble of the sensor on whose boundary the robot is present right now.

6. If such a line exists, the mobile robot moves along this line towards the center of the next sensor.

7. In case this line does not exist, the robot incrementally moves following the boundary of the current sensor bubble, at the same time continuously searching for a point O_j from which it will have a m-line towards the next sensor.

8. When it is on such a point, O_j, the robot leaves the boundary and moves towards the next sensor.

9. These steps are repeated until the robot reaches its destination.

Thus, this path planning algorithm redefines the principles of the Tangent Bug algorithm so that it can be applied to planning the path of a mobile robot between semi-mobile sensor nodes. This is an example of a purely local path planning algorithm at any point the robot has information about the immediate goal. The path generated by this algorithm shows how the mobile robot path will look if it is planned based only on local information.

To apply the Tangent Bug method to our problem, we need to do the following: Firstly, each sensor arranges itself on the basis of the Algorithm 1. For simulation purpose, these static sensor locations are sent as input to the modified version of the Tangent Bug algorithm discussed in Algorithm 3. In the real world the mobile robot or the Tangent Bug algorithm will not know the position of all the sensors except the one nearest to it. This is implemented in the Algorithm 3 with the concept of an instant goal or current target position at every iteration. At any iteration, the robot is moving towards the next closest target position instead of the global destination point. The algorithm proceeds in small increments which is called an interval, and then finds all the path points that form the Tangent Bug path and outputs these points at which the mobile robot should sense information. Mobile robot starts from a user defined location in the water column and then moves along the trajectory by joining all such instant goal points along the path.

The simulation experiments and analysis of mobile robot path obtained by this method is discussed in Section 5.8.

4.3.4 Algorithm Implementation

Algorithm 3 shows the implementation of the path planning algorithm based on Tangent Bug Method. The main procedure TanBug receives the location of all the sensors in the water column (*sensors*), the sensing radius of the sensors (r_S), the sensing radius of the robot (r_R), the start position of the robot (p_{start}), and finally the destination or end point (p_{end}) as input. This procedure requires help from two main functions: MoveTowardsTarget and MoveAlongCurve for constructing the mobile robot path. There are five other functions: FindNextPos, FindTangentPoints, IfMoveAlongCurve, Cir2Intersection, and CirLineIntersection which are detailed in the *Appendix* in Section A.

The function MoveTowardsTarget implements the steps which the mobile robot needs to follow when it is moving towards a target. Whereas, the function MoveAlongCurve implements the steps that the mobile robot needs to follow when it is moving along the curve of the sensing boundary of the nearest sensor.

FindNextPos finds the position of the next robot path point when the robot is moving in a straight line path. FindTangentPoints finds the tangents from the current position of the robot to the boundary of the next sensor and returns the two points where the tangents and the sensor boundary touch.

When the robot is sitting on the boundary of the current sensor, the function IfMoveAlongCurve, checks if there exists a m-line from the current robot position to the next sensor which does not pass through the sensing bubble of the current sensor. If such a line exists then the function returns false, else it returns true.

Algorithm 3 TanBugPath: Mobile Robot Path Planning using *Tangent Bug Algorithm*

Require: Original Sensor layout, *sensors*; Sensor View Radius, r_S; Robot View Radius,r_R; Robot Start position, p_{start}; Robot Destination,p_{end}

Ensure: Mobile Robot Waypoints, $path_{points}$;

```
 1: $flag_{intersection} \leftarrow$ FALSE                ▷ Flag to indicate if robot is at sensing boundary
 2: $flag_{target} \leftarrow$ TRUE                       ▷ Flag to indicate if new target needs to be set
 3: $interval \leftarrow$ user_defined_interval
 4:
 5: procedure TanBugPath($sensors, r_S, r_R, p_{start}, p_{end}$)
 6:     $p_{robot} \leftarrow p_{start}$                  ▷ First robot waypoint is $p_{start}$
 7:     $i \leftarrow p_{start X}$                        ▷ Assign x-coordinate of robot start position as i
 8:     while $i < p_{end X}$ do     ▷ Compare x-coordinate of robot end position to find when robot reached destination
 9:         if $flag_{target} ==$ TRUE then
10:             Select $p_{target}$ from $sensors$   ▷ Select next target position, $p_{target}$, based on the min. dist. from current robot position, $p_{robot}$
11:         end if
12:         if $flag_{intersection} ==$ FALSE then
13:             $[p^{new}_{robot}, flag_{intersection}, flag_{target}] \leftarrow$ MoveTowardsTarget()
14:         else if IfMoveAlongCurve($sensor_{curr}$,$p_{target}$, $p_{robot}$) then
15:             $[path_{points}, p^{new}_{robot}, flag_{intersection}, flag_{target}] \leftarrow$ MoveAlongCurve()
16:         end if
17:         $path_{points} \leftarrow [path_{points}; p_{robot}]$     ▷ Add new robot waypoint
18:         $p_{robot} \leftarrow p^{new}_{robot}$                    ▷ Update robot's current position
19:         $i \leftarrow i + interval$
20:     end while
21:     return $path_{points}$
22: end procedure
23:
24: procedure MoveTowardsTarget
25:     if DIST($p_{robot}$,$p_{target}$) $> (r_R + r_S)$ then
26:         $p^{new}_{robot} \leftarrow$ FindNextPos($p_{target}$, $p_{robot}$, $interval$)    ▷ Go straight towards $p_{target}$
27:         $flag_{intersection}, flag_{target} \leftarrow$ FALSE
28:     else
29:         $[flag, t_1, t_2] \leftarrow$ FindTangentPoints($p_{robot}$,$p_{target}$,$r_R$,$r_S$)  ▷ Find tangent point and move towards it.
30:         $p^{next}_{target} \leftarrow$ tentative next target location from $sensors$
31:         if DIST($t_1$, $p^{next}_{target}$)+DIST($t_1$, $p_{robot}$) $<$ DIST($t_2$, $p^{next}_{target}$)+DIST($t_2$, $p_{robot}$) then   ▷ Select intersection point closer to next target
32:             $t \leftarrow t_1$
33:         else
34:             $t \leftarrow t_2$
35:         end if
36:         if DIST($p_{robot}$, $t$) $\leq interval$ OR DIST($p_{robot}$, $p_{target}$) $\leq r_S$ then   ▷ If distance is below threshold then go to the intersection point directly
37:             $p^{new}_{robot} \leftarrow t$
38:             $flag_{intersection}, flag_{target} \leftarrow$ TRUE
39:             $S_{near} \leftarrow p_{target}$
40:         else                                ▷ Else move towards the intersection in small intervals
41:             $p^{new}_{robot} \leftarrow$ FindNextPos($t$, $p_{robot}$, $interval$)
42:             if DIST($p^{new}_{robot}$, $p_{target}$) $\leq (r_S + interval)$) then
43:                 $flag_{intersection}, flag_{target} \leftarrow$ TRUE
44:                 $S_{near} \leftarrow p_{target}$
45:             else
46:                 $flag_{intersection}, flag_{target} \leftarrow$ FALSE
47:             end if
48:         end if
49:     end if
50: end procedure
```

Algorithm 3 TanBugPath: Mobile Robot Path Planning using *Tangent Bug Algorithm* (cont.)

```
51: procedure MoveAlongCurve
52:     $d \leftarrow \text{DIST}(p_{target}, S_{near})$
53:     $r \leftarrow \sqrt{|d^2 - r_S{}^2|}$
54:     $circle_1 \leftarrow [p_{target}, r]$
55:     $circle_2 \leftarrow [S_{near}, r_S]$
56:     $[fl, p_1, p_2] \leftarrow \text{CIR2INTERSECTION}(circle_1, circle_2)$
57:     if $\text{DIST}(p_{target}, p_{robot}) < \text{DIST}(p_{target}, p_1)$ AND $\text{DIST}(p_{target}, p_{robot}) < \text{DIST}(p_{target}, p_2)$ then    ▷ Both
    intersection point is behind current rob pos. So going in st. line towards next target.
58:         $p_{robot}^{new} \leftarrow \text{FindNextPos}(p_{target}, p_{robot}, interval)$
59:         $flag_{intersection}, flag_{target} \leftarrow \text{FALSE}$
60:     else    ▷ Need to follow sensor radius circumference. So find the intersection point closest to $p_{robot}$
61:         if $\text{DIST}(p_{robot}, p_1) < \text{DIST}(p_{robot}, p_2)$ then
62:             $p_{robot}^{new} \leftarrow p_1$
63:         else
64:             $p_{robot}^{new} \leftarrow p_2$
65:         end if
66:         $flag_{intersection} \leftarrow \text{TRUE}$
67:         $flag_{target} \leftarrow \text{FALSE}$
68:         $S_{near} \leftarrow p_{target}$
69:         $P \leftarrow \text{circleCircPts}(S_{near}, r_S, p_{robot}, p_{robot}^{new})$    ▷ Move along circumference in small intervals. Store
    these points in array $P$.
70:         for $n = 1 \rightarrow size(P)$ do
71:             $path_{points} \leftarrow [path_{points}; P(n)]$
72:             $k \leftarrow k + 1$
73:         end for
74:     end if
75: end procedure
```

The function Cir2Intersection is used to find out if two circles intersect each other or not. There can be four different possibilities:

- The two circles do not intersect and lie apart from each other, hence, there are no intersection points;
- One circle lies inside the other circle so there is are no intersection points;
- The two circles are co-incident, so there are infinite solutions; and
- The circles overlap and thus intersect each other at two specific points.

Lastly, the function CirLineIntersection is used to find out if a given line and a given circle intersect each other or not. For this function also there can be several outcomes:

- The line does not intersect the circle and they lie apart from each other;

- The line is a tangent to the circle, so there is one intersection point; and
- The line is a secant to the given circle so it cuts the circle at two specific points.

Depending on the outcome of the algorithms different actions should be taken.

Finally, at Line 69 of Algorithm 3, we use one more function, circleCircPts (not been shown in the algorithm section for simplicity) which incrementally finds out the next position of the robot when it is moving along the boundary of the current sensor node.

The Algorithm3 is long but simple. It varies between two principle states:

- When robot is NOT on any intersection point; and
- When robot is on one intersection point.

Each of the two states can again be divided into two sub-states each.

When the robot is NOT on any intersection point, then it can either be:

- Far from the target (next sensor) so keeps moving towards it in a straight line; or
- Close to the sensor boundary so finds the intersection point of the tangents from current position to the sensor boundary and moves along the the gradient of this tangent. The current robot position changes after each iteration and son the intersection point changes. If the interval is small, the robot follows a curved path and does not touch the sensing boundary. This can be seen in Fig 4.5.

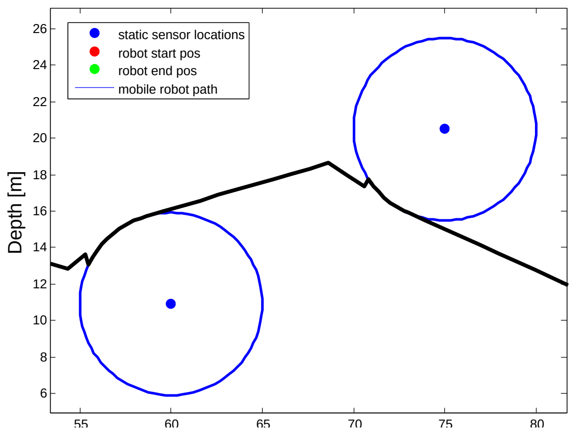


Figure 4.6: Plot showing a section of Sensors, Sensing Bubble surrounding the Sensors and Mobile Robot Path for path planning with modified Tangent Bug Algorithm. $(r_S = 5; r_R = 2)$

When the robot is on one intersection point, it can either:

- It starts moving towards the next sensor as there is a straight path connecting them; or
- It has to move along the boundary of the current sensor till it reaches a point on the sensing boundary where from where a straight path exists. This can be observed in Fig 4.6 where the robot moves to the sensor boundary when it knows it needs to follow the boundary before it can moves in a straight lines towards the next sensor. In the figure, we can notice that there is a sudden kink in the robot path when it suddenly moves from a position along the straight line of gradient of the tangent to a point on the boundary of the nearest sensor. This is because as soon as the robot figures out it needs to move along the sensor boundary it moves to the nearest intersection point and the interval between the two positions is not large enough (since $r_R = 2$) to guarantee a smooth path. We will notice that this phenomenon does not happen when $r_R = 5$ or more.

Another functionality we wanted to incorporate in the modified Tangent Bug algorithm is that as and when the mobile robot is moving through the network of sensors, the sensors which have already interacted with the robot should adapt to it's presence and change it's depth such that the overall sensing in the region is maximized. The simulation experiments and analysis of mobile robot path obtained by this method is discussed in Section 5.8.

4.4 Adaptive Path Algorithm

The sensor network described in Chapter 3 has a depth adjustment system that makes it very efficient for gathering information from the full water column. However, these sensors are anchored to the bottom of the water column so their horizontal position cannot change. The sensors are deployed randomly so there is no control on where they land. The Algorithm 1 is able to position the sensors to obtain a good estimate of the entire region of interest irrespective of the position of each node. However, intuitively, a better estimate of the system can be obtained if this constraint could be relaxed. To overcome this limitation, we introduce a mobile underwater robot into the system which should be able to gather information from the region at locations where the constrained sensors cannot reach. In the previous sections, we have discussed one global method (Section 4.2) and one local method (Section 4.3) for planning the path of such a mobile underwater robot through the sensors already deployed in the water column. In this section, we adapt our decentralized control algorithm to allow the sensor network to tell the mobile robot the best path to traverse. The algorithm also adapts the positions of the underwater sensor nodes with respect to path of the mobile robot. We propose a decentralized approach where each sensor along the robot's path informs the robot of the next part of the path. It is important to ensure that the network does not need to transmit all information to the robot a priori.

This approach is unique in that it lets the network of sensors reconfigure its position to new depths before the mobile robot is introduced into the system. This way, data can be collected from locations of highest utility in parallel. This is efficient and unique because if the mobile robot communicated with local nodes and decided its trajectory online, just like in Section 4.3, then this reconfiguration would not be possible.

In the following sections, we discus how we redefine the objective function, the decentralized controller and modify the Algorithm 1 to work with mobile underwater robots.

4.4.1 Problem Formulation

In order to allow the underwater sensor network to compute the best path for the robot, we start by extending our basic controller described by Equation 3.5 to include sensing locations along the robot's path, $\{p_1^R, ..., p_M^R\}$. We use the superscript ^{R} to indicate that this is a sensing location along the robot's path. We choose these as points evenly spaced between the locations of the static sensor nodes. Ideally, the mobile robot should be sensing continuously along its path of traversal, and there should be infinite points along the robot's path. Although, in simulation, we use just a few intermediate points between the sensors to minimize path complexity. This is because, due to the nature of the medium of water, if the water column is sampled continuously then we might not be able to observe the difference in the constituents of one particular position with the next in the water column. So to get consistent reading of the water column, without contamination from one point to the next, a possible better method is to collect samples of water from different points which are apart from each other and strategically important. This is the reason why we assign robot waypoints which are apart from each other in the horizontal plane and let the algorithm align the points so that they are placed in the strategically important positions along the depth of the water column.

4.4.2 Sensing Objective Function

The objective function over the region of interest given by Equation 3.5 now includes sensing locations $\{p_1^R, ..., p_M^R\}$ along the robot's path. Therefore, the new sensing objective function is:

$$\mathcal{H}(p_1, ..., p_N, p_1^R, ..., p_M^R) = \int_Q g(q, p_1, ..., p_N, p_1^R, ..., p_M^R)\, dq \tag{4.1}$$

This controller works to optimize the sensing with all of the sensor node locations and all of the robot sampling points. However, this could result in paths where the robot alternates moving between the surface and bottom of the water column. This is costly in terms of traversal time and does not take advantage of the sensor nodes that can cover the extreme depths more easily than the robot. This is why we need to include another term to the objective function (discussed in the next section).

4.4.3 Distance Cost Function

To reduce the length of the path the robot travels, we introduce an additional term that aims to minimize the length of the path that the robot traverses. We call this $\mathcal{P}$ and this is dependent on the total distance that the mobile underwater robot traverses in the water column. If the virtual sensor at location i is represented as $p_i^R = (x_i^R, y_i^R, z_i^R)$ then the Euclidean distance between p_i^R and it's adjacent sensor p_{i+1}^R is

$$dist(p_i^R, p_{i+1}^R) = \sqrt{(x_{i+1}^R - x_i^R)^2 + (y_{i+1}^R - y_i^R)^2 + (z_{i+1}^R - z_i^R)^2} \tag{4.2}$$

For controlling the path length of the mobile robot we want to control the total path it traverses. If there are M points where the mobile robot senses in the water column, the total path traversed by the mobile robot can be calculated by summing the straight line distance between these M points (assuming the mobile robot always travels in a straight line between the points of sensing and the distance between points at i and $i+1$ is not very long). This cost function $\mathcal{P}(p_1^R, ..., p_M^R)$, henceforth referred to as only $\mathcal{P}$, can be denoted with the total distance:

$$\begin{aligned} \mathcal{P}(p_1^R, ..., p_M^R) &= \sum_{i=1}^{M-1} dist(p_i^R, p_{i+1}^R) && (4.3) \\ &= \sum_{i=1}^{M-1} \sqrt{(x_{i+1}^R - x_i^R)^2 + (y_{i+1}^R - y_i^R)^2 + (z_{i+1}^R - z_i^R)^2} && (4.4) \end{aligned}$$

4.4.4 Robot–Sensor Node Objective Function

The sensing objective function distributes the sensors in the water column. The distance objective controls the maximum path length of the mobile robot. These two objective functions are combined together to achieve the goals of the algorithm. To combine these two objective functions together we need to decide on a weight function α. We combine $\mathcal{H}(p_1, ..., p_N, p_1^R, ..., p_M^R)$ and $\mathcal{P}(p_1^R, ..., p_M^R)$ to create a new objective function that we use as the new decentralized controller that controls the robot path and the sensor node locations. We call this controller $\mathcal{H}^R(p_1, ..., p_N, p_1^R, ..., p_M^R)$, henceforth referred to as only $\mathcal{H}^R$, and define it as:

$$\mathcal{H}^R(p_1, ..., p_N, p_1^R, ..., p_M^R) = (1-\alpha)\mathcal{H}(p_1, ..., p_N, p_1^R, ..., p_M^R) + \alpha\mathcal{P}(p_1^R, ..., p_M^R) \tag{4.5}$$

Simplified,

$$\mathcal{H}^R = (1-\alpha)\mathcal{H} + \alpha\mathcal{P} \tag{4.6}$$

As discussed before, the first component of Eqn. 4.6 optimizes the positions for sensing and the second component of that equation minimizes the path length the robot travels.

4.4.5 Robot–Sensor Node Decentralized Controller

As in Section 3.3.4, we develop a decentralized controller by taking the gradient of $\mathcal{H}^R$ with respect to changes in depth and move each sensor node and robot sensing location by $-k\frac{\partial \mathcal{H}}{\partial z_i}^R$.

4.4.5.1 Decentralized Controller for Sensing Objective Function

The controller derivation for the sensing objective function closely follows that of the controller solely for sensor nodes Equation 3.12 and 3.14. We take the partial derivative of the sensing function from Equation 4.1 to complete the gradient of our objective function shown in Equation 3.8. As before, the gradient of the sensing function $\frac{\partial}{\partial z_i} f(p_i, q)$ is:

$$\begin{aligned} \frac{\partial}{\partial z_i} f(p_i, q) &= \frac{\partial}{\partial z_i} A e^{-\left(\frac{(x_i-x_q)^2+(y_i-y_q)^2}{2\sigma_s^2} + \frac{(z_i-z_q)^2}{2\sigma_d^2}\right)} \\ &= -f(p_i, q)\frac{(z_i - z_q)}{\sigma_d^2} \end{aligned} \tag{4.7}$$

Substituting this into Equation 3.8, we get the objective function:

$$\begin{aligned} \frac{\partial \mathcal{H}}{\partial z_i} &= -\int_Q \left(\sum_{j=1}^{N} f(p_j, q)\right)^{-2} \frac{\partial}{\partial z_i} f(p_i, q)^{-1}\, dq + \frac{\partial}{\partial z_i}\phi(p_i) \\ &= \int_Q g(q, p_1, ..., p_N, p_1^R, ..., p_M^R)^2 f(p_i, q)\frac{(z_i - z_q)}{\sigma_d^2} dq + \frac{\partial}{\partial z_i}\phi(p_i) \end{aligned} \tag{4.8}$$

4.4.5.2 Decentralized Controller for Distance Cost Function

To minimize the cost function in eqn.(4.4), i.e., $\mathcal{P}$, we take the gradient of $\mathcal{P}$ with respect to z_i.

$$\begin{aligned}
\frac{d\mathcal{P}}{dz_i} &= \frac{d}{dz_i}\sum_{i=1}^{M-1} dist(p_i^R, p_{i+1}^R) && (4.9)\\
&= \sum_{i=1}^{M-1} \frac{d}{dz_i} dist(p_i^R, p_{i+1}^R) && (4.10)\\
&= \sum_{i=1}^{M-1} \left(\frac{d}{dz_i^R}\sqrt{(x_{i+1}^R - x_i^R)^2 + (y_{i+1}^R - y_i^R)^2 + (z_{i+1}^R - z_i^R)^2} \right) && (4.11)\\
&= \sum_{i=1}^{M-1} \left(\frac{(z_{i+1}^R - z_i^R)}{\sqrt{(x_{i+1}^R - x_i^R)^2 + (y_{i+1}^R - y_i^R)^2 + (z_{i+1}^R - z_i^R)^2}} \right) && (4.12)
\end{aligned}$$

We can specify the number of neighboring sensing locations that can control the gradient descent function $\frac{d\mathcal{P}}{dz_i}$. This is specified through a parameter called *hops* which denotes the number of sensor locations that mobile robot can look at in both directions. So the limits of summation can be anywhere between 2 to $(M-1)$.

4.4.5.3 Combined Controller

Similar to Section 4.4.4, the combined value of the derivative of objective function is:

$$\frac{d\mathcal{H}^R}{dz_i} = (1-\alpha)\frac{d\mathcal{H}}{dz_i} + \alpha\frac{d\mathcal{P}}{dz_i} \quad (4.13)$$

4.4.6 Controller Convergence

To prove that our gradient controller Equation 4.6 converges to a critical point of $\mathcal{H}^R$, just like in Section 3.3.7, we must show that satisfied [6, 45, 56]:

1. $\mathcal{H}^R$ must be differentiable;

2. $\frac{\partial\mathcal{H}^R}{\partial z_i}$ must be locally Lipschitz;

3. $\mathcal{H}^R$ must have a lower bound;

4. $\mathcal{H}^R$ must be radially unbounded or the trajectories of the system must be bounded.

These points assure convergence to a critical point of $\mathcal{H}^R$, but a critical point could be a maximum or a minimum. Fortunately, small changes to the system will typically result in the system converging to a local minimum, as it is more stable. It may not be the global minimum, but we show in simulation that we obtain good results.

Theorem 1. *The controller* $-k\dfrac{d\mathcal{H}^R}{dz_i}$ *converges to a critical point of* $\mathcal{H}$. *In other words as time, t, progresses the output of the controller will go to zero:*

$$\lim_{l\leftarrow\infty} -k\frac{d\mathcal{H}^R}{dz_i} = 0 \tag{4.14}$$

Proof. We show that the objective function satisfies the conditions outlined above. In Section 4.4.5, we have shown that each of the two components of $\mathcal{H}^R$, $\mathcal{H}$ and $\mathcal{P}$ are differentiable with respect to z_i. And since α and $(1-\alpha)$ are both constants, this means $\mathcal{H}^R$ is differentiable with respect to z_i, thus condition 1 is satisfied.
The slope of $\frac{\partial\mathcal{H}}{\partial z_i}$ is locally bounded and $\mathcal{P}$ is a continuously differentiable function. As continuous functions are locally bounded, its gradient, $\frac{\partial\mathcal{P}}{\partial z_i}$, is locally bounded as well. Therefore, $\frac{\partial\mathcal{H}}{\partial z_i}^R$ is locally bounded, meaning it is locally Lipschitz and satisfies condition 2.
To show that $\mathcal{H}^R$ is bounded below, to satisfy condition 3, consider the composition of the objective function:

$$\mathcal{H}^R(p_1,...,p_N,p_1^R,...,p_M^R) = (1-\alpha)\int_Q g(q,p_1,...,p_N,p_1^R,...,p_M^R)\,dq + \alpha\sum_{i=1}^{M-1} dist(p_i^R,p_{i+1}^R) \tag{4.15}$$

As in [20, 25], expanding the integral term we get:

$$\int_Q g(q,p_1,...,p_N,p_1^R,...,p_M^R)dq = \int_Q \left(\sum_{i=1}^{N} f(p_i,q) + \sum_{j=1}^{M} f(p_j^R,q)\right)^{-1} dq \tag{4.16}$$

The terms $f(p_i,q)$ and $f(p_j^R,q)$ are both Gaussian functions which is always positive. The sum and the integral of the positive Gaussian function is also positive. The term $(1-\alpha)$ is

always positive as α lies between $[0, 1]$ as we will show in Section 4.4.7.2. Thus this term is bounded below by zero. The second term, $dist(p_i^R, p_{i+1}^R)$, is the distance between two points which is always positive. The sum of positive terms is also positive. Thus, both the terms in $\mathcal{H}^R$ and thus $\mathcal{H}^R$ itself is bounded below by zero, satisfying condition 3.
Unfortunately, both terms in $\mathcal{H}^R$ is not radially unbounded. However, the trajectories of the system can only vary along the z-axis within the water column. Thus, the trajectories of the system are bounded, satisfying condition 4.
Hence, we have satisfied all the conditions for controller convergence, proving that the controller $-k\dfrac{d\mathcal{H}^R}{dz_i}$ converges.

4.4.7 Normalization

It is difficult to decide the range of α since the ratio between the sensing objective function and the distance objective function can be very large (often more than 25000). The maximum value of the sensing objective function occurs when the nodes are laid out in a straight line and the minimum value occurs when the nodes are laid out in an evenly fashion throughout the water column. On the other hand, for the distance objective function, maximum value occurs when the nodes are evenly distributed in a zigzagged fashion and minimum value occurs when the nodes are in the same straight line. These values depend upon the network size. To limit the range of α from 0 to 1 the values of the sensing objective function and the distance objective function are *normalized* so that both the sensing and distance objective function vary in the range of 0 to 1 depending on their initial value.

4.4.7.1 Objective Function

To normalize we first find the maximum and minimum values of the sensing and distance objective function for given number of intermediate nodes for network of different sizes.

Then we fit the values into a function using Matlab function:

$$\rho = polyfit(X, Y, N) \tag{4.17}$$

which finds the coefficients of a polynomial P(X) of degree N that fits the data Y best in a least-squares sense. $\rho = [\rho_1, \rho_2, \rho_3, \cdots \rho_{N+1}]$ is a row vector of length $N + 1$ containing the polynomial coefficients in descending powers. We do it for $N = 2$, so $\rho = [\rho_1, \rho_2, \rho_3]$ as in Section 5 we simulate these and find it is a good fit. We do it for all the 4 quantities - maximum sensing objective function, minimum sensing objective function, maximum distance objective function, minimum distance objective function. Then to find out the range of the objective function for a given size of network and a certain number of intermediate sensing locations, the total number sensing locations can be found with the equation:

$$s = r + i.(r - 1) \tag{4.18}$$

where r = total number of sensors; i = number of robot waypoints between two real nodes; s = total number of sensing locations.

Then, using the following equation we can estimate the minimum and maximum sensing and distance objective function value, v for that particular network size:

$$v = \rho_1 * s^2 + \rho_2 * s + \rho_3 \tag{4.19}$$

Let the maximum sensing objective function be $\mathcal{H}_{max}$, minimum sensing objective function be $\mathcal{H}_{min}$, maximum distance objective function be $\mathcal{P}_{max}$, and minimum distance objective function be $\mathcal{P}_{min}$. The normalized sensing and distance objective function can be found out by the equation:

$$\overline{\mathcal{H}} = \frac{\mathcal{H} - \mathcal{H}_{min}}{\mathcal{H}_{max} - \mathcal{H}_{min}} \tag{4.20}$$

$$\overline{\mathcal{P}} = \frac{\mathcal{P} - \mathcal{P}_{min}}{\mathcal{P}_{max} - \mathcal{P}_{min}} \tag{4.21}$$

These values are determined in the simulation section and vary depending on the chosen network.

4.4.7.2 Robot–Sensor Node Objective Function

We take the Eqn. 4.5 and normalize each of its components to get $\overline{\mathcal{H}}(p_1, ..., p_N, p_1^R, ..., p_M^R)$ and $\overline{\mathcal{P}}(p_1^R, ..., p_M^R)$ according to Eqn. 4.20 and Eqn. 4.21 respectively. We then combine these two terms to obtain the normalized form of the combined objective function. For simplicity, we still refer to this controller as $\mathcal{H}^R(p_1, ..., p_N, p_1^R, ..., p_M^R)$, or $\mathcal{H}^R$ and the weight factor α.

$$\mathcal{H}^R(p_1, ..., p_N, p_1^R, ..., p_M^R) = (1-\alpha)\overline{\mathcal{H}}(p_1, ..., p_N, p_1^R, ..., p_M^R) + \alpha\overline{\mathcal{P}}(p_1^R, ..., p_M^R) \tag{4.22}$$

Simplified,

$$\mathcal{H}^R = (1-\alpha)\overline{\mathcal{H}} + \alpha\overline{\mathcal{P}} \tag{4.23}$$

4.4.7.3 Robot–Sensor Node Decentralized Controller

Similar to Section 4.4.7.2, the normalized value of the derivative of sensing and distance objective function can be found out by the equation:

$$\overline{\frac{d\mathcal{H}}{dz_i}} = \frac{1}{\mathcal{H}_{max} - \mathcal{H}_{min}} \frac{d\mathcal{H}}{dz_i} \tag{4.24}$$

$$\overline{\frac{d\mathcal{P}}{dz_i}} = \frac{1}{\mathcal{P}_{max} - \mathcal{P}_{min}} \frac{d\mathcal{P}}{dz_i} \tag{4.25}$$

The normalized combined value of the derivative of objective function is:

$$\frac{d\mathcal{H}^R}{dz_i} = (1-\alpha)\overline{\frac{d\mathcal{H}}{dz_i}} + \alpha\overline{\frac{d\mathcal{P}}{dz_i}} \tag{4.26}$$

For simplicity, again we refer to the normalized equation in the same terms as shown in Eqn. 4.13.

The normalization does not affect the convergence of the controller because:

- Both $\mathcal{H}_{max}$ and $\mathcal{H}_{min}$ are bounded below by zero;
- $\mathcal{H}_{max} > \mathcal{H}_{min}$ always;

This implies that, depending on the value of $\mathcal{H}$, sometimes $\overline{\mathcal{H}}$ can have a negative value which is bounded below by $-\mathcal{H}_{min}$. Similarly, $\overline{\mathcal{P}}$ is bounded below by $-\mathcal{P}_{min}$. Since, all other requirements for convergence remain same, the normalization does not affect the convergence of this controller.

4.4.8 Algorithm Implementation

Algorithm 4 shows the implementation of the decentralized depth controller as described in Section 4.4.5. The procedure receives the depths of all nodes, sensors and robot sensing positions. It also needs the other parameters as input like weight (α), neighborhood size (specified by the variable *hops*), the threshold limit of the objective function difference between two iterations ($threshold_{lim}$), and the minimum number of turns ($turns_{lim}$) for which the difference is objective function value is continuously below $threshold_{lim}$.

Once again, the procedure requires two functions `F(p,q)` and `FDz(p,q)`. These functions take the sensor location or robot waypoint p and the point q in the water column that we want to cover. The first function, `F(p,q)`, computes the covariance between the sensor location and the point of interest. The second function, `FDz(p,q)`, computes the gradient of the covariance function with respect to z at the same pair of points.

In Algorithm 1 discussed in Section 3.3.8, the user specifies the maximum number of iterations for which the the UPDATEDEPTH procedure is run on all the sensors. The user has to know approximately how many iterations it takes for the system to converge. We get rid of this user dependency in our algorithm by introducing two parameters $threshold_{lim}$ and $turns_{lim}$. We know if the algorithm has converged with the help of the these parameters. When the algorithm converges, the value of the objective function in the previous iteration is either greater than, or equal to that of the current iteration. The $threshold_{lim}$ is a user defined limit of the difference in the values of objective function in between two iterations and $turns_{lim}$ is the minimum number of iterations for which the difference in objective function value should be below $threshold_{lim}$. So here, unlike Algorithm 1, we need a calling function, AQUANODE, that calls the function UPDATEDEPTH to update the depth of all the N sensors and M robot points that are present in the system (Line 39) till the algorithm converges.

Another important input parameter to the algorithm is the variable $hops$ which specifies the communication range of the mobile robot. For example, $hops = 1$ implies that the sensors can communicate with one sensor ahead of it and one sensor behind it; similarly $hops = 2$ implies that the sensors can communicate with two sensors ahead of it and two sensors behind it and so on.

By default, $hops = 1$ for Algorithm 4 as the adjacent sensors has to know each others location to determine its own position in the water column. This implies that the location of all the robot waypoints between two sensors is known to either. Thus, when optimizing the position of a particular node, which can be a sensor or a robot waypoint, the location of all the sensors and robot waypoints that this particular node can communicate with will be taken into consideration by the algorithm. So all the sensors and in between robot waypoints in the communication range of a particular node forms the neighborhood of the node (Line 58). All the nodes in this neighborhood affects the location of this node in the water column. Thus the variable $hops$ determines the neighborhood size .

Algorithm 4 AdaptivePath: Mobile Robot Path Planning using *Adaptive Decentralized Algorithm*

Require: Initial location of Sensors and Robot waypoints; weight, α; neighborhood size, *hops*; objective function threshold, $threshold_{lim}$; minimum number of turns, $turns_{lim}$
Ensure: Final location of Sensors and Robot waypoints.

```
 1:
 2: procedure AdaptivePath($\{p_1 \cdots p_N, p_1^R \cdots p_M^R\}$, $\alpha$, $hops$, $threshold_{lim}$, $turns_{lim}$)
 3:     $iteration \leftarrow 0$
 4:     $loopFlag \leftarrow$ TRUE
 5:     while $loopFlag$ do
 6:         $iteration \leftarrow iteration + 1$
 7:         $obj \leftarrow 0$
 8:         for $x = x_{min}$ to $x_{max}$ do
 9:             for $z = z_{min}$ to $z_{max}$ do
10:                 for $i = 1$ to $N$ do
11:                     $obj += \mathrm{F}(p_i,[x,y,z])$
12:                 end for
13:                 for $i = 1$ to $M$ do
14:                     $obj += \mathrm{F}(p_i^R,[x,y,z])$
15:                     $dist += \sum_{j=i-1}^{i+1} dist(p_i^R, p_j^R)$
16:                 end for
17:             end for
18:         end for
19:         $obj^R = (1 - \alpha) * obj + \alpha * dist$
20:         if $iteration \neq 1$ then
21:             $threshold \leftarrow (obj_{prev}^R - obj^R)$
22:             if $threshold > 0$ AND $iteration > threshold_{lim}$ then
23:                 if $turns == 0$ then
24:                     $turns \leftarrow 1$
25:                     $iteration_{prev} \leftarrow iteration$
26:                 else
27:                     if $iteration == iteration_{prev} + 1$ then
28:                         $turns \leftarrow turns + 1$
29:                         $iteration_{prev} \leftarrow iteration$
30:                     else
31:                         $turns \leftarrow 1$
32:                         $iteration_{prev} \leftarrow iteration$
33:                     end if
34:                 end if
35:             end if
36:         end if
37:         $obj_{prev}^R \leftarrow obj^R$
38:         for $i = 1$ to $(N + M)$ do
39:             updateDepth($\{p_1 \cdots p_N, p_1^R \cdots p_M^R\}$, $p_i$, $\alpha$, $hops$)
40:         end for
41:         if $turns \geq turns_{lim}$ then
42:             $loopFlag \leftarrow$ FALSE
43:         end if
44:     end while
45: end procedure
```

Algorithm 4 AdaptivePath: Mobile Robot Path Planning using *Adaptive Decentralized Algorithm* (cont.)

```
46:
47: procedure updateDepth($\{p_1 \cdots p_N, p_1^R \cdots p_M^R\}, q, \alpha, hops$)
48:     $objDz \leftarrow 0$
49:     for $x = x_{min}$ to $x_{max}$ do
50:         for $z = z_{min}$ to $z_{max}$ do
51:             for $i = 1$ to $N$ do
52:                 $objDz+ = \text{Fdz}(p_i, q)$
53:             end for
54:             for $i = 1$ to $M$ do
55:                 $objDz+ = \text{Fdz}(p_i^R, q)$
56:             end for
57:             if $q$ is RobotWaypoint then
58:                 $neighbours \leftarrow$ getNeighbours($\{p_1^R \cdots p_M^R\}, q, hops$)
59:                 $distDz+ = \frac{\sum_{neighbors}(z_q - z_{neighbor})}{\sum_{neighbors} dist(q, p_{neighbor})}$
60:             end if
61:         end for
62:     end for
63:     $objDz^R = (1 - \alpha) * objDz + \alpha * distDz$
64:     $delta = K * objDz^R$
65:     if $delta > maxspeed$ then
66:         $delta = maxspeed$
67:     end if
68:     if $delta < -maxspeed$ then
69:         $delta = -maxspeed$
70:     end if
71:     changeDepth(delta)
72: end procedure
```

In addition to these, the user can also specify a specific area within the water column which is of special interest. Such a region is marked by a different value of covariances. The algorithm is capable of finding a path which takes this factor into consideration. This feature is discussed in detail in Section 5.9.6.

Apart from the above mentioned modifications, the main changes in this algorithm from Algorithm 1 are:

1. Line 55 which adds the sensing contribution from the robot sensing locations. This is applied both when computing the control for the sensor nodes and for the robot sensing locations.

2. Line 59 which calculates the path length of the mobile robot. This is applied only for computing the control for the robot waypoints.

3. Line 63 is added only when control is being calculated for the robot sensing locations. This adds in the component of the controller related to the distance between robot sensing locations.

After the procedure computes the numeric integral, it computes the change in the desired depth. This change is bounded by the maximum speed the node can travel. Finally, the procedure changes the node depth. `changeDepth` puts a hard limit on the range of the nodes, preventing the nodes from moving out of the water column.

Each sensor node is responsible for computing and updating the robot sensing locations p^R that are closest to it. When the robot comes into the communication range of a node, it will transmit the next sensing locations to the robot.

In practice, the sensor network may have a set schedule of times when the robot enters the network, or the robot might inform nearby nodes when it enters the network. These nodes will then start running the depth adjustment algorithm to compute the best path for the robot and any corresponding changes in the depths for the sensor nodes. This will propagate through the network, and as the robot moves, it will receive updated way points from nearby sensor nodes.

The simulation experiments and analysis of mobile robot path obtained by this method are discussed in Section 5.4.

4.5 Summary

In this Section, we introduce three different path planning algorithms for planning the path of a mobile underwater robot through a network of semi-mobile sensors. The goal of the algorithms is to find sensing locations such that the overall sensing of the water column using the network is increased. The path planning algorithms are divided into three different categories: 1) Global Path Planning with VoronoiPath method, 2) Local Path Planning with the help of TanBugPath method, and 3) Adaptive path Planning with the AdaptivePath method. In the next chapter (Chapter 5), we simulate all the algorithms in Matlab. Then we perform experiments by varying different parameters to study their performance. Finally, we compare the paths planned by the three algorithms with respect to a particular sensor network.

Chapter 5

Adaptive Sampling for Underwater Mobile Robots: Simulations & Experiments

5.1 Introduction

In Chapter 4, we introduced three different algorithms that optimize the path of a mobile underwater robots through a network of sensors with the goal of improving sensing. Several practical considerations arise in implementing these algorithms in simulation. Since our goal is to implement these algorithms on real hardware like in [20, 25], we model the sensors and the robot taking these limitations in to consideration. In this chapter, we explore the performance of the different algorithms in simulation and present the results. We discuss parameter sensitivity, positioning sensitivity, and comparison between the different methods.

5.2 Practical Considerations

We must take into account a number of practical considerations that are not in the theory developed in Chapter 4 before implementing the algorithms in simulation. These are:

1. In the simulations and experiments we consider the exact location of the sensors and robot waypoints. However, in the real world, it is not possible for the algorithm to know the exact location of all the sensors. We use MATLAB's `rand` command to implement this uncertainty in the simulation experiments.

2. The acoustic communication has limited bandwidth and messages are transmitted infrequently, so we must limit the amount of transmitted data.

3. In case of TANBUGPATH algorithm, even though at the beginning of the simulation the location and depth of every sensor is input to the algorithm, during run time the algorithm gets the location and depth information of only the nearest sensor.

4. Similarly, in case of Adaptive Decentralized method, the algorithm gets as input the location and depth of every sensor and robot waypoint. However, during run time, the location and depth of only the nearest neighbors are known to the algorithm. This is taken care of by the $hops$ parameter.

5. In case of the modified adaptive decentralized algorithm, the controller is continuously integrated over an area Q. The region must discretized for numeric integration. There are two factors which affect how the region is discretized:

   - Desired sensing accuracy
   - Computational complexity

   The algorithm will not differentiate between different configurations if the discretization is very rough. However, if it is very fine, the computation takes a very long time to converge.

We implemented our simulations with these parameters and show in this chapter that the algorithms still work with these.

5.3 Global Knowledge vs. Local Knowledge

An important concept in the case of the path planning algorithm is if it needs global knowledge or local knowledge about the location of the sensor nodes. This factor depends upon how much information is needed by an algorithm before it starts planning the path of the mobile robot in the water column. A global knowledge algorithm (as in the case of VoronoiPath algorithm) implies that the information about the location of all the nodes should be available before hand. On the other hand, local knowledge implies that the algorithm requires the knowledge about the location of only the nearest (in case of the TanBugPath algorithm) or the neighboring nodes (in case of the AdaptivePath algorithm). The cost of forwarding depth information (required for the algorithm) over multiple hops in the network is difficult over a limited acoustic bandwidth, hence a local path planing algorithm is comparatively more efficient.

For the VoronoiPath algorithm, global knowledge is needed. This means that before the robot is released in the water column, the algorithm should have knowledge about the location of all the sensor nodes so that it can plan ahead which points in the water column the mobile robot needs to visit to maximize information gain. In practice, the robot does not need to know the entire network of the sensors to decide its position. For example, if there are ten sensors in the network and if the robot is just entering the water column, then the robot does not have to know the position of the tenth node immediately. Instead, the algorithm can just know the location of the nearest subset of nodes and plan the robot's immediate path while the rest of the sensor nodes' locations are gradually made available to the algorithm.

In case of the TanBugPath algorithm, even though during the simulation the algorithm is supplied with all the sensor locations at the beginning, it uses location of only the next nearest sensor to calculate the immediate path of the mobile robot. Once the robot is close to this sensor, the location of the next sensor is required to plan the rest of the path. This is done to imitate the real world situation when the location of the next sensor is supplied to the robot by the nearest sensor. Therefore, this algorithm needs local knowledge.

For the AdaptivePath algorithm, each sensor knows only the positions of its neighbors and neighboring waypoints. This assumption is okay in case of a real world deployment of the sensors and a robot because the effect of far away sensors is minimal as the Gaussian covariance function decays rapidly with distance. As such, the algorithm can ignore the effect of sensors whose location it does not know.

5.4 Simulation Setup

We simulated the different algorithms in Matlab to test their performance. In these experiments, unless otherwise stated:

- The sensors are placed in a line spaced 15 meters apart from each other.
- The water column is of depth 30 meters.

In case of the AdapativePath algorithm, we assume the following:

- An 1 meter grid is used to integrate over for all operations.
- The base Gaussian covariance function described in Section 3.3.5 with and having $\sigma_s = 5$ and $\sigma_d = 4$, unless otherwise stated.
- The maximum speed of the sensors and robot waypoints is capped at 2 meters/second.
- The value of k is assumed to be $k = 4000$, unless otherwise stated.
- The values of $threshold_{lim} = 0.00001$ and $turns_{lim} = 5$ are assumed, unless otherwise stated.

In Section 5.7, we first analyze the simulation experiments with the VoronoiPath algorithm. In Section 5.8, we discuss the path planning with the Tangent Bug method.

In (Section 5.9) we discuss the parameter sensitivity, positioning sensitivity, and data reconstruction with the AdaptivePath algorithm. And finally in Section 5.10, we compare the different methods to each other.

5.5 Posterior Error

A common metric for defining how well an area is covered by sensors is to examine the posterior error of the system [35]. Calculating the posterior error requires the system to be modeled as a Gaussian process. This is a fairly general model and valid in many setups. The posterior error of a point can be calculated as:

$$\sigma^2_{q|P} = Cov(q,q) - \Sigma_{q,P} \cdot \Sigma^{-1}_{P,P} \cdot \Sigma_{P,q} \tag{5.1}$$

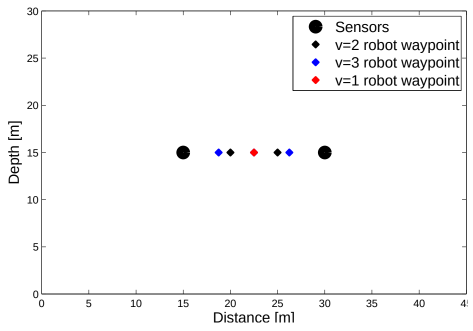
The vector $\Sigma_{q,P}$ is the vector of covariances between q and the sensor node positions $P = \{p_1, ..., p_N\}$. The vector $\Sigma_{P,q}$ is $\Sigma_{q,P}$ transposed. The matrix $\Sigma_{P,P}$ is the covariance matrix for the sensor node positions. The values of $\Sigma_{P,P}$ are $\Sigma_{p_i,p_j} = Cov(p_i, p_j)$ for each entry (i, j).

This computation requires an inversion of the full covariance matrix which is impractical on real sensor network hardware that has limited computation power and memory [20]. So, we use posterior error calculation as a metric to evaluate the performance of the algorithms discussed in this thesis. Throughout the experiments section we will be using posterior error for this purpose.

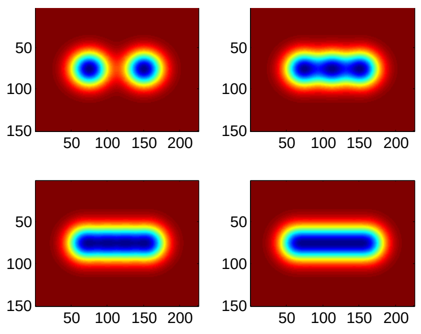
5.5.1 Manual Experiments with Posterior Error

In this section, we perform a simple experiment with different layout of sensors to show that the posterior error calculations work as expected. We consider a very simple sensor layout, shown in Figure 5.1(a), with only two sensors which are 15 meters apart from each other. We test with four different layouts:

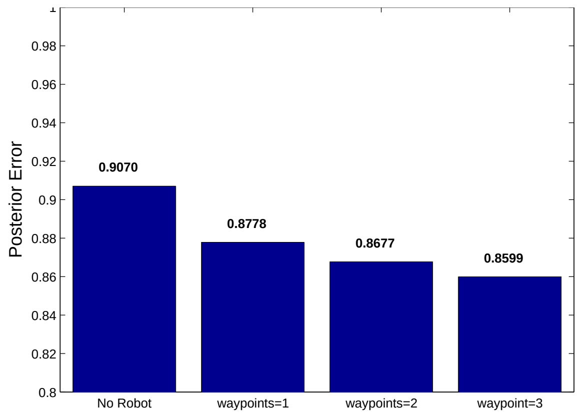
- Only the two sensors (represented by the black dots)



(a) Simple Sensor Layout



(b) Coverage



(c) Posterior Error Values

Figure 5.1: Experiments with Posterior Error

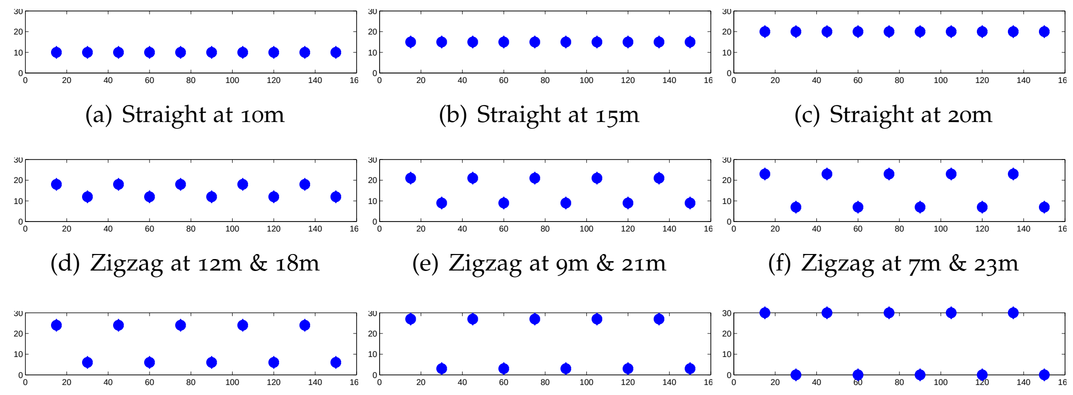
- Adding 1 sensor between the two sensors (represented by the red diamond)
- Adding 2 sensors between the two sensors (represented by black diamonds)
- Adding 3 sensors between the two sensors (represented by red and blue diamonds)

The sensor coverage for the four layouts can be seen in Figure 5.1(b) shown in a clockwise manner. The posterior error value for layout is shown in Figure 5.1(c). As we can see, adding more number of sensors covers more area of the water column and hence the posterior error value decreases. We performed similar experiments for different layouts and found the same result. However, the value of the posterior error depends on the area covered. So, if the same number of sensors gives a better coverage of the area of water column than another layout, the posterior error value for the first layout will be lesser than that of the later one.

5.6 Manual Experiments with Sensor Layout

In this section, we performed some manual experiments using several layout of sensors and robot way points. In Section 5.6.1. we study different sensor layout in terms of the posterior error calculation for each layout. In Section 5.6.3, we keep the sensor layout fixed and introduce several robot way points to study the different layouts in terms of the posterior error values. These experiments are performed to get a better idea about the layout of sensor and robot waypoints. These experiments helped to understand the performance of the different algorithms described previously.

5.6.1 Experiments on Sensor layout



(a) Straight at 10m (b) Straight at 15m (c) Straight at 20m

(d) Zigzag at 12m & 18m (e) Zigzag at 9m & 21m (f) Zigzag at 7m & 23m

(g) Zigzag at 6m & 24m (h) Zigzag at 3m & 27m (i) Zigzag at 0m & 30m

Figure 5.2: Plots showing the different layouts for Sensors ($R = 10$) to compare the posterior error for each layout

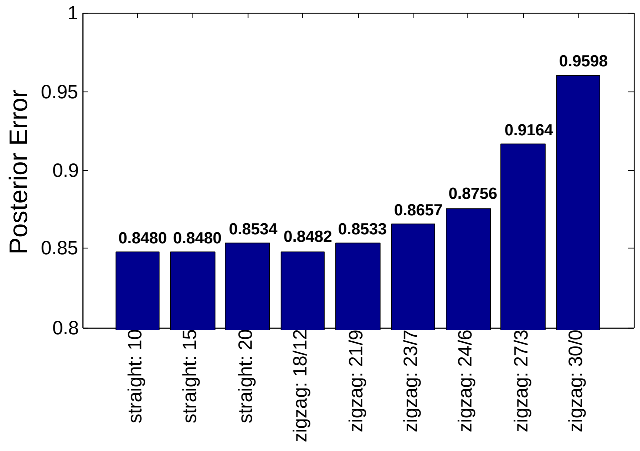
In this section, we describe the experiments performed on different layouts with only sensors. For this set of experiments, we took a set of ten sensor nodes and manually arranged them in different layouts along the depth of the water column (Fig. 5.2). For each of the layouts shown in Figure 5.2, we found the posterior error for different values of σ_s and σ_d. The posterior error values for the different layouts for $\sigma_s = 5$ and $\sigma_d = 4$ is shown in Figure 5.3(a), for $\sigma_s = 10$ and $\sigma_d = 4$ is shown in Figure 5.3(b), and for $\sigma_s = 5$ and $\sigma_d = 10$ is shown in Figure 5.3(c), respectively.

The parameter σ_s determines to what extent the sensor can detect information in the horizontal direction or along the surface (hence, the subscript 's') of the water column. A larger value of σ_s implies it is able to sense at larger distances from its current position. Similarly, σ_d determines to what extent the sensor can detect information in the vertical direction or along the depth of the water column. As we can see, both σ_s and σ_d affect the coverage of the water column. If we keep σ_d constant and double σ_s, the posterior error value decreases for layouts in which the sensors are far part from each other, preferably in a zigzagged arrangement. On the other hand, keeping σ_s constant and doubling σ_d, we find the posterior error value decreases for layouts in which the sensors are placed close to each other, often in a colinear arrangement. For any value of σ_s and σ_d, the layouts in Figure 5.2(d) and Figure 5.2(f) seem to have better coverage in terms of posterior error, whereas the layout shown in Figure 5.2(i) has the least favorable coverage.

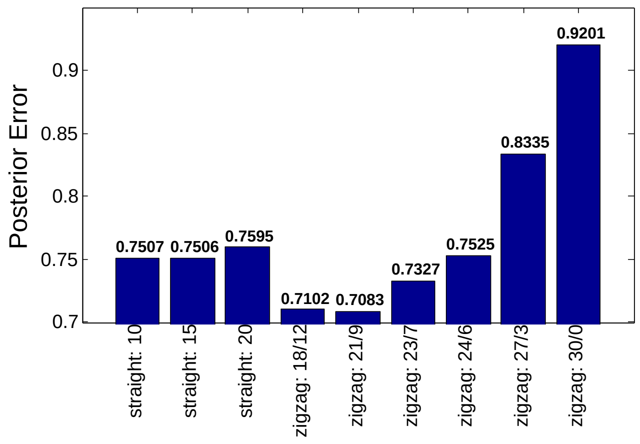
5.6.2 Experiments on number of Robot Waypoints in between Sensors

In this section, we describe the experiments with different layouts having fixed sensor positions, but different number of robot waypoints in between the sensors. We study a total of three configurations. For the sensor layout we have chosen a zigzagged configuration with sensors alternately at 25 meters and 5 meters respectively. For all the three configurations we start with the robot waypoints at the center of the water column at 15 meters. The different configurations are shown in Figure 5.4. We calculated the posterior error values for each of these configurations for different values of σ_s and σ_d. The values are shown in Figure 5.5.

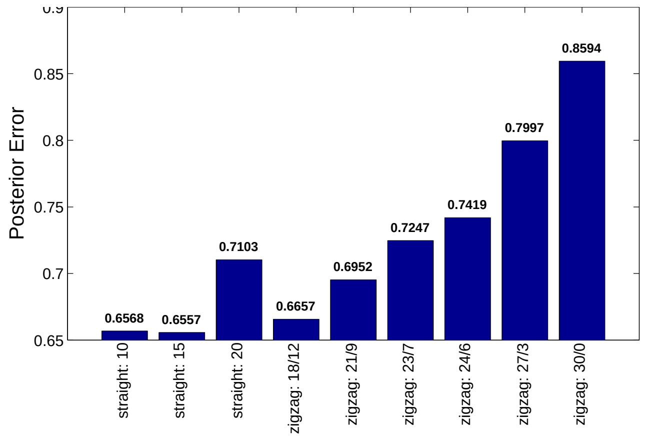
The posterior error values for the different layouts for $\sigma_s = 5$ and $\sigma_d = 4$ is shown in Figure 5.5(a), for $\sigma_s = 10$ and $\sigma_d = 4$ is shown in Figure 5.5(b), and for $\sigma_s = 5$ and $\sigma_d = 10$ is shown in Figure 5.5(c) respectively. From the figures, we can observe that the layouts where there are more robot waypoints have smaller posterior error value for any



(a) Posterior Error ($\sigma_s, \sigma_d = 5, 4$)

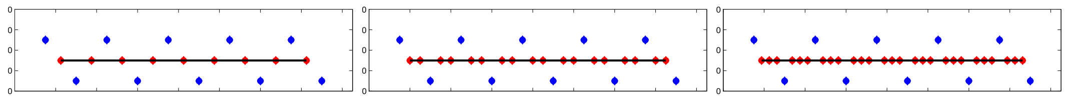


(b) Posterior Error ($\sigma_s, \sigma_d = 10, 4$)



(c) Posterior Error ($\sigma_s, \sigma_d = 5, 10$)

Figure 5.3: Comparison of Posterior Error for layouts shown in Figure 5.2



(a) Straight at 15m with $V = 1$ (b) Straight at 15m with $V = 2$ (c) Straight at 15m with $V = 3$

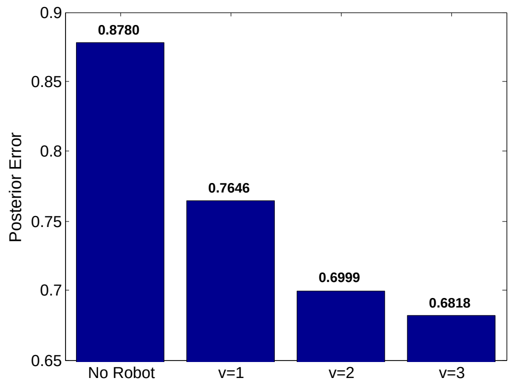
Figure 5.4: Plots showing the different layouts for Sensors ($R = 10$) with different intermediate robot waypoint $V = [1, 2, 3]$ between each sensor to compare the posterior error for each layout

value of σ_s and σ_d. However, the layout which has higher value of surface covariance σ_s (Figure 5.5(b)), the gradient of decrease in posterior error value with increasing number of robot waypoints is much less steep than for lesser values of σ_s. This is because larger σ_s creates greater correlation between sensors and they sense increased areas of overlapping space.

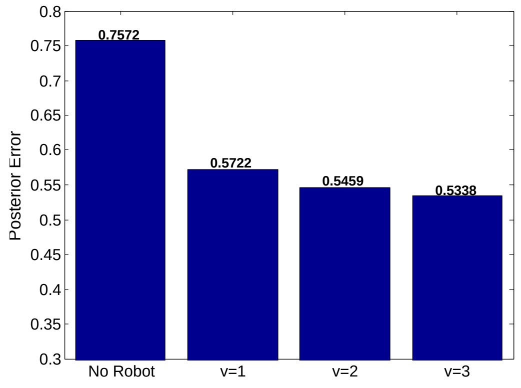
5.6.3 Experiments on layout of Mobile Robot Path

In this section we describe the experiments with different layouts having fixed sensor positions but varying positions of robot waypoints. We study a total of six configurations. For the sensor layout we have chosen a zigzagged configuration with sensors alternately at 25 meters and 5 meters respectively. For the six configurations we start with all the robot waypoints at the center and gradually increase the distance till they are placed on the edges of the water column. The different configurations are shown in Figure 5.6. We calculated the posterior error values for each of these configurations for different values of σ_s and σ_d. The values are shown in Figure 5.7.

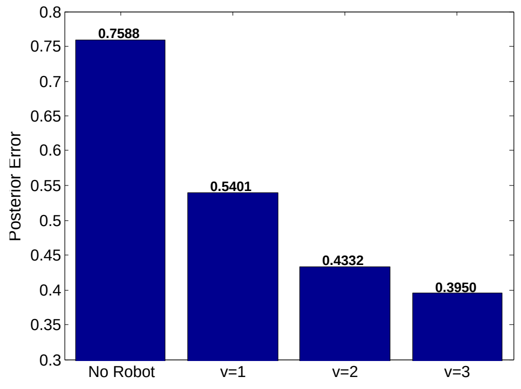
The posterior error value for the different layouts for $\sigma_s = 5$ and $\sigma_d = 4$ is shown in Figure 5.7(a), for $\sigma_s = 10$ and $\sigma_d = 4$ is shown in Figure 5.7(b), and for $\sigma_s = 5$ and $\sigma_d = 10$ is shown in Figure 5.7(c) respectively. From the figures, we can see that all of them follow the same pattern. The layouts where the sensors and the robot waypoints are far apart from each other perform better than the ones in which they are placed close together. For the next set of experiments we mainly chose $\sigma_s = 5$ and $\sigma_d = 4$ unless otherwise specified. These values were chosen for three main reasons:



(a) $\sigma_s = 5, \sigma_d = 4$



(b) $\sigma_s = 10, \sigma_d = 4$



(c) $\sigma_s = 5, \sigma_d = 10$

Figure 5.5: Comparison of Posterior Error for layouts shown in Figure 5.4

- If we limit the area of sensing then it implies that we can use less powerful sensors and robots;
- By decreasing the area of sensing, the interference between adjacent sensors and

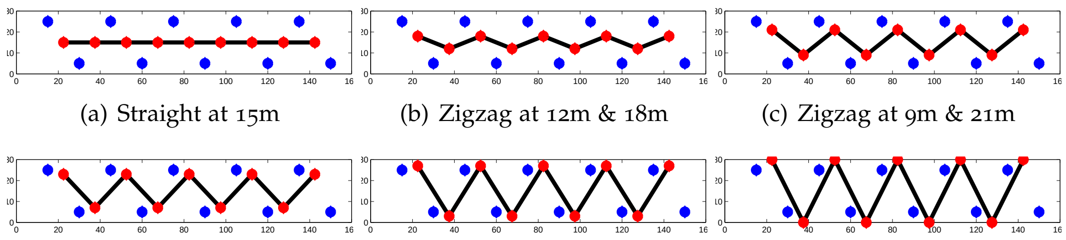

(a) Straight at 15m (b) Zigzag at 12m & 18m (c) Zigzag at 9m & 21m (d) Zigzag at 7m & 23m (e) Zigzag at 3m & 27m (f) Zigzag at 0m & 30m

Figure 5.6: Plots showing the different layouts for Sensors ($R = 10$) and one intermediate robot waypoint between each sensor to compare the posterior error for each layout

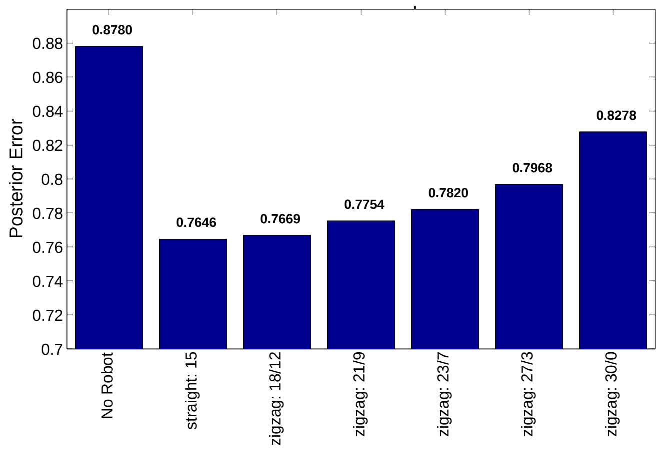
robot waypoints is minimized;

- If all the sensors and robot waypoints sense from a smaller area, the algorithm will distribute them such that greater area is covered to cover the total area of the water column. Then the effect of changing other parameters can be studied better.

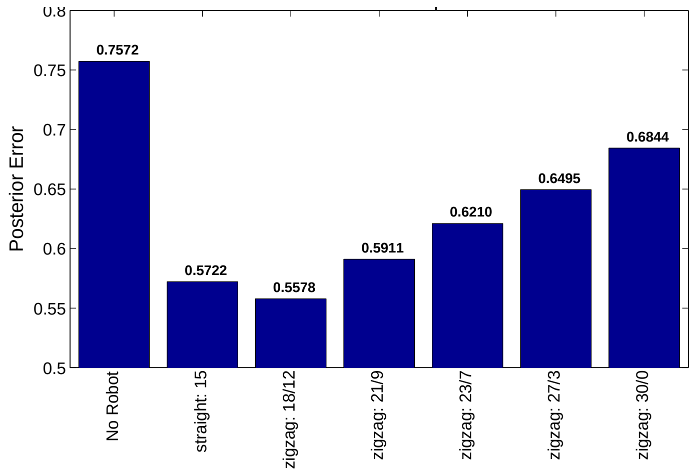
5.7 Experiments on Voronoi Path Algorithm

The VoronoiPath is the global path planning algorithm that places the robot path points at the position of least influence by the neighboring sensors. Due to this, the distribution of the sensors and the robot way point in the water column obtained after running this algorithm should be the best possible scenario. We perform the experiments on the VoronoiPath algorithm in three different ways.

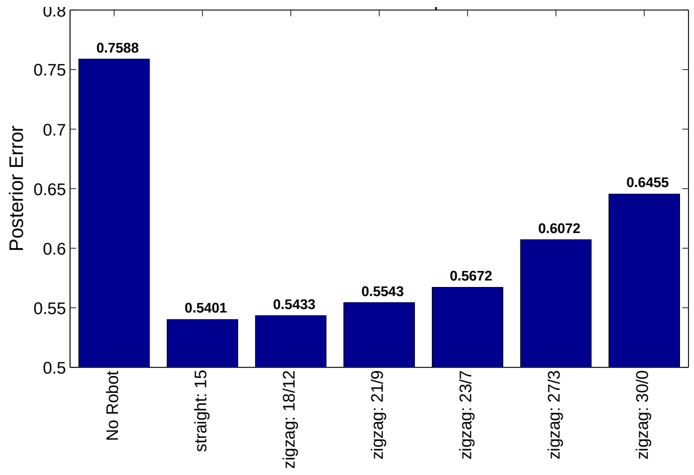
In the first layout, we manually placed 10 sensors in a zigzagged fashion at the edges of the water column, i.e., some are at depth 0 meters while the alternate ones are at 30 meters. By placing the sensors at the edges, we are sacrificing half of the sensing by these sensors. So, this is not an ideal position. Then the path of a mobile robot is planned with the help of Algorithm 1. The posterior error of the system is calculated first without and then with the robot path. Figure 5.8 shows us the layout of the mobile robot path obtained from this algorithm and Figure 5.13 shows us the comparison of the posterior error obtained when posterior error is calculated with only the sensors in the system and when it is calculated



(a) $\sigma_s = 5, \sigma_d = 4$



(b) $\sigma_s = 10, \sigma_d = 4$



(c) $\sigma_s = 5, \sigma_d = 10$

Figure 5.7: Comparison of Posterior Error for layouts shown in Figure 5.6

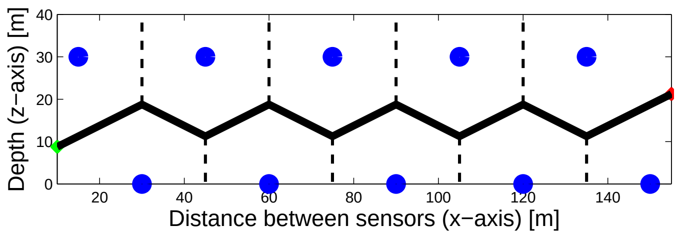


Figure 5.8: Layout 1: Plots showing the sensor layout and mobile robot path for Layout with sensors placed zigzaggedly at the edge of the water column

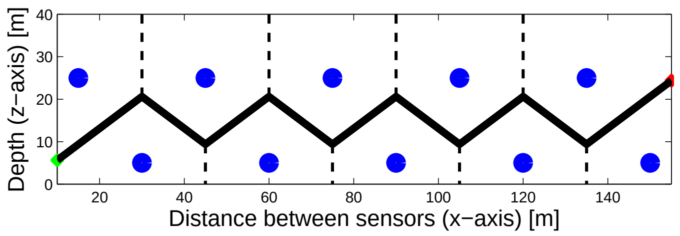


Figure 5.9: Layout 2: Plots showing the sensor layout and mobile robot path for Layout with nodes placed zigzaggedly at 5m and 25m in the water column

by adding the Voronoi vertices and intermediate path points to the system. We found that the posterior error is 0.9365 when there are no mobile robot path and 0.8097 when the mobile robot passes through the sensors. As we can see from this figure, the posterior error of the system decreases on adding the mobile robot path. Therefore, it means that introducing the mobile robot improves the overall sensing in the water column as a lower value of the posterior error implies better sensing.

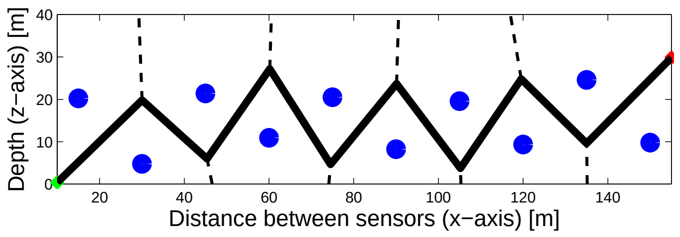


Figure 5.10: Layout 3: Plots showing the sensor layout and mobile robot path for a-priori deployed realistic Sensor Network

In the second layout, we again manually placed 10 sensors in a zigzagged fashion at depth 5 meters and the alternate ones are at 25 meters. This placement of the sensors should give us a better reading than the one obtained by placing the sensors at the edges.

Similar to Layout 1, the path isplanned with the help of Algorithm 1. The posterior error of the system is calculated first without and then with the robot path. Figure 5.9 shows us the layout of the mobile robot path obtained from this algorithm and Figure 5.13 shows us the comparison of the posterior error obtained when posterior error is calculated with only the sensors in the system and when it is calculated by adding the Voronoi vertices and intermediate path points to the system. We found that the posterior error is still 0.8780 when there are no mobile robot path but it decreases to 0.7519 when the mobile robot passes through the sensors. Since a lower value of the posterior error implies a better sensing. It can be inferred that sensing is improved in Layout 2 compared to Layout 1.

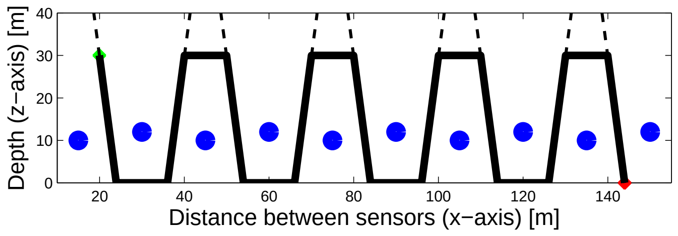
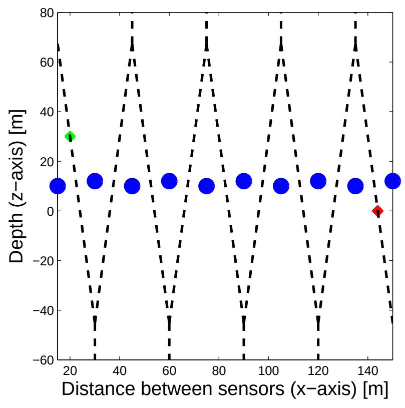
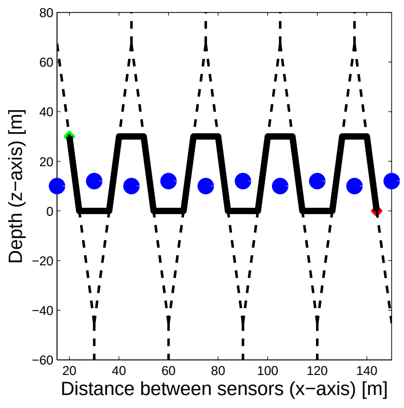


Figure 5.11: Layout 4: Plots showing the sensor layout and mobile robot path for Layout with sensor locations in a zigzagged arrangement but very close to each other

In this layout, we took 10 sensors, placed them all at depth 10 meters and then optimized the location of these sensors by using Algorithm 4 without intermediate robot waypoints. Then we input the optimized locations of these sensors to Algorithm 1. Figure 5.10 shows us the layout of the mobile robot path obtained from this algorithm and Figure 5.13 shows us the comparison of the posterior error obtained when posterior error is calculated with only the sensors in the system and when it is calculated by adding the Voronoi vertices and intermediate path points to the system as well. As we can see from this figure, the posterior error of the system without the mobile robot path is 0.8741 and with the mobile robot path is 0.7638. The value of the posterior error with the mobile robot path is not less than Layout 2 even though it was expected as this time the layout of the sensors was optimized with Algorithm 4. This is because the location of the sensors is optimized such that it provides the best possible sensing with only the 10 nodes in the system. It does not consider if a mobile robot is introduced in the system later and thus the algorithm does not readjust the positions of the sensors in the wake of the presence of the additional mobile robot. This is taken care of in Algorithm 4.



(a) Voronoi Cells



(b) Voronoi Path after Truncation

Figure 5.12: Plots enlarging Figure 5.11

Another important problem we face when we are using this method is if all the sensors are collinear, i.e., they all lie in the same straight line. In this case, the algorithm is unable to compute the Delaunay triangulation. However, we do not need to worry much about this case as the sensor locations obtained after running Algorithm 4 is mostly arranged in a zigzagged fashion and almost never in a straight line.

Table 5.1: Posterior Error Comparison for Voronoi Tesselation method of Different layouts described in Section 5.7

Layout #	Only Sensors	Sensors + Robot Path
1	0.9365	0.8097
2	0.8780	0.7519
3	0.8741	0.7638
4	0.8733	0.7615

Figure 5.11 shows the path when the sensors are very close to one another (alternately at 10 meters and 12 meters respectively). From the figure, we can see that the mobile robot path does not connect the Voronoi vertices for this layout.

When we expand the field of view (Figure 5.12(a)), we can see that the Voronoi vertices

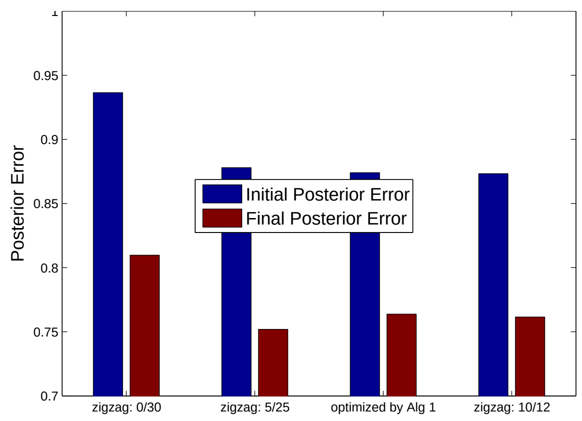


Figure 5.13: Posterior Error Comparison for Different layouts for the Voronoi Tesselation method

for this arrangement of sensors do not lie within the depth of the water column. So the Voronoi path needs to be truncated at the boundary of the water column so that the robot does not escape the water (Figure 5.12(b)). When only a few edges needs to be truncated, the planned path can be very similar to the ideal Voronoi path. However, in this case, since all the Voronoi vertices lie outside the region of interest, all the edges needed to be truncated at the boundary. For simplicity, when the Voronoi edges are truncated, the robot travels in a straight line across the surface, or along the bottom, of the water column, as applicable. In Figure 5.13 we can observe that the posterior error for the layout without any mobile robot is 0.8733 and with the mobile robot path is 0.7615. Adding more sensor locations covers more region in the water column, and as a result, the posterior error still decreases.

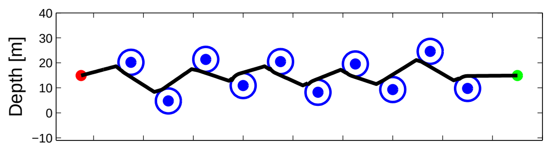
5.8 Experiments on Tangent Bug Path algorithm

The Algorithm 3 is the path planning algorithm inspired from Tangent Bug method. In Algorithm 2, the mobile robot senses the water column at previously determined points and the path of the mobile robot is planned by connecting these sensing points together. It is possible as the location of the sensors is known before hand. Here, the algorithm knows the location of the sensor right ahead of the robot only. Thus, this is is a local path planning algorithm. The main parameters that affect the planned path are:

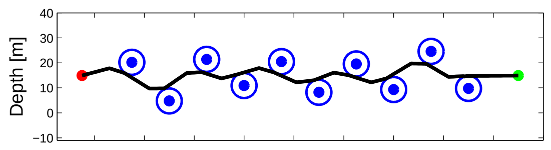
- Sensing radius of the robot, r_R
- Sensing radius of the sensors, r_S
- Interval between each decision, *interval*

The other important factor is the layout of the sensors. We must note that r_R and r_S are the properties of the robots. Whereas, *interval* is a parameter for the algorithm and the layout is a completely external factor affecting the path of the mobile robot. If the sensors and the mobile robot are simlar to each other in characteristics, then we might have $r_R = r_S$.

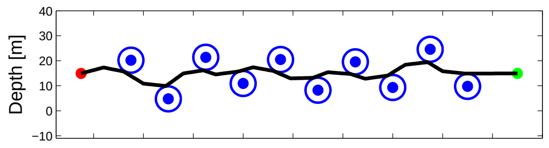
5.8.1 Changing Robot's Sensing Radius



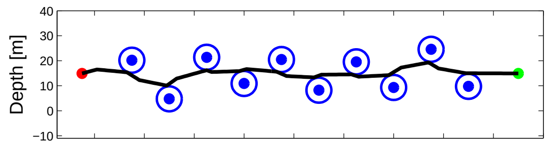
(a) Robot's Sensing Radius = 2



(b) Robot's Sensing Radius = 5



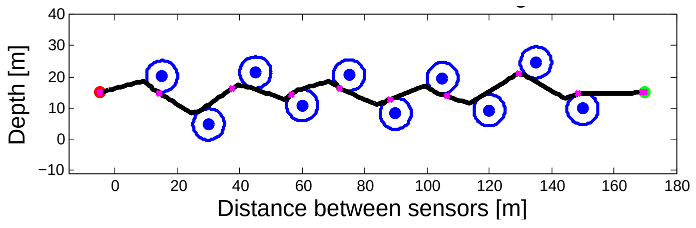
(c) Robot's Sensing Radius = 7



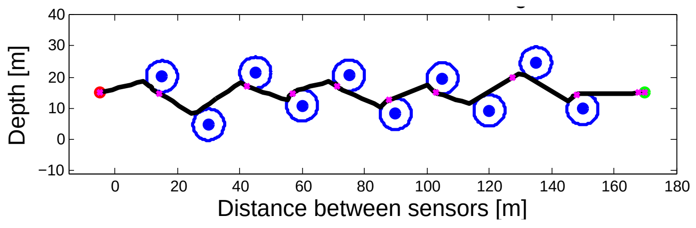
(d) Robot's Sensing Radius = 10

Figure 5.14: Path Planning with Tangent Bug algorithm for different r_R of the robot with $interval = 1$ and sensor's sensing radius, $r_S = 5$

In the following sections, the effect of above mentioned four parameters are examined by keeping the other factors constant. In Section 5.8.1, we examine the effect of changing r_R for $interval = 1$ keeping everything else constant. In Section 5.8.2, we perform the same set of experiments as Section 5.8.1 but this time we have $interval = 5$. We compare the results with the previous section. In Section 5.8.3, the value of r_S is changed keeping r_R constant at 10 meters and $interval = 1$. For these experiments, instead of choosing manual layouts, we use the same layout of the sensors as was used in Section 5.7 for the layout in which the position of the sensor network is optimized by Algorithm 4 without intermediate robot waypoints. In Section 5.8.4, we examine the performance of the algorithm under other layouts, speciffically we study the behavior when the sensors are placed close to each other.



(a) interval = 1



(b) interval = 5

Figure 5.15: Plots showing the sensing positions for a mobile robot path having $r_R = 2$ and $r_S = 5$

In this section, we vary the sensing radius r_R of the robots keeping the sensing radius of the sensor constant ($r_S = 5$) and $interval = 1$. The values of r_R are 2 meters, 5 meters, 7 meters, and 10 meters, respectively. These values were randomly chosen to cover the entire spectrum of ranges. As the horizontal difference between any two sensors is 15 meters, a value of r_R which is greater than 10 meters is not chosen as the path planning will not be efficient. Figure 5.14 shows the different layouts and the posterior errors for the different layouts when the mobile robot path is absent and present. From the figures, we can conclude that a smaller value of the r_R gives a greater control on the planned path. Since the robot cannot see a great distance ahead of it, it takes a longer time to decide where it should go next. For example, there is more sensor boundary following; or the mobile robot path intersects the sensor boundary even when the next sensor is clearly visible. So a path planned with a lower value of r_R is more detailed than a path planned with a higher value.

One important consideration with the Tangent Bug algorithm is that the mobile robot path has a large number of path points, N_{path}. This number depends on the parameter *interval* which is specified by the user. This is discussed in detail in the Section 5.8.2. However, the mobile robots are constrained and it is not possible to sense the water column continuously. Therefore, we assume the robot is able to sense at only a certain number of points N_{sense} in the water column. Since there are ten sensors in the network, we assume that $N_{sense} = 9$. This number is chosen to maintain consistency with Section 5.9. To keep it simple, we assume that the mobile robot can sense only at every $floor(\frac{N_{path}}{N_{sense}})^{th}$ point on its path.

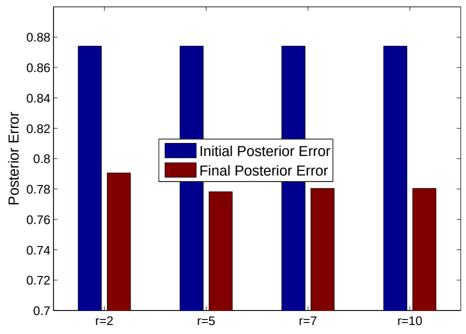


Figure 5.16: Posterior Error Comparison for Figure 5.14

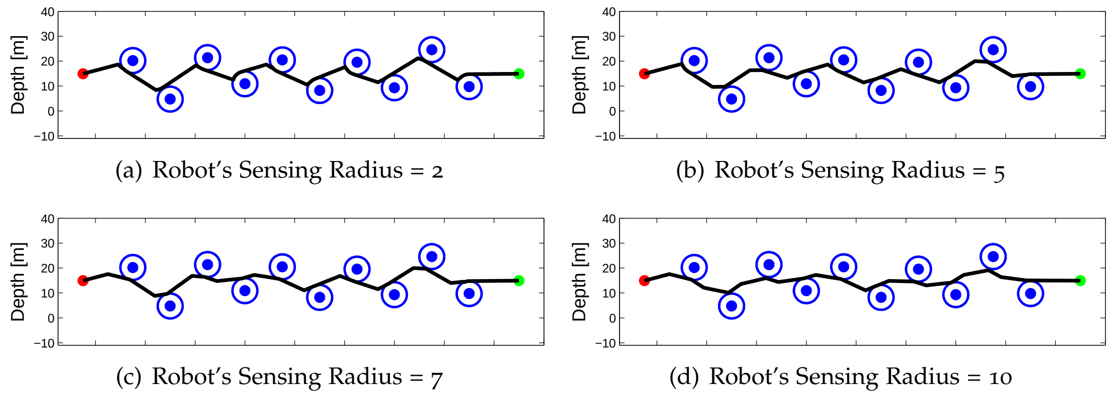
Table 5.2: Posterior Error Comparison for Tangent Bug method of Different layouts described in Section 5.8.1

Layout #	Only Sensors	Sensors + Robot Path
1	0.8741	0.7906
2	0.8741	0.7782
3	0.8741	0.7804
4	0.8741	0.7804

Figure 5.15(a) shows the sensing locations marked in magenta color for the Figure 5.14(a). Due to the nature of the algorithm, these points can lie anywhere. Instead of choosing a point which is in between two sensors, a point on the sensing boundary can be chosen. Therefore, the posterior error calculation will depend on which points are chosen to be the sensing location. This can be improved if a new algorithm is introduced which chooses strategically important sensing locations. However, for simplicity this was not implemented in the scope of this thesis.

5.8.2 Changing Size of Interval

In this section we will study what happens when we change the interval.



(a) Robot's Sensing Radius = 2
(b) Robot's Sensing Radius = 5
(c) Robot's Sensing Radius = 7
(d) Robot's Sensing Radius = 10

Figure 5.17: Path Planning with Tangent Bug algorithm for different r_R of the robot with $interval = 5$ and sensor's sensing radius, $r_S = 5$

Table 5.2 shows the posterior error values obtained for the different layouts first without the mobile robot path and then with the mobile robot path. In all the cases, the addition of the mobile robot path improves sensing of the region. We see that $r_R = 5$ gives the least value of posterior error implying that it is the best value and $r_R = 2$ gives the maximum value of posterior error implying it is the least favorable. However, since these values are dependent on the number and position of the sensing locations, these conclusions are not rigid.

There are a few important consideration in determining the *interval* size. They are as follows:

- The interval has to be a positive number. It controls the number of iterations needed to go from the p_{start} to p_{end} in the robot path. The smaller the number, the larger the number of iterations, which in turn increases the time taken for the algorithm.

- If $interval > r_R$, then we set $interval = r_R$. This is because if the robot cannot see distance beyond r_R, it will not be able to position itself *interval* ahead of its current position and the algorithm will fail.

In this section, we vary the sensing radius r_R of the robots keeping the sensing radius of the sensor constant ($r_S = 5$) and $interval = 5$. The different values of r_R chosen are 2 meters, 5 meters, 7 meters, and 10 meters to maintain consistency with the previous section. Figure 5.17 shows the different layouts and the posterior errors for the different layouts when the mobile robot path is absent and present.

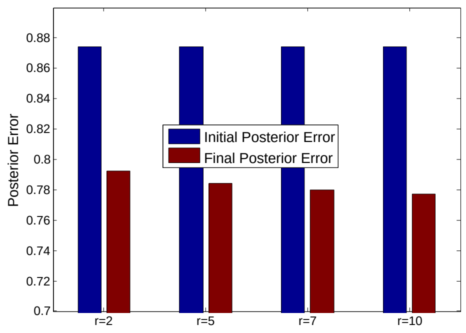


Figure 5.18: Posterior Error Comparison for Figure 5.17

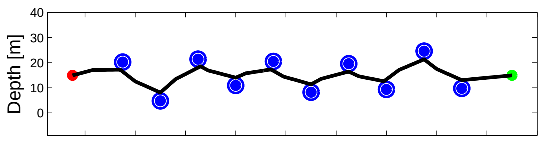
Table 5.3: Posterior Error Comparison for Tangent Bug method of Different layouts described in Section 5.8.2

Layout #	Only Sensors	Sensors + Robot Path
1	0.8741	0.7922
2	0.8741	0.7843
3	0.8741	0.7800
4	0.8741	0.7772

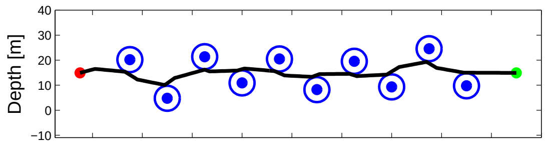
From the figures we can see that, similar to Section 5.8.2, a smaller value of the r_R gives a greater control on the planned path, there is more sensor boundary following etc. However, in this case we get $\frac{N_{path}}{5}$ path points instead of N_{path}. Figure 5.15(b) shows the sensing locations for $r_R = 2$ in magenta. As we can observe, these points are different than the ones chosen for $interval = 1$.

5.8.3 Changing Sensor's Sensing Radius

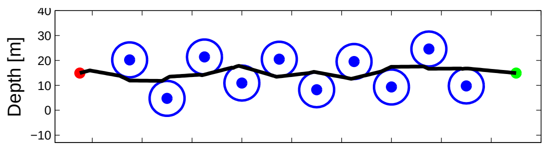
Table 5.3 shows the posterior error values obtained for the different layouts first without the mobile robot path and then with the mobile robot path. We observe that for $interval = 5$, the mobile robot path planned with $r_R = 2$ gives the worst performance and that with $r_R = 10$ gives the best performance. However, since these data are based on the randomly selected path points on the mobile robot path we may get a different result by changing



(a) Sensor's Sensing Radius = 2



(b) Sensor's Sensing Radius = 5



(c) Sensor's Sensing Radius = 7

Figure 5.19: Different r_S of the sensors; $interval = 1$ and robot's sensing radius, $r_R = 10$

the positions of these points.

Table 5.4: Posterior Error Comparison for TanBugPath algorithm for Different layouts described in Section 5.8.3

Layout #	Only Sensors	Sensors + Robot Path
1	0.8741	0.7888
2	0.8741	0.7805
3	0.8741	0.7790

In this section we vary the sensing radius r_S of the sensors keeping the sensing radius of the mobile robot constant ($r_R = 10$) and $interval = 1$. The different values of r_S that are chosen are 3 meters, 5 meters, and 7 meters respectively. A value which is smaller than 3 meters will bring the mobile robot too close to the sensor. Intuitively, such a value may be useful to plan a path when the mobile robots are very close to one another but not so much otherwise. A value of sensing radius greater than 7 meters implies that the sensing regions of the adjacent nodes overlap. This is because the horizontal distance between any two sensors is 15 meters. In such case, the sensors together become one big obstacle and the algorithm is unable to find an efficient path through the network. Thus, the above mentioned values were chosen. Figure 5.19 shows the different layouts and the posterior errors when the mobile robot path is absent and present.

The comparison between the different layouts with respect to the posterior error calculated in each scenario can be seen in Table 5.4 and is represented in Figure 5.20. As discussed earlier, we assume that the mobile robot can only sense at nine locations through the entire water column and hence it senses at every $floor(\frac{N_{path}}{9})^{th}$ location. When the r_S is smaller, the mobile robot path is zigzagged, but when r_S is increased, the path becomes more like a straight line. Even though the sensing locations are randomly chosen, from the figures and the posterior error values, $r_S = 10$ seems to perform better than $r_S = 5$ which in turn performs better than $r_S = 2$. When a path point is chosen on the sensing boundary with a lower value of r_S, the sensor and the robot are very close together and their sensing bubble collides. This gives a poor coverage and, therefore, a higher value of posterior error. When the sensing radius of the sensors is larger, the path is planned such that the mobile robot maintains a distance from the sensor which is at most equal to the sensing radius of the sensor. Thus the mobile robot and sensors maintain maximum distance from each other. As a result, it covers more area of the water column than is possible when the radius is smaller.

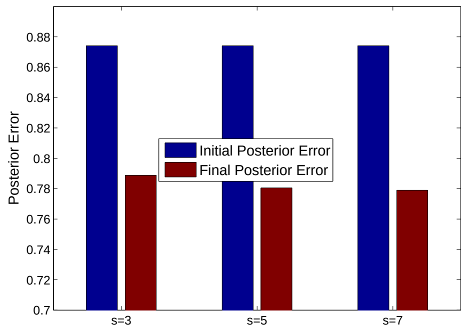
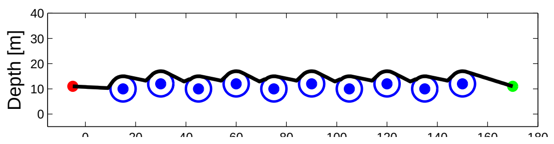


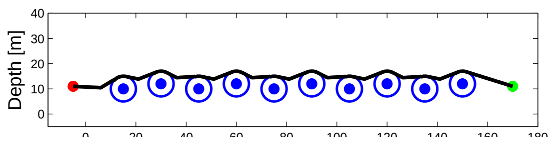
Figure 5.20: Posterior Error Comparison for Figure 5.19

5.8.4 Path Planning when Sensors lie close to each other

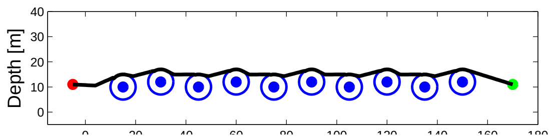
Unlike Section 5.7, in which the Algorithm 2 is unable to plan a path of the mobile robot if the sensors are arranged in a co-linear fashion, this algorithm is able to find a robot path. However, in this case the mobile robot path will be formed by tracing the sensing boundaries of the sensors, so much of their sensing zone will be overlapped and most of the rest of the water column will remain unexamined. A similar scenario occurs when the sensors lie very close to one another as shown in Figure 5.21. In this figure we show the



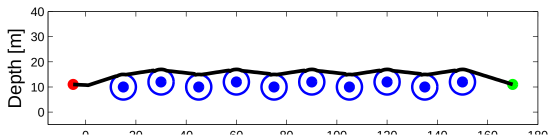
(a) Robot's Sensing Radius = 2



(b) Robot's Sensing Radius = 5



(c) Robot's Sensing Radius = 7



(d) Robot's Sensing Radius = 10

Figure 5.21: Different r_R of the robot with $interval = 1$ and sensor's sensing radius, $r_S = 5$

path of the mobile robot when the sensors are arranged very close to one another in a zigzagged fashion at 10 meters and 12 meters respectively. We test this scenario with the sensors having a sensing radius of $r_S = 5$ and $interval = 1$. The robot's sensing radius is varied from 2 to 10 meters.

Table 5.5: Posterior Error Comparison for TanBugPath algorithm for Different layouts described in Section 5.8.4

Layout #	Only Sensors	Sensors + Robot Path
1	0.8733	0.7975
2	0.8733	0.7956
3	0.8733	0.7947
4	0.8733	0.7948

We observe that, for all cases, the path planned for the mobile robot passes over sensing bubble of the sensors. If the r_R is larger, the path is straighter. The comparison between the different layouts with respect to the posterior error calculated in each scenario can be seen in Table 5.5 and is represented in Figure 5.22. As before, we assume that the mobile robot can only

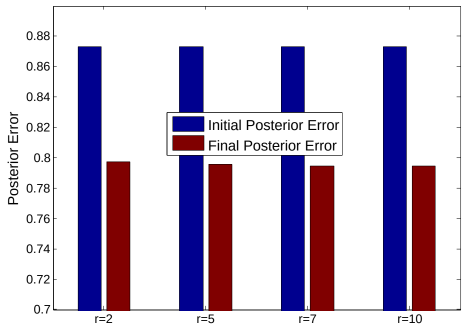


Figure 5.22: Posterior Error Comparison for Figure 5.21

sense nine times through the entire water column. So the mobile robot senses at every $floor(\frac{N_{path}}{9})^{th}$ location from a set of N_{path} points that are there on its path. As discussed before, these $floor(\frac{N_{path}}{9})$ positions are randomly chosen and as such, do not have as much influence on the posterior error as it could have had if they had been chosen strategically.

5.9 Experiments on Adaptive Path Algorithm

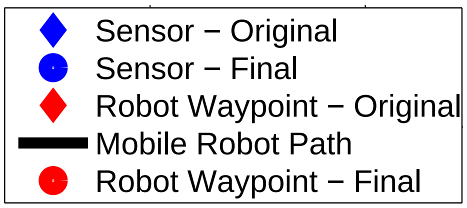
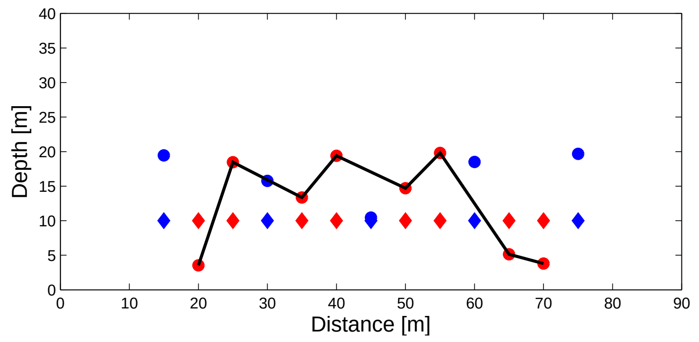


Figure 5.23: Legend

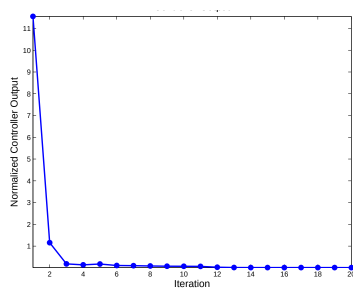
In this section, we study the performance of Algorithm 4. At first, we show the performance of the algorithm on varying node layouts and show that the controller output, described in Section 4.14 goes to zero. After that, we vary different parameters that are set by the user at the beginning of the algorithm and study the effect of each on the final node layout and overall sensing. In all the following sections, unless otherwise stated, the marking conventions stated in Figure 5.23 will be followed for all figures showing the initial and final node layouts.

The important parameters that govern when the algorithm believes the system has converged are $threshold_{lim}$ and $turns_{lim}$. These two factors are set by the user at the beginning of the algorithm. We found that if we do not set a $threshold_{lim}$, the algorithm will stop after the first 5 iterations as the objective function always goes down during this time. Thus the algorithm has not actually converged. Hence we need to set a $threshold_{lim}$ so that the algorithm at least runs for that many iterations before it comes to a full stop. We cannot set a very high value of $threshold_{lim}$, as the algorithm will then run longer than needed. We discuss this in detail when we talk about Figure 5.28 later. At the same time we cannot set a low value of this parameter. A $threshold_{lim} \leq 10$ is often not sufficient for some configurations as the system converges very fast (example Figure 5.24). For all the experiments, the value of the parameters $turns_{lim}$ was set to 5.

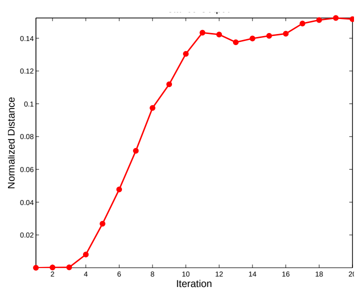
In the layout shown in Figure 5.24(a), we have 5 sensors and 2 robot waypoints between



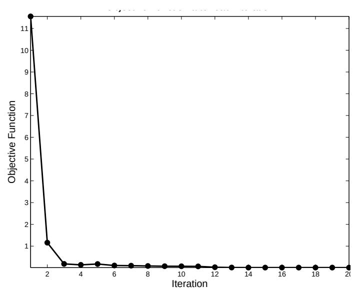
(a) Node Layout



(b) Controller Objective Function

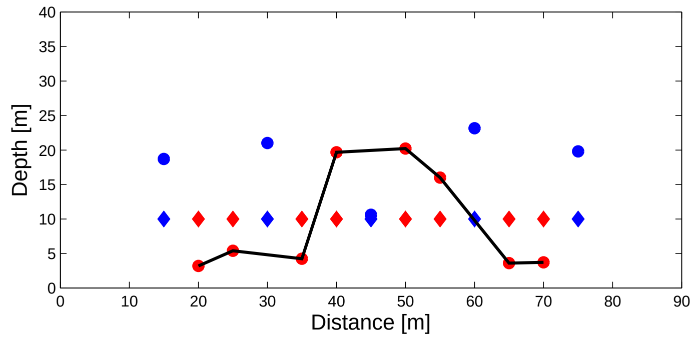


(c) Distance Objective Function

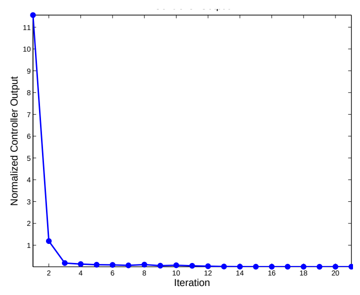


(d) Overall Objective Function

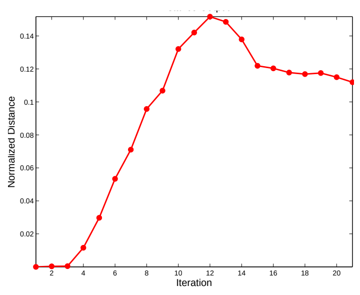
Figure 5.24: Layout 1: Initial and Final layout with Algorithm 4



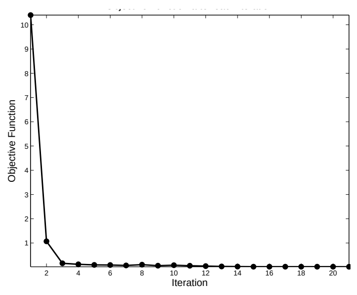
(a) Node Layout



(b) Controller Objective Function



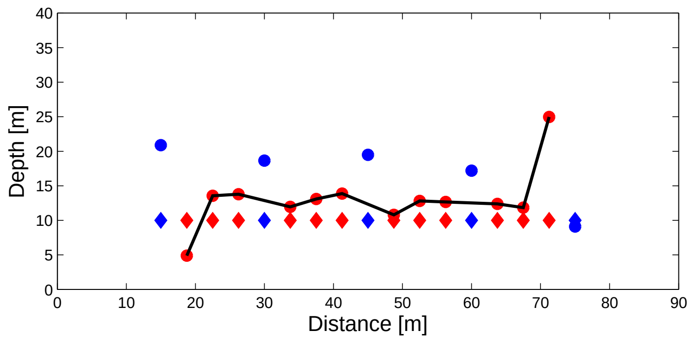
(c) Distance Objective Function



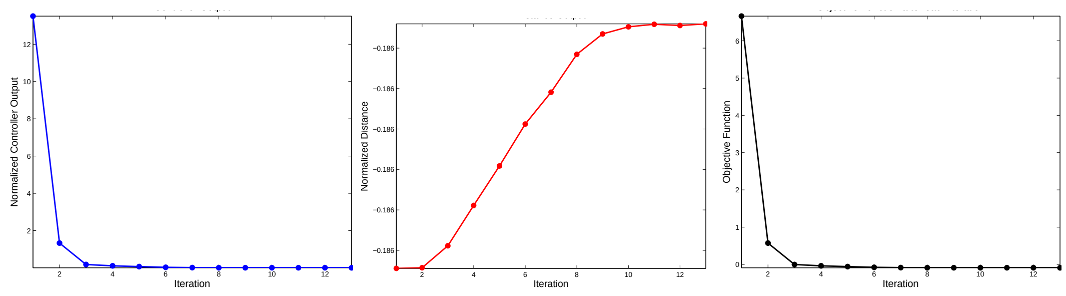
(d) Overall Objective Function

Figure 5.25: Layout 2: Initial and Final layout with Algorithm 4

each of them. The sensor and robot waypoints are all at a depth of 10m. The maximum number of iterations is set to 300 and we are using $k = 2000$ and $\alpha = 0$ for this layout. Since $\alpha = 0$, the distance function described in Section 4.4.3 has no effect on the final layout of the sensor nodes. The algorithm takes 20 iterations to converge. The change in the value of the controller objective function, the distance function and the overall controller objective function can be seen in Figs 5.24(b),5.24(c), and 5.24(d) respectively. An important observation from this configuration, which can be observed throughout all configurations, is that the peripheral sensors and robot waypoints, on the same side, tend to move in opposite direction so that the region is effectively covered. We also observe that as the algorithm converges, overall controller objective function gradually decreases and tends to go to 0 and the distance function stabilizes. The final layout of the algorithm is not an ideal zigzagged layout as between the distances 30m to 60m there is only one sensor in the lower half (at 45m) at a depth of 10.44m. Thus from this layout we learned that the value of $threshold_{lim}$ should be greater than 10.



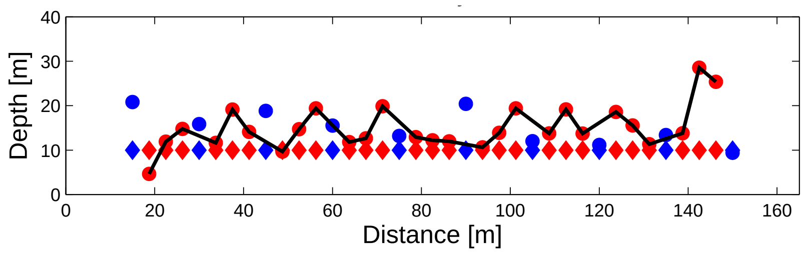
(a) Node Layout



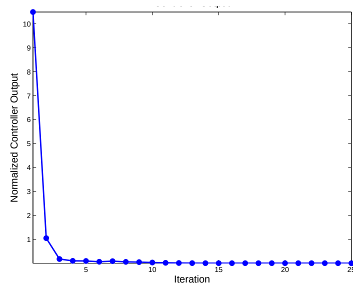
(b) Controller Objective Function (c) Distance Objective Function (d) Overall Objective Function

Figure 5.26: Layout 3: Initial and Final layout with Algorithm 4

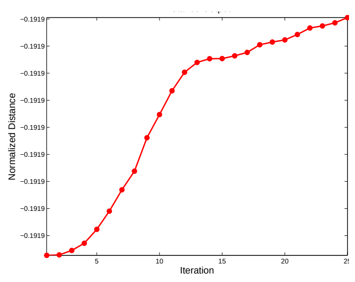
In Figure 5.25(a), there are 5 sensors and 2 intermediate robot waypoints at a depth of 10m. Just like before, the maximum number of iterations is set to 300, $k = 2000$ but $\alpha = 0.10$ for this layout. Since $\alpha = 0.10$, the overall objective function is composed of 90% value of Equation 4.1 and the rest is the distance function described in Section 4.4.3. The algorithm requires 21 iterations to converge. The change in the value of the controller objective function, the distance function and the overall controller objective function can be seen in Figs 5.25(b),5.25(c),and 5.25(d) respectively. An important observation from the final layout is that just by changing the α by a fraction we can observe that the robot waypoints tend to stick together as much as possible. Thus, when the first robot waypoint goes towards the bottom, the next robot waypoint instead of moving in the opposite directions places itself closer to the first waypoint and so on.



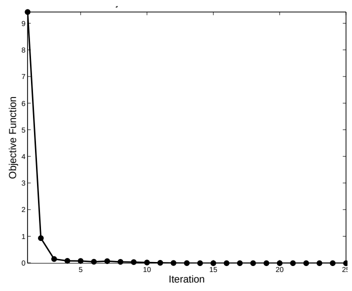
(a) Node Layout



(b) Controller Objective Function



(c) Distance Objective Function



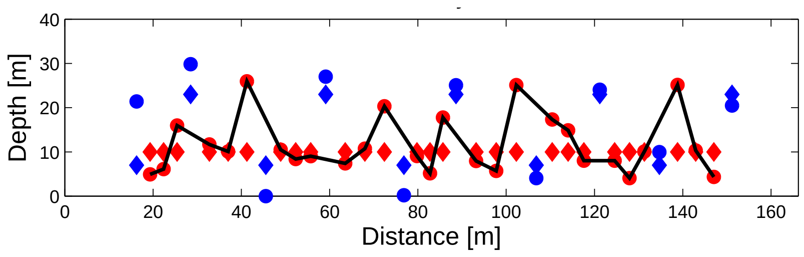
(d) Overall Objective Function

Figure 5.27: Layout 4: Initial and Final layout with Algorithm 4

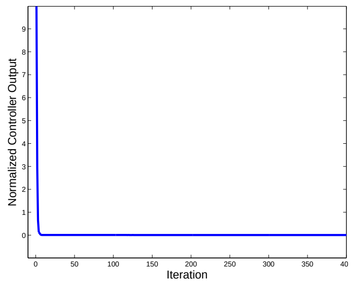
Once again in Figure 5.26(a), we study a network which has 5 sensors. But intermediate robot waypoints is increased to 3 and placed at a depth of 10m. The maximum number of iterations is set to 300, $k = 2000$ and $\alpha = 0.50$ for this layout. Since $\alpha = 0.50$, the overall objective function is composed of 50% of Equation 4.1 and 50% Equarion 4.2. The algorithm requires only 13 iterations to converge. The change in the value of the controller objective function, the distance function and the overall controller objective function can be seen in Figs 5.26(b),5.26(c), and 5.26(d) respectively. Increasing the value of α has the effect of assigning more weight towards minimizing the distance. This is very prominent in this configuration. However, the peripheral nodes still move away to provide better coverage.

Now that we have studied the basic layout of the sensor network with 5 sensors, we study bigger sized networks and verify if these also act similar to the networks with 5 sensors. In these experiments, the $threshold_{lim} = 20$ and $turns_{lim} = 5$. In the layout in Figure 5.27(a) there are 10 sensors and 3 intermediate robot waypoints, all initially at a depth of 10m. The maximum number of iterations is set to 300, $k = 2000$ and $\alpha = 0.10$. Unlike the previous experiments, the value of *hops* is set to 3 instead of 1. Therefore, we assume that the robot is able to communicate with the 3 nearest sensors on either side. This implies that at the time of optimizing the current robot waypoint, the location of all robot waypoints that are there between this one up to the third sensor with which it can communicate will be considered. The algorithm requires 25 iterations to converge. The change in the value of the controller objective function, the distance function and the overall controller objective function can be seen in Figs 5.27(b),5.27(c), and 5.27(d) respectively. We observed that even though the distance objective function seems to increase in magnitude, the displacement is very miniscule. The controller objective function is gradually decreasing and since that has a major effect on the overall controller objective function, the controller objective function decreases and tends to 0. In the final layout, we observe that all the sensors and robot waypoints except the peripheral ones tend to get arranged around the middle of the water column.

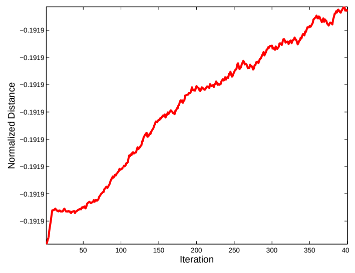
In the next layout shown in Figure 5.28(a), we have 10 sensors in zigzagged positions at 7m and 23m respectively. There are 3 intermediate robot waypoints at a depth of 10m between each pair of sensors. The number of iterations is set to 400, $k = 2000$ and $\alpha = 0.10$. No *thresholdlim* and $turns_{lim}$ were set so that the algorithm would run for all 400 iterations. The change in the value of the controller objective function, the distance function and



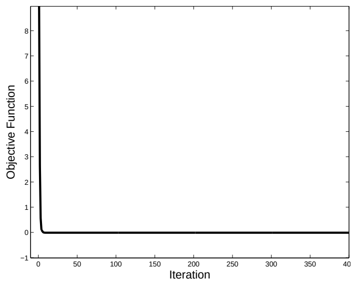
(a) Node Layout (i=400)



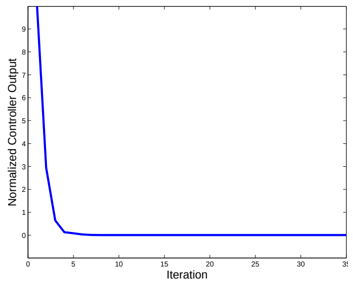
(b) Controller Objective Function (i=400)



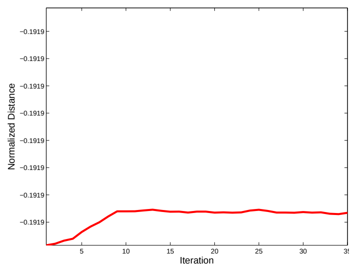
(c) Distance Objective Function (i=400)



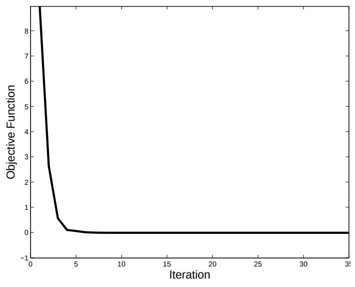
(d) Overall Objective Function (i=400)



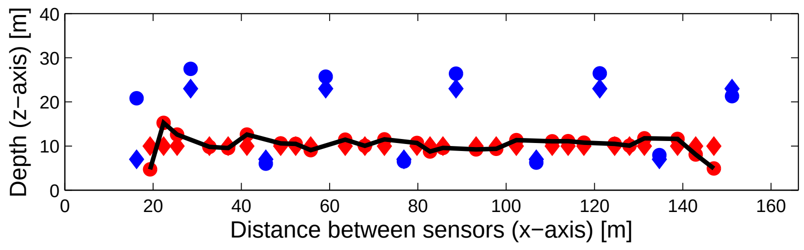
(e) Controller Objective Function (i=35)



(f) Distance Objective Function (i=35)



(g) Overall Objective Function (i=35)

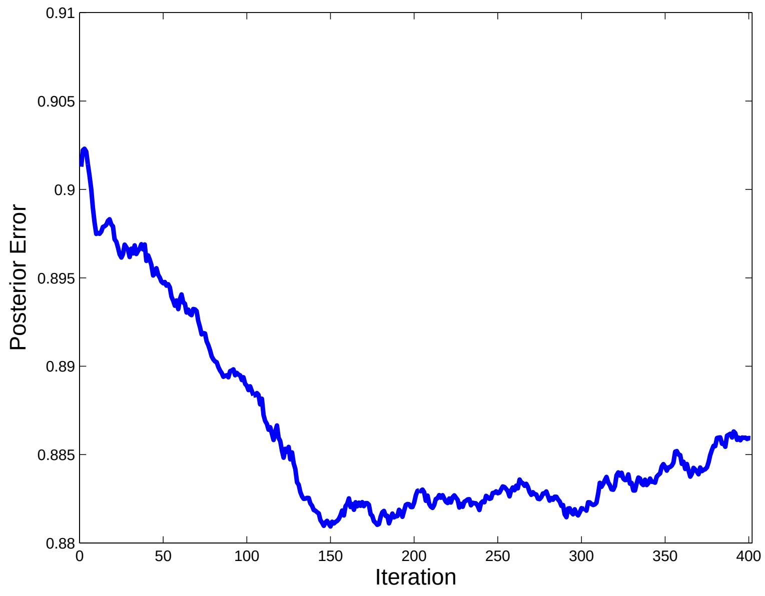


(h) Node Layout (i=35)

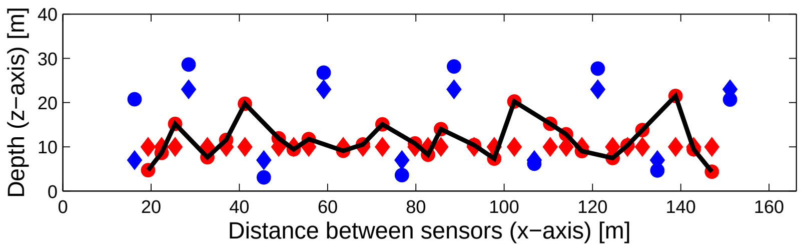
Figure 5.28: Layout 5: Initial and Final layout with Algorithm 4

the overall controller objective function can be seen in Figs 5.30(a),5.28(c), and 5.28(d) respectively. A zoomed section with only 35 iterations is shown in Figs 5.28(e),5.28(f), and 5.28(g) respectively. We found that the distance function increases for the first 9 iterations and then remains comparatively steady till 35 iterations. But after that, the value gradually increases. The more scattered network means better coverage and hence the controller objective function gradually decreases. Since the change in magnitude of the distance function is very small, the overall controller objective function is not greatly affected and decreases over time leading to 0. The layout after 35 iterations can be seen in Figure 5.28(h).

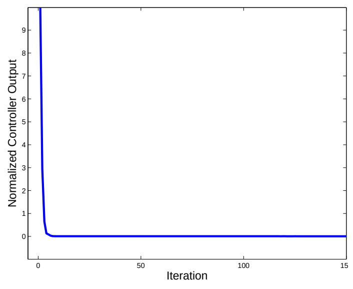
We can see that the robot path is very different in the two diagrams. So, we calculated the posterior error for each iteration in this configuration (Figure 5.29(a)). We can observe that the least posterior error is obtained when the iterations are around **150**. We plotted the mobile robot path at 150 iterations in Figure 5.29(b). The total path length for this configuration is **388.45m** and for the the layout with 400 iterations is **433.12m**. The difference in path length is only 44.67m and the difference is posterior error value is -0.005. The change in the value of the controller objective function, the distance function and the overall controller objective function for 150 iterations can be seen in Figs 5.29(c),5.29(d),5.29(e) respectively. We observe that even though the distance objective function is increasing, the value is extremely small and negative. At the same time, as distance is increasing, the robot waypoints are getting gradually scattered which subsequently decreases the controller objective function. Since the $\alpha = 0.10$, this has major effect on the overall objective function, and which decreases as well.

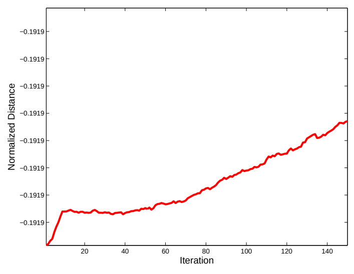


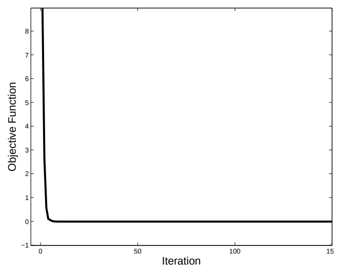
(a) Posterior Error Plot



(b) Node Layout (i=150)







(c) Controller Objective Function (i=150)

(d) Distance Objective Function (i=150)

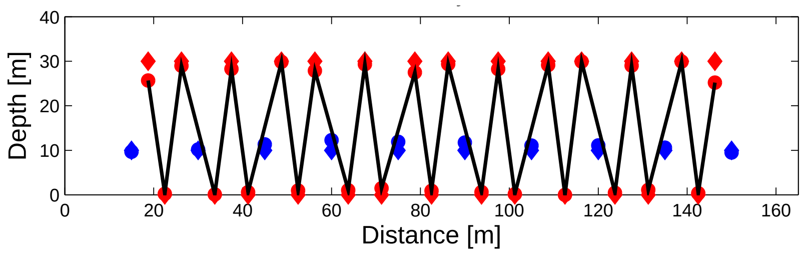
(e) Distance Objective Function (i=150)

Figure 5.29: Additional Information on Figure 5.28

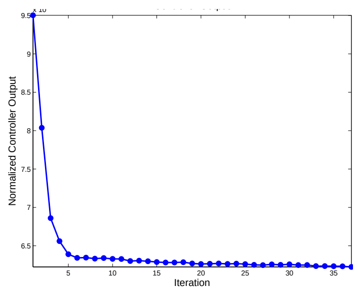
Finally, in the last layout we discuss an input dependency of Algorithm 4. In the layout shown in Figure 5.30(a) we have 10 sensors and 3 robot waypoints between each of them.

The sensors are all at 10m but the robot waypoints are initially placed in a zigzagged layout at 0m and 30m respectively. The maximum adaptive number of iterations is set to 300 and we are using $k = 4000$ and $\alpha = 0.10$. Just like Figure 5.28 the distance function described in Section 4.4.3 has only 10% effect on the final layout of the sensor nodes. The algorithm requires 37 iterations to converge. The change in the value of the controller objective function, the distance function and the overall controller objective function can be seen in Figs 5.30(b),5.30(c), and 5.30(d) respectively. We can observe that the distance function gradually decreases, but this is because the robot waypoints are already placed at the maximum distance positions at the edges of the water column and the algorithm works so that the nodes cannot be placed outside the water column. Hence, any displacement in the original positions will cause a decrease in distance. The controller objective function and the overall objective function also decreases and the algorithm converges. But, in the final layout, we find that the robot waypoints are still at the extremities and the sensors lie around the middle of the water column. This is not a low cost path because, ideally, the positions of the sensors and the robot waypoints would be interchanged in a low cost path. But in the algorithm the sensors and robot waypoints are treated as individual sensing points, and since the $\alpha = 0.10$, the effect of minimizing distance is not prominent.

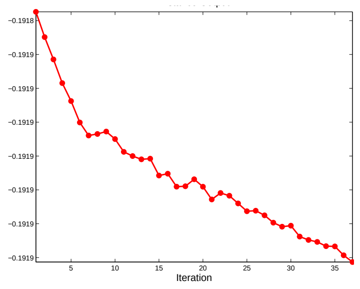
When $\alpha = 0.10$, the maximum path length is **742.32m**, and it takes the system only 37 iterations to converge. The posterior error values for the initial node layout, final node layout, and final node layout without robot are 0.7169, 0.6726 and 0.8733 respectively. When $\alpha = 0.50$, then the maximum path length is **725.26m**, and it takes the system more than 100 iterations to converge. The posterior error values for the initial node layout, final node layout and final node layout without robot are 0.7169, 0.6613 and 0.8733 respectively. The maximum path length is smaller for $\alpha = 0.50$ than for 0.10 as expected. However, the effect of minimizing the distance is still not prominent because the sensor nodes which were at 10m of the water column still stays closer to their starting locations, and the same applies for the robot waypoints. Also the system is more stable when α values is lower. For example, it takes lesser number of iterations to converge when $\alpha = 0.10$ than when it is 0.50. Therefore, the algorithm is dependent on the starting configurations of the sensors and robot waypoints. This is, in fact, a local minimum of the objective function and the algorithms stops there. The dependency of the final layout of nodes on the input layout of the sensors and robot waypoints is examined in detail in Sections 5.9.7 and 5.9.7, respectively.



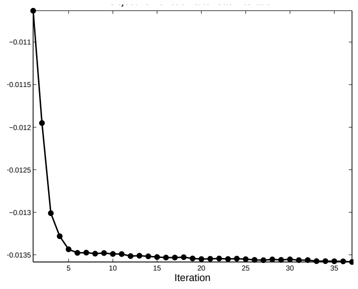
(a) Node Layout



(b) Controller Objective Function



(c) Distance Objective Function



(d) Overall Objective Function

Figure 5.30: Layout 6: Initial and Final layout with Algorithm 4

In the following sections we study the performance of Algorithm 4 on varying different parameters. Firstly, in Section 5.9.1, we study the effect of k on the final result by keeping the other factors constant. In Section 5.9.2 we study the algorithm by varying α. In Section 5.9.3 the effect of varying the number of intermediate robot waypoints V between two sensors nodes is evaluated. The effect of increasing network size on the running time of the algorithm is studied in Section 5.9.4. In Section 5.9.5 we verify if the algorithm is able to plan the the mobile robot path efficiently when one or more sensors stop working. Section 5.9.6 lists the behavior of the algorithm when the water column has regions with varying properties (different values of covariance). In Section 5.9.7 and Section 5.9.8, we vary the starting configuration of the sensors and the robot waypoints respectively and see how that affects the resulting final layout. Finally in Section 5.9.9 we study the effect of varying the *hops* parameter on the final path length.

5.9.1 Experiments on k

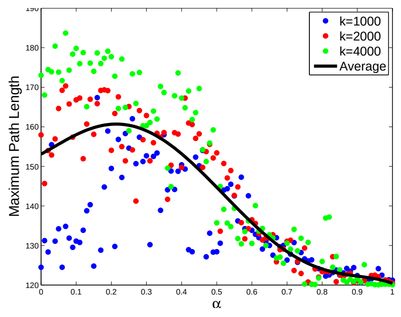
The parameter k is the constant in Equation 4.14 that controls the rate at which the depth of the sensors changes in the water column. We performed experiments by taking different values of $k = [1000, 2000, 4000]$ and by keeping the rest of the parameters constant. We experimented with three different networks:

- A network having 10 sensors which were arranged in a zigzagged manner at depths 7m and 23m respectively. $V = 1$, i.e., only one robot waypoint between a pair of sensors, at depth of 10m (Figure 5.31(a)).
- A network having 10 sensors all at depth of 10m at starting. $V = 1$, i.e., only one robot waypoint between a pair of sensors, at depth of 10m (Figure 5.31(b)).
- A network having 10 sensors all at depth of 10m at starting. $V = 3$, i.e., three robot waypoints between a pair of sensors, at depth of 10m (Figure 5.31(c)).

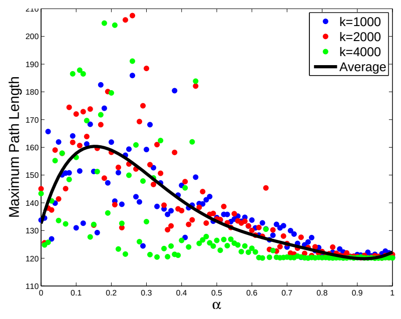
The $threshold_{lim}$ and $turns_{lim}$ were set to 20 and 5 respectively with the rest of the parameters set as $hop = 1$, $iterations_{max} = 300$, and $[\sigma_s, \sigma_d] = [5, 4]$ respectively. The results are shown in Figure 5.31 where the blue dots represents the value of the maximum path length when $k = 1000$, red dots for $k = 2000$ and green dots for $k = 4000$. We averaged the value of maximum path length for the three different k values and then fitted a curve to it. We can observe that the maximum path length value for small k is smaller than others.

We noticed that increasing the value of k causes the nodes to move down the gradient of H more quickly. If k value is small, these changes are slow and more gradual. If k is too large, that can lead to oscillations around the final configuration or lead to instabilities in the system. However, if the value of k is very small, then the system may not move fast enough to converge within a reasonable number of steps.

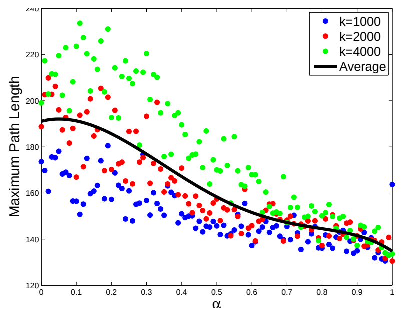
Since the k controls how fast the depth of the sensors can change, this implies that with smaller values the system changes the depths at a slower rate. We observed that when $V = 1$, the peripheral sensors and robot waypoints oscillate and hence, the system often does not converge within the specified $iterations_{max}$. This is more prominent when the k values are bigger, because then the depth changes in bigger steps in each iteration. We found that value of k around 2000 gives the best result for all the layouts.



(a) zigzag with $V = 1$



(b) straight with $V = 1$



(c) straight with $V = 3$

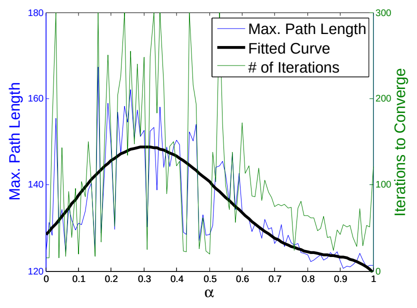
Figure 5.31: α vs. Maximum Path Length and Iterations required to Converge for a 10 sensor network starting in a zigzagged layout with sensors at 7m and 23m respectively. $V = 1$ at 10 m

5.9.2 Experiments on Changing α

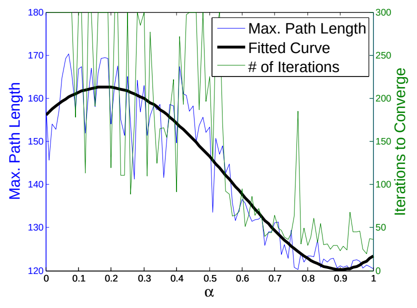
The weight, α, determines whether the sensing objective function or the distance function plays major role in determining the length of the mobile sensor path, which determines the position of the robot waypoints in the water column. We perform experiments to determine how to control the length of the mobile robot path by varying α. The α is used in the algorithm to determine which of the objective functions has a more prominent role in determining where the robot way points will finally end up in the water column.

The sensing objective function tries to distribute the way points evenly through the water column in a zigzagged fashion so that sensing is maximized. However, the distance objective function tries to align the robot waypoints in a straight line so that the distance is minimized. The α controls the impact of the objective functions on the robot path length. So if the $\alpha = 0$, the sensing objective function plays the major role and the distance objective function has no effect on the robot path length. On the other hand, if $\alpha = 1$, the sensing objective function does not play any role and the distance function should try to align the robot way points in a minimum distance path, i.e., ideally, a straight line.

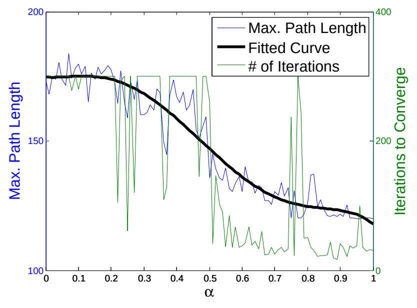
Figure 5.32 shows the data for a network having 10 sensors which were arranged in a zigzagged manner at depths of 7m and 23m alternately. There is only one robot waypoint between a pair of sensors at 10m. The $threshold_{lim}$ and $turns_{lim}$ were set to 20 and 5 respectively. The rest of the parameters were set as $hop = 1$, $iterations_{max} = 300$ and $[\sigma_s, \sigma_d] = [5, 4]$. $hop = 1$ signifies that the mobile robot can communicate with only the nearest node on each side.



(a) k=1000



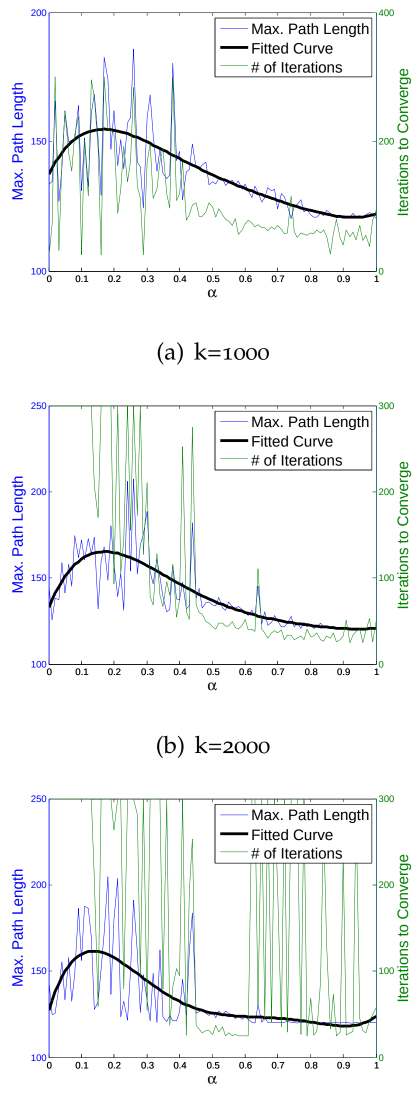
(b) k=2000



(c) k=4000

Figure 5.32: α vs. Distance, Iterations for 10 sensors zigzagged at 7m and 23m respectively. $V = 1$ at 10m

$iterations_{max} = 300$ signifies that the algorithm is run for a maximum of 300 iterations, i.e., if the algorithm does not converge within 300 iterations the algorithm is stopped. We increase the value of α from 0 to 1 in the intervals of 0.01. We get Fig 5.32(a) for $k = 1000$, Fig 5.32(b) for $k = 2000$, and Fig 5.32(c) for $k = 4000$. The magnitude of the robot path length vs. α is plotted along with the number of iterations it has taken each of the layouts. A curve is fitted to the maximum path length. We observed that for $k = 1000$, the system converges for each layout. However for $k = 2000$ and $k = 4000$, the system does not always converge and run till the $iterations_{max}$, especially for small values of α. We can observe, between $\alpha = 0.0$ to $\alpha = 0.7$, the length of the path varies widely. However, after $\alpha = 0.7$, the path length either remains constant or gradually decreases. The curve fitting for maximum path length has a similar pattern for all values of k. Therefore, for a zigzagged initial layout of sensors, we can call $\alpha = 0.7$ as the transition α such that the distance covered by mobile robot for $\alpha < 0.7$ is greater than that covered by it when $\alpha \geq 0.7$.



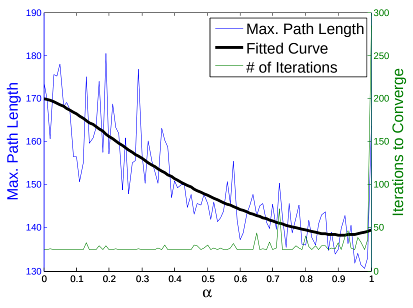
(a) k=1000

(b) k=2000

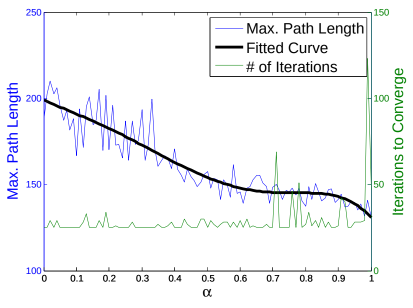
(c) k=4000

Figure 5.33: α vs. Distance, Iterations for 10 sensors starting in a straight layout 10m. $V = 1$ at 10m

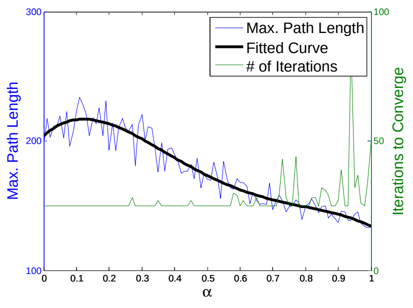
Figure 5.33 shows the data for a network having 10 sensors which were arranged in a straight colinear layout at a depth of 10m in the water column. There is only one robot waypoint between a pair of sensors at the same depth. Just like in the case before, we set $threshold_{lim} = 20$, $turns_{lim} = 5$, $hop = 1$, $iterations_{max} = 300$ and $[\sigma_s, \sigma_d] = [5, 4]$ respectively. Once again, we increase the value of α from 0 to 1 in the intervals of 0.01. We get Fig 5.33(a) for $k = 1000$, Fig 5.33(b) for $k = 2000$, and Fig 5.33(c) for $k = 4000$. In this layout, we observed that for $k = 1000$, the system converges for each layout. However, for $k = 2000$, the system does not converge for small values of α and for $k = 4000$, the system does not converge for small as well as large values of α. In these cases, the system runs till the $iterations_{max}$. However, the fitted curve for all values pf k are comparable and have similar maximum path length irrespective of if the system converged or not. For this layout, we observed, between $\alpha = 0.0$ to $\alpha = 0.5$, the length of the path varies widely. However, after $\alpha = 0.5$ the path length variation decreases. The curve fitting for maximum path length has a similar pattern for all values of k. Therefore, for a straight layout of sensors with one intermediate robot waypoint, $\alpha = 0.5$ is the transition α such that the distance covered by mobile robot



(a) k=1000



(b) k=2000



(c) k=4000

Figure 5.34: α vs. Distance, Iterations for 10 sensors starting in a straight layout at 10m. $V = 3$ at 10m

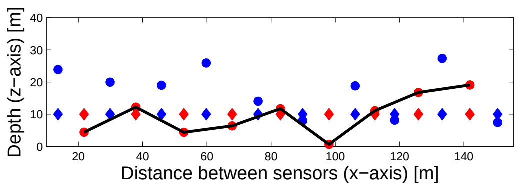
for $\alpha < 0.5$ is greater than that covered by it when $\alpha \geq 0.5$.

Figure 5.34 shows the data for a network having 10 sensors which were arranged in a straight colinear layout at a depth of 10m in the water column. There are three robot waypoints between a pair of sensors at the same depth. We set $threshold_{lim} = 20$, $turns_{lim} = 5$, $hop = 1$, $iterations_{max} = 300$ and $[\sigma_s, \sigma_d] = [5, 4]$ respectively. Once again, we increase the value of α from 0 to 1 in the intervals of 0.01. We get Fig 5.34(a) for $k = 1000$, Fig 5.34(b) for $k = 2000$, and Fig 5.34(c) for $k = 4000$. In this layout, the system converges for all values of α. The maximum path length for $k = 2000$ and 4000 are comparable, but for $k = 1000$ the path length is lesser for smaller values of α. For all the k for this layout, between $\alpha = 0.0$ to $\alpha = 0.6$, the length of the path varies widely. However, after $\alpha = 0.6$ the path length variation decreases. Therefore, for a straight layout of sensors with three intermediate robot waypoints, $\alpha = 0.6$ is the transition α such that the distance covered by mobile robot for $\alpha < 0.6$ is greater than that covered by it when $\alpha \geq 0.6$.

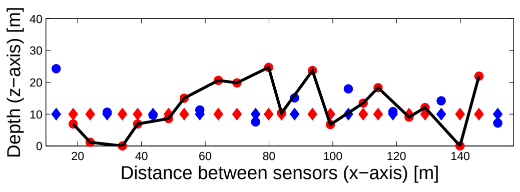
5.9.3 Experiments on Number of Intermediate Nodes, V

An important parameter that decides the final mobile robot path length is the number of robot waypoints, V, that have been specified between two sensor nodes. More number of robot waypoints gives the algorithm a finer control on the length and layout of the mobile robot path. Each robot waypoint represents a new position where the mobile robot can travel to collect data. However, intuitively, more number of robot way points also mean that the system will need more number of iterations to converge. In this section we perform experiments by varying the number of robot waypoints for two different starting layouts:

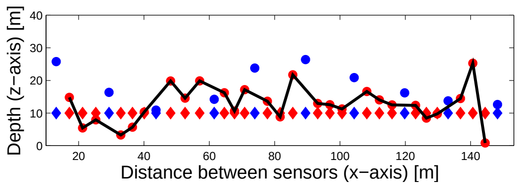
- All the sensors and robot waypoints are arranged colinearly at 10 meters;
- Robot waypoints are arranged colinearly at 10 meters and sensors are arranged in a alternate zigzagged layout at 30m and 0m respectively.



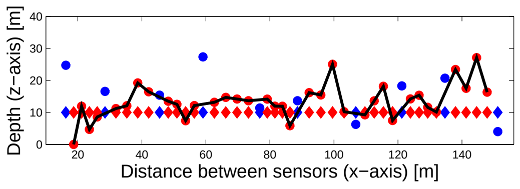
(a) Initial & Final Node Layout for $V = 1$



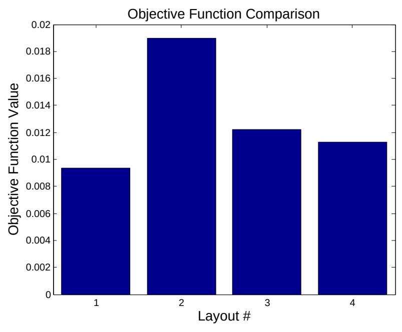
(b) Initial & Final Node Layout for $V = 2$



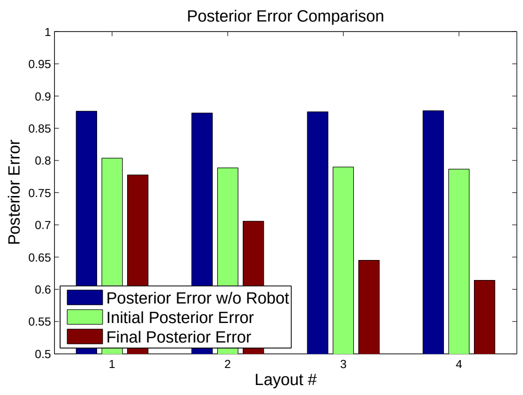
(c) Initial & Final Node Layout for $V = 3$



(d) Initial & Final Node Layout for $V = 4$



(e) Final Objective Function Value

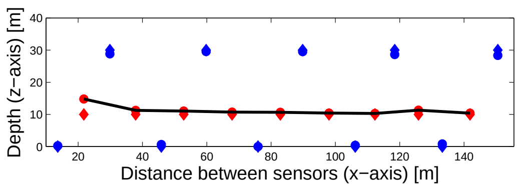


(f) Final Posterior Error Value

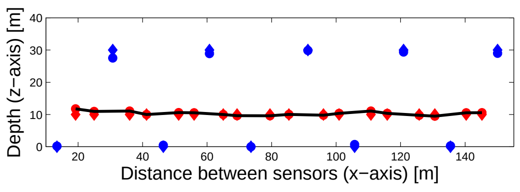
Figure 5.35: Final Mobile Robot Path for a straight starting layout for $R = 10$, $k = 4000$, $hop = 1$, $iters_{max} = 200$, $\alpha = 0.10$, $\sigma_s = 10$, $\sigma_d = 4$)

The modified *Adaptive Decentralized Control* algorithm is run on each of these layouts for four different values of V.

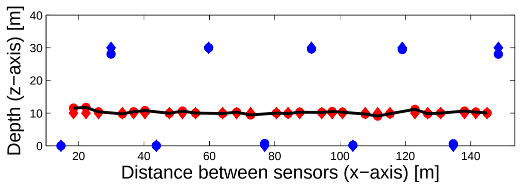
The initial and final layouts for straight and zigzagged layout are shown in Figure 5.35 and Figure 5.36 respectively. The corresponding value of the final objective function at the beginning and end of the experiment are shown in Figure 5.35(e) and 5.36(e); and that of the posterior error value in Figure 5.35(f) and 5.36(f). Table 5.6 shows the comparison of the total distance traveled by mobile robot, total number of iterations required to converge, and initial and final objective function values.



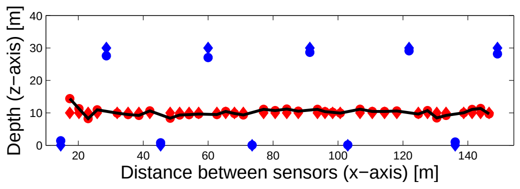
(a) Initial & Final Node Layout for $V = 1$



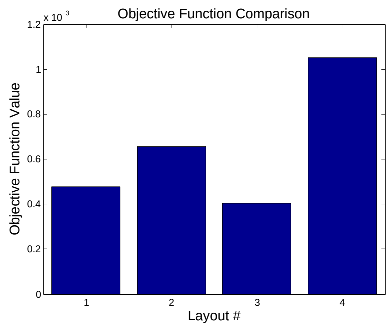
(b) Initial & Final Node Layout for $V = 2$



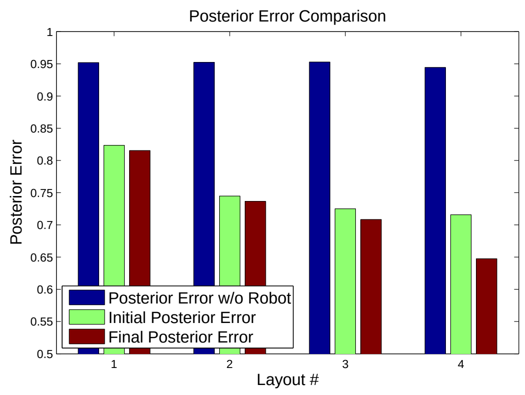
(c) Initial & Final Node Layout for $V = 3$



(d) Initial & Final Node Layout for $V = 4$



(e) Final Objective Function Value



(f) Final Posterior Error Value

Figure 5.36: Final Mobile Robot Path for a zigzagged starting layout for $R = 10$, $k = 4000$, $hop = 1$, $iters_{max} = 200$, $\alpha = 0.10$, $\sigma_s = 10$, $\sigma_d = 4$)

We can see in Figure 5.35, when the sensors are in a straight layout at the starting, there is possibility of more variation in the mobile robot path because the sensors and robot waypoints are treated equally and, hence, can move in either direction leading to wide difference in planned paths. Also, we can observe that the total distance traversed by the mobile robot increases on increasing the number of intermediate robot waypoints. This is because increasing the robot waypoints gives granular control on where they should be placed and due to the nature of the algorithm minimum distance path is not the most effective path for gaining information. The maximum iterations was set to 200 and none of the layouts needed more than that to converge. The number of iterations required for the

Table 5.6: Comparison for layouts shown in Figure 5.35 and Figure 5.36

Layout	V	Total Distance	Initial Objective	Final Objective	Iterations
Straight	1	133.25	0.8890	0.0093	80
	2	199.90	0.8484	0.0190	89
	3	207.90	0.9320	0.0122	115
	4	234.64	0.8936	0.0113	140
Zigzag	1	120.53	0.0006	0.0005	12
	2	126.41	0.0008	0.0007	11
	3	127.75	0.0005	0.0004	10
	4	136.78	0.0006	0.0011	32

algorithm to converge increases on increasing the number of intermediate robot waypoints, which is expected. The final objective function value is only 1-2% of the initial objective function value. The posterior error value on adding the mobile robot path is lesser than when there is no robot in each of these cases. And the final posterior value is lesser than the initial posterior error value even when the mobile robot is present in the system. On increasing the number of intermediate robot waypoints, the value of the posterior error of the final layout gradually decreases.

However, from Figure 5.36, we can observe that if the starting location of the sensors is zigzagged or has been already optimized with the *Adaptive Decentralized Control* the robot waypoints generally end up not moving much from their initial location. As we can observe from the data in Table 5.6, the objective function at the beginning of the algorithm is already very low compared to the values in the previous layout. This means the starting layout is already capable of sensing the area very efficiently. The final value of objective function for this starting layout is 80-90% of the initial value and the number of iterations required to converge is much less than the other layout. This implies that the algorithm considers the starting layout as a good layout for sensing and does not change it.

5.9.4 Experiments on Number of Iterations

The size of the sensor network depends on the sensors as well as the number of robot waypoints in between two sensor nodes. On increasing the number of robot way points in between two sensors, the number of sensing locations in the water column increases. This directly impacts the size of the network, even though the area of sensing remains the same. The running time of the *Adaptive Decentralized Control* depends upon the size of the network. Intuitively, if the network size increases, the time required for the *Adaptive Decentralized Control* algorithm to converge should increase. But this may not be the case for the Algorithm 4 because the coverage of the area of water column is more important. Denser networks, i.e., networks having more number of sensors and robot sensing points automatically cover more area of water and hence, often converge faster than sparse ones.

The number of iterations that the algorithm needs to converge can be a good estimation for the running time of this algorithm. In this section we present iteration data to estimate the running time for the algorithm on different sized networks and study the factors which affect the running time. However, we should note that one iteration for a network with less number of nodes does not take exactly same time as a network with more number of nodes. This is because each of the sensors and robot way points are optimized individually. So the number of iterations is only a rough estimation of the running time.

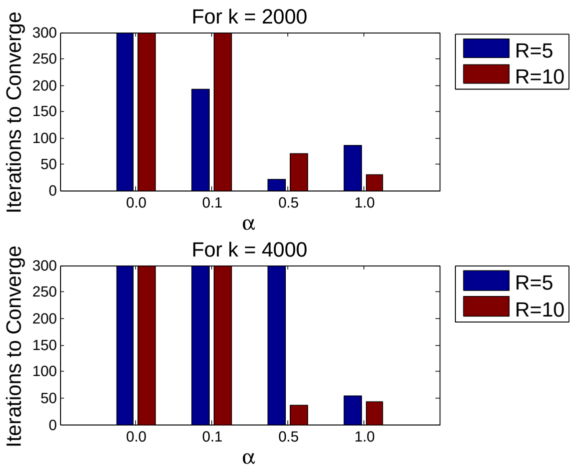


Figure 5.37: Number of iterations to converge for different α values and different network size for $V = 1$ and $threshold_{lim} = 10$

A network with ten sensors and one intermediate robot waypoint roughly takes 105 to 115 seconds per iteration, i.e., approximately 6 seconds per node per iteration. On the other hand, a network with ten sensors and three intermediate robot waypoints takes roughly 380 to 395 seconds per iteration, i.e., approximately 10 seconds per node per iteration. This is when all the nodes are arranged in a straight layout at the beginning of the algorithm. For a zigzagged starting layout, the time taken per node per iteration is approximately 8 seconds.

As stated earlier in Section 5.9, the main factors that affects the total number of iterations required to converge are $threshold_{lim}$ and $turns_{lim}$. These two factors are set by the user at the beginning of the algorithm. We ran the experiment with $threshold_{lim} = 10$ for one intermediate robot waypoint in a network of 10 sensors. The results are shown in Figure 5.37. Here, the number of intermediate robot waypoints was fixed at $V = 1$ and the number of sensors was varied in $R = [5, 10]$ and weight, $\alpha = [0.00, 0.10, 0.30, 0.50, 1.00]$.

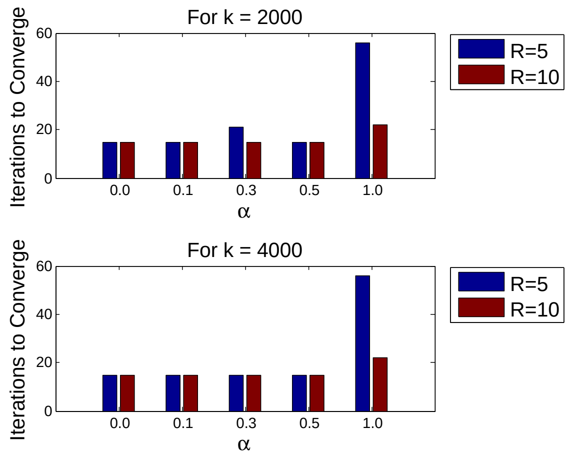


Figure 5.38: Number of iterations to converge for different α values and different network size for $V = 3$ and $threshold_{lim} = 10$

We found, firstly, that experiments with one intermediate node is not very stable as the peripheral nodes keep oscillating and often the algorithms runs till the $iterations_{max}$ is reached. Secondly, if $threshold_{lim} \leq 10$, the algorithm stops very soon for some configurations. This is because the value of the objective function is monotonically decreasing from a very high value to a low value in the first few iterations. If $threshold_{lim} \leq 10$, then this decrease can be wrongly interpreted as a gradient minimum, and the algorithm assumes that it has converged.

Considering the above observations, we again ran two sets of identical experiments, one with $threshold_{lim} = 10$ and the other with $threshold_{lim} = 20$ respectively for a network of sensors with three intermediate robot waypoints ($V = 3$). The value of the parameters $turns_{lim}$ was set to 5. For these experiments, we considered two different sizes of sensor network, $R = [5, 10]$. All the sensors were initially placed at 10 meters. Then we varied the weight factor, $\alpha = [0.00, 0.10, 0.30, 0.50, 1.00]$, and $k = [2000, 4000]$ respectively. The maximum number of iterations was set to $iterations_{max} = 300$, i.e., if the algorithm does not converge within 300 iterations then it stops. The field *Iterations to Converge* is the total number of iterations that the algorithm needs to converge to the steady state.

Table 5.7: Iterations required to converge

α	**iterations**	**Path Length**	**Posterior Error**		
			Initial	Final	Final w/o Robot
$R = 5, V = 3, k = 2000$:					
0.0	15	107.23	0.811	0.699	0.884
0.1	15	96.499	0.811	0.694	0.884
0.3	21	90.695	0.811	0.696	0.884
0.5	15	91.087	0.811	0.689	0.884
1.0	56	53.724	0.811	0.705	0.884
$R = 5, V = 3, k = 4000$:					
0.0	15	116.94	0.811	0.692	0.884
0.1	15	101.47	0.811	0.681	0.884
0.3	15	76.829	0.811	0.691	0.884
0.5	15	83.786	0.811	0.688	0.884
1.0	56	53.724	0.811	0.705	0.884
$R = 10, V = 3, k = 2000$:					
0.0	15	176.1	0.790	0.665	0.873
0.1	15	196.4	0.790	0.664	0.873
0.3	15	159.68	0.790	0.669	0.873
0.5	15	144.01	0.790	0.667	0.873
1.0	22	128.89	0.790	0.668	0.873
$R = 10, V = 3, k = 4000$:					
0.0	15	189.7	0.790	0.652	0.873
0.1	15	214.93	0.790	0.646	0.873
0.3	15	194.3	0.790	0.652	0.873
0.5	15	154.28	0.790	0.651	0.873
1.0	22	128.89	0.790	0.667	0.873

In Figure 5.39, we show the data for the experiment with the number of intermediate robot waypoints fixed at $V = 3$ and varied the number of sensors, $R = [5, 10]$ and weight, $\alpha = [0.00, 0.10, 0.50, 1.00]$ respectively for when $threshold_{lim} = 10$. The results are shown in Table 5.7. We can observe that, keeping the network size constant, the number of iterations depends both on α and k. In Figure 5.39, we show the same data for $threshold_{lim} = 20$.

We can observe that there is not much difference in the number of iterations for $k = 2000$ and $\alpha = 0.00$. Sometimes the number of iterations for $k = 2000$ is more than that for $k = 4000$. This is because when $k = 2000$, the depths of sensors and robot waypoints are changed in small increments. So $k = 2000$ takes longer time to converge. Another important observation is that the number of iteration required to converge is always more for $threshold_{lim} = 20$ than for $threshold_{lim} = 10$. One reason is that in the first case we are deliberately running the algorithm for more number of iterations. At the same time, this also shows that for $threshold_{lim} = 10$ the algorithm can converge at a local minimum instead of the global minimum.

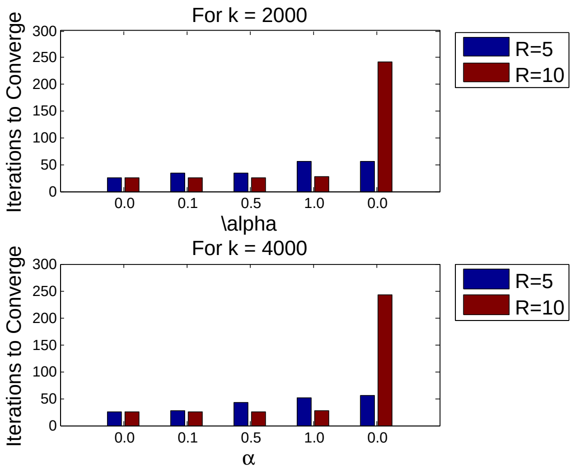


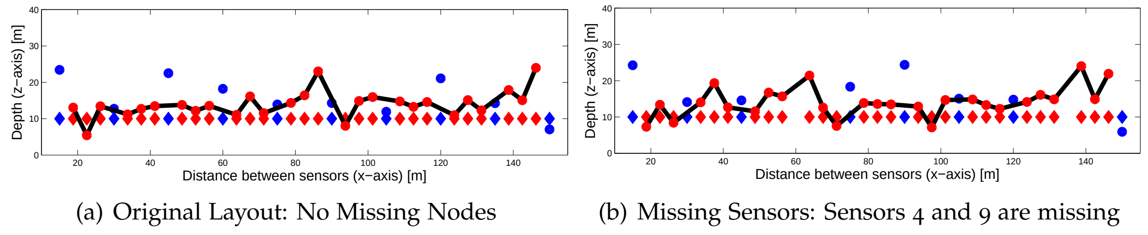
Figure 5.39: Number of iterations to converge for different α values and different network size for $V = 3$ and $threshold_{lim} = 20$

5.9.5 Experiments on Missing Sensors (Uneven node layout)

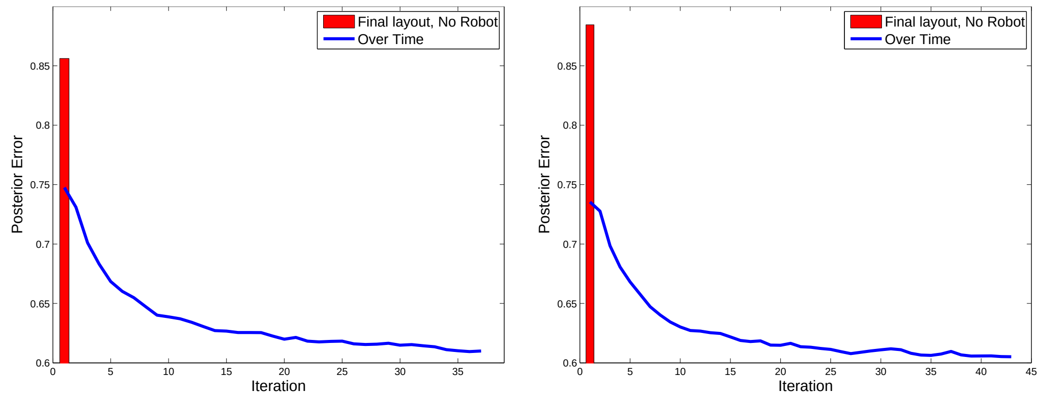
Sometimes, unforeseen circumstances may lead to one of more sensors not acting as expected. The sensors might not be deployed at the correct location, or battery might die leading to they sensors becoming inactive. In such scenario, Algorithm 4 should plan the path of the mobile robot such the entire water column is covered in the best possible manner. A method of testing that is comparing two layouts:

- all sensors are working;
- one or more sensors not working.

We simulated such a scenario with the help of ten sensor nodes. In the first layout, we arranged all the sensor nodes at a depth of 10 meters and added three intermediate robot waypoints between each of them. In the second layout, everything was kept same except two sensors were erased from the layout. These were the fourth sensor (at a distance of 60 meters from the shore) and the ninth sensor (at a distance of 135 meters from the shore). The algorithm was run on both the scenarios and the results are presented in Figure 5.40 and table 5.8.



(a) Original Layout: No Missing Nodes (b) Missing Sensors: Sensors 4 and 9 are missing



(c) Change in Posterior Error over time - original layout (d) Change in Posterior Error over time - Missing Sensors layout

Figure 5.40: Initial and Final Sensor and Mobile Robot Path layout for Uneven Node layout (Details:$k = 4000, hop = 1, Iterations_{max} = 100, \alpha = 0.10, \sigma_s = 5, \sigma_d = 4$)

Table 5.8: Comparison for layouts shown in Figure 5.40

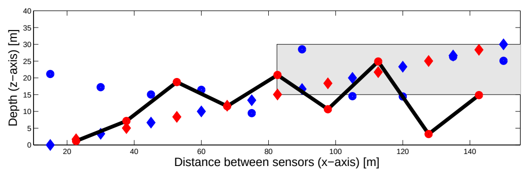
Layout	**Posterior Error**			**Iterations**	**Total Distance**
	Initial	Final	Final w/o Robot		
Original	0.748	0.610	0.856	36	175.05
Missing Sensors	0.735	0.605	0.884	42	181.11

Figure 5.40(a) shows the initial and final node layout for the original layout and Figure 5.40(b) for the one with the missing sensors. As we can see, when a sensor is missing the robot waypoint is moved in such a way that the location originally covered by a sensor is now covered by the robot waypoints that are nearest to it. This is especially

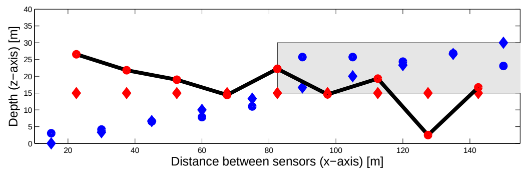
prominent for the missing sensor at 60 meters from the shore line. The posterior error changes over time for the original layout and the layout with missing sensors is represented in Figure 5.40(c) and Figure 5.40(d) respectively. The red bar represents the value of the posterior error at the final location when mobile robot path is not considered. The blue line represents the magnitude of the posterior error changing over time from the initial layout to the final layout. This data is presented in Table 5.8. We can observe, for the final layout without the mobile robot path, the decrease in posterior error is 3.17% from when the fourth and ninth sensors are missing to when all the sensors are present. However, when the mobile robot path is added in the calculation of the posterior error, we observe that the decrease is -0.83%, the negative sign implying that the sensing is better in absence of the the fourth and ninth sensors with the mobile robot path than when the sensors are present. However, in this scenario, the algorithm takes a longer time to converge and the total distance covered by the mobile robot is more than when the sensors are present. We tested this algorithm on several different layout and found similar results. Thus we can conclude that the algorithm behaves as expected.

5.9.6 Experiments on Region of Interest

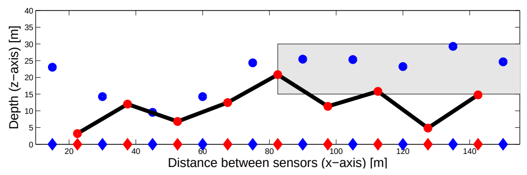
The characteristic of a column of water may not be uniform at all regions. Information content of one region may vary from the adjacent region due to various reasons. The algorithm should respond to such characteristics and align the sensor network and the plan the mobile robot path such that all regions are well covered. In this section, we simulate an area of water column with different properties by changing the covariances, σ_s and σ_d, of a particular region and keeping the covariances of the rest of the region constant. A region with lower values of σ_s and σ_d signifies that there is lesser correlation between the adjacent points in the water column. Therefore, there is more variation. Such a region is called a *region of interest* because to estimate the overall information from such region, more number of sensors needs to be deployed. Intuitively, if an area of water column has such a *region of interest* of interest, on running the Algorithm 4, the sensors and robot



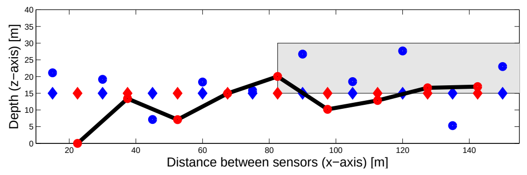
(a) Sensors and Robot Way Points in ascending pattern



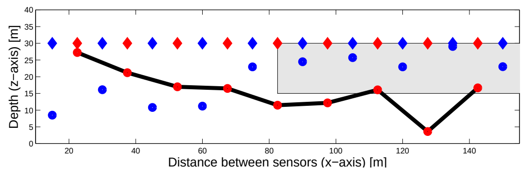
(b) Sensors in ascending pattern; Robot Way Points in a straight line at 15m



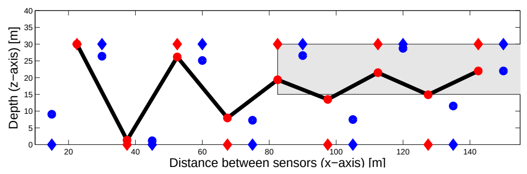
(c) Sensors and Robot Way Points at 0m



(d) Sensors and Robot Way Points at 15m



(e) Sensors and Robot Way Points at 30m



(f) Sensors and Robot Way Points starting in a zigzagged pattern

Figure 5.41: Plots showing the initial & final layout for Sensors and Robot Way Points in presence of a specific *Region of Interest* denoted by the grey region. The sensors and robot way points concentrate there to gain more information. (Details: $R = 10$, $V = 1$, $k = 2000$, $hop = 1$, $iterations = 40$, $\alpha = 0.10$).

waypoints should concentrate towards this region instead of getting distributed uniformly throughout the water column. This is an important property because, in the real world, the concentration of organic matter is often not uniform, and to gain information, the sensors should be deployed in such a way that information gain is maximized.

To implement this functionality, we need to incorporate a small change to the algorithm. While calculating $Cov(p_i, q)$ in Equation 3.11, if the q is a point in the *region of interest*, we use σ_s and σ_d that are exclusive for this region (${\sigma_s}^{ROI}$ and ${\sigma_d}^{ROI}$) instead of using the values which are same for the rest of the water column. The characteristics of the water column can change over time. This can again be reflected in the varying values of covariances over time. The algorithm will take care of any such changes by using a different value of σ_s and σ_d at different iterations, as specified by the user.

For running this simulation we designated the top right-hand-side quadrant of the water column as the *region of interest* (ROI). It was specified by halving the value of σ_d, i.e, while the rest of the water column had a covariance $[\sigma_s, \sigma_d] = [10, 4]$, this region had the covariances set to $[\sigma_s, \sigma_d] = [5, 2]$. The experiments were run for: $V = 1$, $k = 2000$, $hop = 1$, $iterations_{max} = 40$ and $\alpha = 0.10$. The sensors and the robot waypoints were started at six different layouts and then the algorithm was run on each layout for 40 iterations. Figure 5.41 shows the results of the simulation.

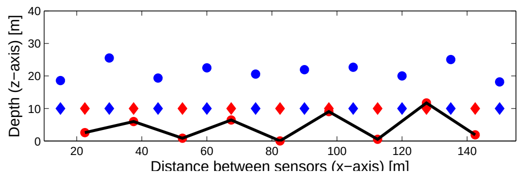
We tested the algorithm with six different starting configurations as shown in Figure 5.41. The six different starting layouts are:

- Both Sensors and Robot waypoints in ascending pattern
- Sensors in an ascending pattern and all Robot waypoints at 10 meters
- Both Sensors and Robot waypoints in at 0 meters ,i.e., bottom of water column
- Both Sensors and Robot waypoints in at 10 meters
- Both Sensors and Robot waypoints in at 30 meters ,i.e., top of water column
- Alternating Zigzagged layout for both Sensor and Robot waypoints

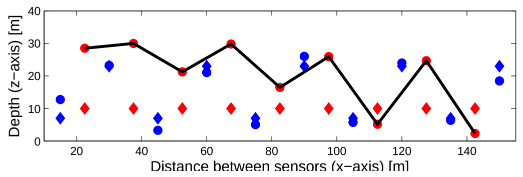
For all these layouts, after running the algorithm for 40 iterations we find that all the sensors and robot waypoints are placed either inside the ROI or very close to the ROI. We can thus observe that the algorithm performs as per expectation.

5.9.7 Experiments on Starting Configuration of Sensors

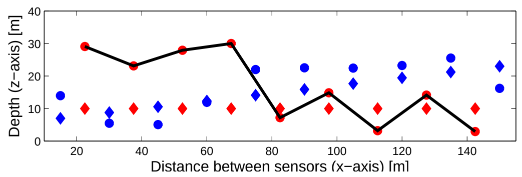
When a sensor network is optimized with Algorithm 4 with or without intermediate robot waypoints, most of the times, the sensors get aligned in an alternately zigzagged position to maximize sensing irrespective of their starting position. The same layout is obtained when the same sensors are optimized through Algorithm 4 since it is an extension of the previous algorithm. In absence of the constraint of the distance the sensors and robot way



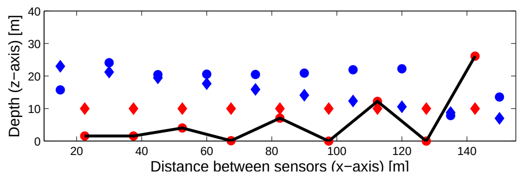
(a) Robot Waypoints initially in a straight line at depth 10m



(b) Robot Waypoints initially in zigzag patter between 7m and 23m



(c) Robot Waypoints initially in an ascending order from 7m to 23m



(d) Robot Waypoints initially in a descending order from 23m to 7m

Figure 5.42: Initial and Final Sensor and Mobile Robot Path layout for varying Sensor start locations but fixed robot waypoints start location (Details: R=10, k=4000, hop=1, $\alpha = 0.10$, $\sigma_d = 5$)

points would ideally get arranged in a similar fashion. However, when the sensors and the robot waypoints are optimized together, the starting position of the nodes may or may not affect the final layout. To get the resolution to this question, we examined four different starting layouts and studied the results. The four different layouts that are studied are:

- All sensors arranged at 10 meters in a colinear fashion.
- Sensors arranged in a zigzagged fashion at 7 meters and 23 meters alternately.
- Sensors arranged in a monotonically ascending order from 7 meters to 23 meters.
- Sensors arranged in a monotonically descending order from 23 meters to 7 meters.

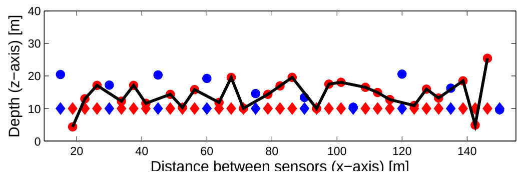
In all the cases, we placed one robot waypoint in between two sensors and arranged at 10 meters. The final configurations are shown in Figure 5.42. We observed that irrespective of the starting positions, in each of the layouts, the sensors and the robot waypoints are arranged alternately. However, the exact positions of sensors and robot waypoints depend on their positions in the starting layout.

Figure 5.42(a) has a layout in which initially both the sensors and the robot waypoints start at 10 meters. In the final layout, all the sensors move up and all the robot way points move towards the bottom. Alternately, between sensors and robot waypoints, the zigzagged arrangement of nodes is maintained.

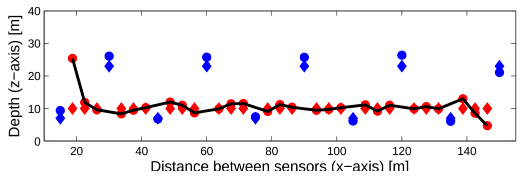
In Figure 5.42(c), the sensors at the bottom of the water column go further down, whereas the ones that are near the surface go further up. Robot waypoints, that were in between sensors which go down, goes up and vice versa. A similar pattern is observed in Figure 5.42(d) where the sensors are arranged in a descending order. In both these scenarios the peripheral robot waypoints moves in the opposite direction of the nearest sensors to provide better coverage. In some cases, we observed oscillation in the ending positions of the peripheral sensors and robot way points until the algorithm stops running.

The difference in the zigzagged pattern between sensors and robot waypoints occur in the case of Figure 5.42(b). Here, sensors are arranged in a zigzagged fashion and robot waypoints are across the center of the water column at 10m. In this layout, in the initial few iterations, the first robot waypoint places itself near the surface, because the sensor just lying before it is moving towards the bottom of the water column. Due to the distance constraint, this upward movement of the first robot waypoint, pulls all the robot waypoints upwards towards the surface. But as per initial layout, the last sensor position is towards the surface. Therefore, the peripheral robot waypoints try to stay closer towards the bottom of the water column. A zigzagged layout of sensors is already an efficient layout for sensing because the layout of the sensors does not change much in this scenario. We observe a lot of oscillations and the layout does not converge until the maximum number of iterations is reached.

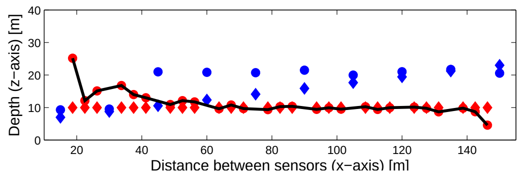
In all these layouts, we observed that the position of the first robot waypoint plays a major role in how the rest of the nodes are going to be arranged. In every iteration, the location of the robot waypoints are optimized based on their order in the water column. Hence, the first and the last robot waypoints play a major role in determining the layout of the path with more emphasis on the first robot waypoint.



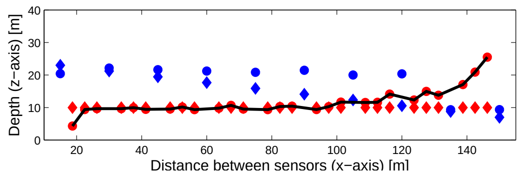
(a) Robot Waypoints initially in a straight line at depth 10m



(b) Robot Waypoints initially in zigzag patter between 7m and 23m



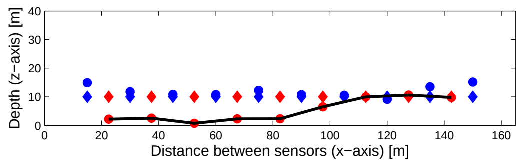
(c) Robot Waypoints initially in an ascending order from 7m to 23m



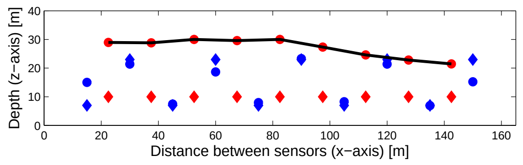
(d) Robot Waypoints initially in a descending order from 23m to 7m

Figure 5.43: Initial and Final Sensor and Mobile Robot Path layout for varying Sensor start locations but fixed robot waypoints start location (Details: R=10, k=4000, hop=1, $\alpha = 0.10$, $\sigma_d = 5$, V=3)

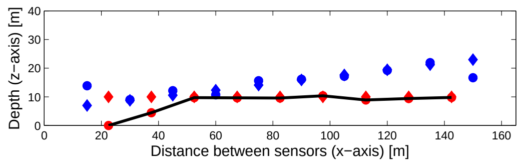
We observed that for none of the configurations, except the one in which the sensors and robot waypoints are all arranged in the same straight line, the final posterior error value is less that the initial posterior error value (Figure 5.45(a)). We found out this was due to the fact that none of the configurations converge within 100 iterations due to oscillation in the peripheral nodes. So we ran the experiments for 300 iterations. The posterior error values of the initial layout, the final layout and that of the final layout without the mobile robot sensing path is shown in Figure 5.45(b). We can observe that even for 300 iterations none of the configurations converge. This might be due to the high value of $k = 4000$ which makes the system slightly unstable. Also, as we have discussed before, the configurations with $V = 1$ are inherently unstable configurations, especially if the sensors are arranged in a zigzagged pattern. This is because the peripheral sensors often oscillate between two equally effective positions, since there other factors like distance constraints with nearest robot waypoint to limit the movement. This change is reflected throughout the layout and affects the position of other robot waypoints in the layout. There are only a few sensors. So a small displacement in the position of one sensor can cause a big difference in distance between the two adjacent robot waypoints. This in turn affects the objective function.



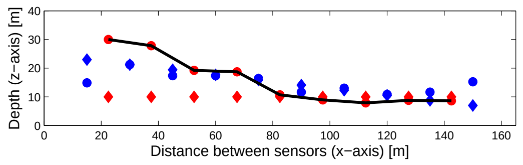
(a) Robot Waypoints initially in a straight line at depth 10m



(b) Robot Waypoints initially in zigzag patter between 7m and 23m



(c) Robot Waypoints initially in an ascending order from 7m to 23m

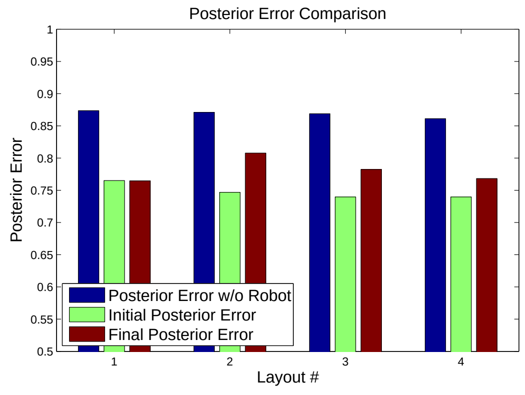


(d) Robot Waypoints initially in a descending order from 23m to 7m

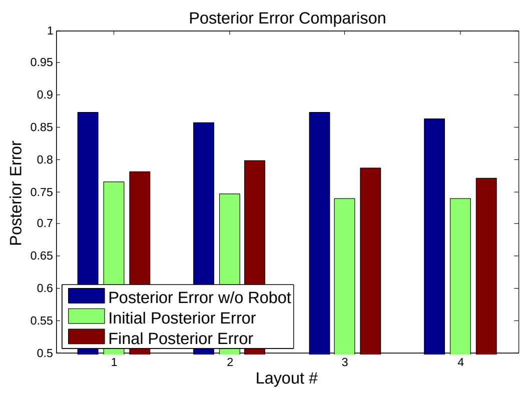
Figure 5.44: Initial and Final Sensor and Mobile Robot Path layout for varying Sensor start locations but fixed robot waypoints start location (Details: R=10, k=4000, hop=1, $\alpha = 0.10$, $\sigma_s = 5$,$\sigma_d = 10$)

Next, we ran the experiment with $V = 3$. The initial and final node layouts are shown in Figure 5.43. For the initial straight layout of robots and sensors (Figure 5.43(a)), we observed that the sensors and robot waypoints get evenly spaced out. For the zigzagged initial layout (Figure 5.43(b)), the robot waypoints tend to stay at the middle of the water column. The two other layouts, ascending and descending layout of sensors act mainly like before, i.e., the nodes closer to the surface climb up and the ones closer to the bottom of the water column go down. For this layout each of the algorithm converges much before the maximum number of iterations is reached. In the layout of Figure 5.43(a), the sensors and robot waypoints spread out evenly to cover the area of water. But in all the other layouts the robot waypoints stay close together, even though α is only 0.10. So these layouts can be one of the many local minima. The posterior error comparison can be seen in Figure 5.45(c).We can observe that the posterior error value of the final layout is less than that of the initial layout in all the cases.

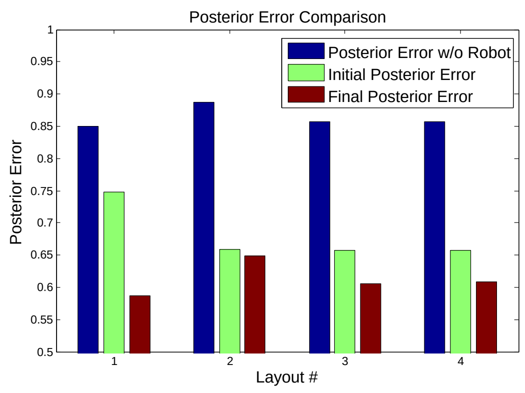
Finally, we ran the experiment with $V = 1$ and $[\sigma_s, \sigma_d] = [5, 10]$. In the other layouts $[\sigma_s, \sigma_d]$ were set to $[5, 4]$. The purpose of this experiment was to test how the layout of the



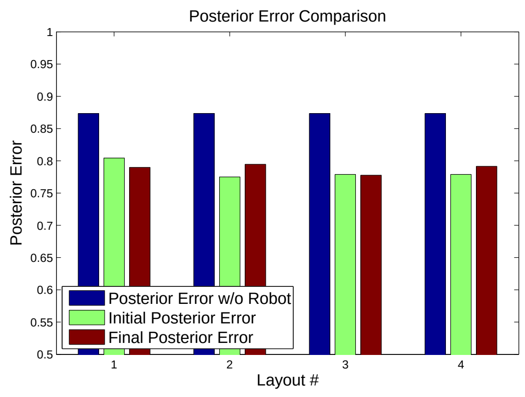
(a) Posterior Error ($iterations_{max} = 100$) for Figure 5.42



(b) Posterior Error ($iterations_{max} = 300$) for Figure 5.42



(c) Posterior Error ($iterations_{max} = 300$) for Figure 5.43



(d) Posterior Error ($iterations_{max} = 300$) for Figure 5.44

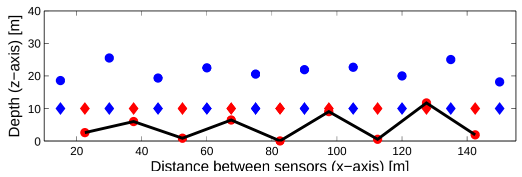
Figure 5.45: Posterior Error Comparison

nodes change when there is greater covariance along the depth of the water column. The initial and final node layouts are shown in Figure 5.44. In this scenario, for all the layouts, we observe that the robot waypoints stay closer to each other even though $V = 1$. The nodes hardly move from their initial positions. The reason may be since staying at the same position the nodes are now able to scan more depth along the water column and they do not need to change their depths to effectively distribute themselves. The posterior error values of the initial layout, the final layout and that of the final layout without the mobile robot sensing path is shown in Figure 5.45(d). We can observe a similar pattern as the other layouts with $V = 1$. However, the difference in posterior error values between the initial and final layouts is not significant. This is again because the nodes do not change from their initial positions by much.

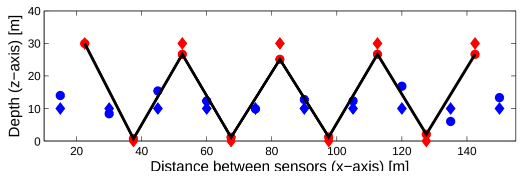
Table 5.9: Comparison of Starting Positions of Sensors where
$R = 10, threshold_{lim} = 10, turns_{lim} = 5, k = 4000, \alpha = 0.10, hops = 1$

Layout	**V**	**Iterations**	**Path Length**	**Posterior Error**		
				Initial	Final	Final w/o Robot
Straight	1	100	134.94	0.765	0.765	0.874
	1	300	187.72	0.765	0.781	0.873
	3	15	205.13	0.790	0.646	0.873
	1	300	187.72	0.804	0.788	0.875
Zigzag	1	100	164.79	0.747	0.808	0.871
	1	300	176.00	0.747	0.798	0.856
	3	35	146.76	0.702	0.672	0.877
	1	300	176.00	0.775	0.796	0.878
Ascending	1	100	147.44	0.740	0.782	0.869
	1	300	158.48	0.740	0.787	0.872
	3	15	143.34	0.657	0.605	0.857
	1	300	165.82	0.779	0.788	0.880
Descending	1	100	147.61	0.740	0.768	0.861
	1	300	148.68	0.740	0.771	0.863
	3	15	137.61	0.657	0.608	0.858
	1	300	145.23	0.779	0.790	0.874

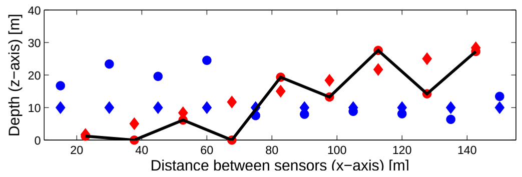
A comparison of the three different runs can be seen in Table 5.9. The posterior error value of the layouts without robot is even more than the value at the initial layout in all the experiments. This is because when the mobile robot sensing path is taken out we are removing all the robot waypoints from the water column. So, firstly, the number of sensing nodes decreases. Secondly, the algorithm is optimized such that sensor and robot waypoints together maximize the sensing in the region. When the robot waypoints are removed, the arrangement of the sensor positions by itself may not be ideal for sensing the information from the entire workspace.



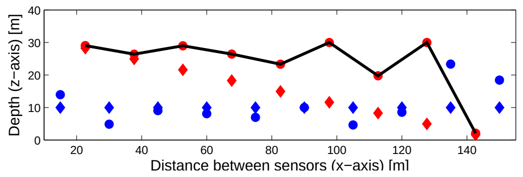
(a) Robot Waypoints initially in a straight line at depth 10m



(b) Robot Waypoints initially in zigzag patter between 7m and 23m



(c) Robot Waypoints initially in an ascending order from 7m to 23m



(d) Robot Waypoints initially in a descending order from 23m to 7m

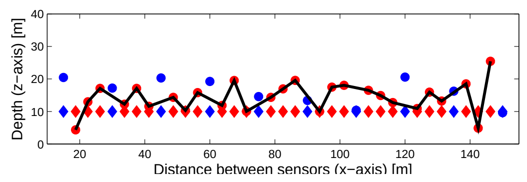
Figure 5.46: Initial and Final Sensor and Mobile Robot Path layout for fixed Sensor start locations but different robot waypoint start location (Details: R=10, k=4000, hop=1, maxIters=100, $\alpha = 0.10$, $\sigma_d = 5$)

5.9.8 Experiments on Starting Configuration of Mobile Robot Waypoints

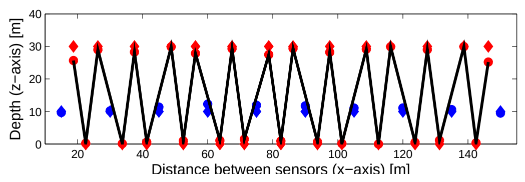
After experimenting with different starting layouts of sensors in Section 5.9.7, in this section we examine the effect of different starting layout of the robot waypoints while keeping the location of the sensors fixed. Again, we chose similar layouts for the robot waypoints so that the results can be compared with Section 5.9.7. In all the cases, the sensors were arranged at 10 meters depth of the water column. In between two sensors, a robot waypoint was placed at different depths depending on the layout. The four different starting layouts for the robot waypoints that were studied are given below:

- All robot waypoints arranged at 10 meters in a colinear fashion.
- Waypoints arranged in a zigzagged fashion at 30 meters and 0 meters alternately.
- Waypoints arranged in a monotonically ascending order from 0 meters to 30 meters.
- Waypoints arranged in a monotonically descending order from 30 meters to 0 meters.

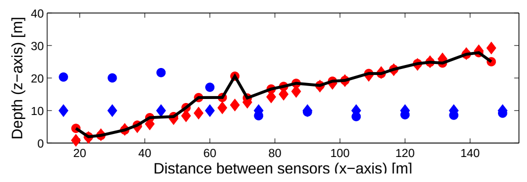
We test the different layouts with all same modifications that we tested for changing start position of sensors. For the first scenario we tried setting $iterations_{max} = 100$ and



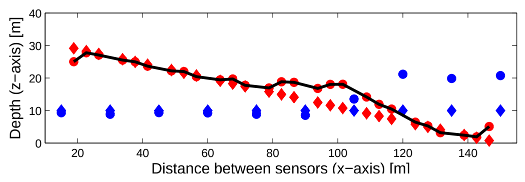
(a) Robot Waypoints initially in a straight line at depth 10m



(b) Robot Waypoints initially in zigzag patter between 7m and 23m



(c) Robot Waypoints initially in an ascending order from 7m to 23m



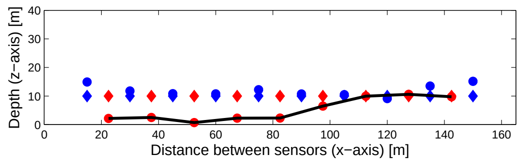
(d) Robot Waypoints initially in a descending order from 23m to 7m

Figure 5.47: Initial and Final Sensor and Mobile Robot Path layout for fixed Sensor start locations but different robot waypoint start location (Details: R=10, k=4000, hop=1, maxIters=300, $\alpha = 0.10$, $\sigma_d = 5$)

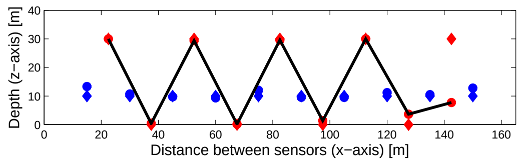
then with $iterations_{max} = 300$. The initial and final node layouts for these experiments are shown in Figure 5.46 and Figure 5.47 respectively.

In all cases, except zigzag layouts shown in Figure 5.46(b), and 5.47(b), we obtain comparable results to the final layouts described in Section 5.9.7. In the ascending robot waypoints scenario (Figure 5.46(c),5.47(c)), and descending robot waypoint scenario (Figure 5.46(d) and 5.46(d)), the robot waypoints are arranged in an ascending and descending fashion respectively. We observed that after optimization, the robot waypoints which were near the bottom of the water column go towards the bottom of the water column and those towards the surface reach closer to the surface. Since there is a distance constraint, the robot waypoints tend to stay close to each other, but the location of the waypoints are also controlled by the neighboring sensors on either side, so this is not very clear when $V = 1$.

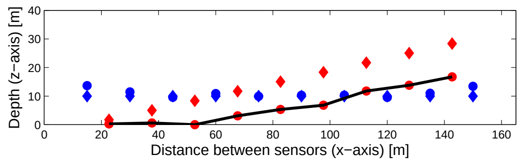
The posterior error values of the initial layout, the final layout and that of the final layout without the mobile robot sensing path is shown in Figure 5.49(a) and Figure 5.49(b) respectively. The posterior error value of the final layout is more than that of the initial layout for all the configurations except when the robot waypoints are arranged in a



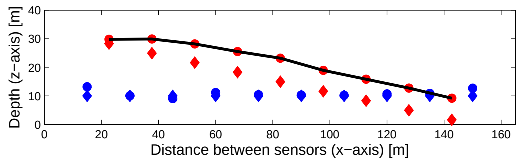
(a) Robot Waypoints initially in a straight line at depth 10m



(b) Robot Waypoints initially in zigzag patter between 7m and 23m



(c) Robot Waypoints initially in an ascending order from 7m to 23m



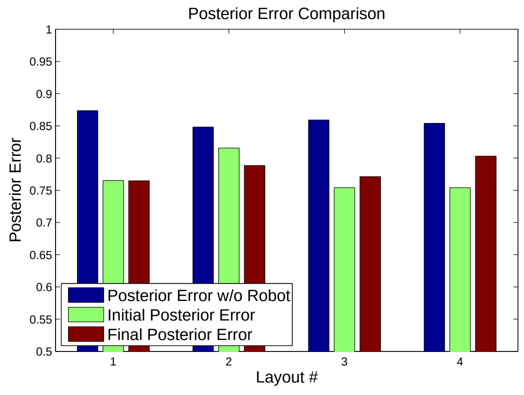
(d) Robot Waypoints initially in a descending order from 23m to 7m

Figure 5.48: Initial and Final Sensor and Mobile Robot Path layout for fixed Sensor start locations but different robot waypoint start location (Details: R=10, k=4000, hop=1, maxIters=100, $\alpha = 0.10$, $\sigma_d = 5$)

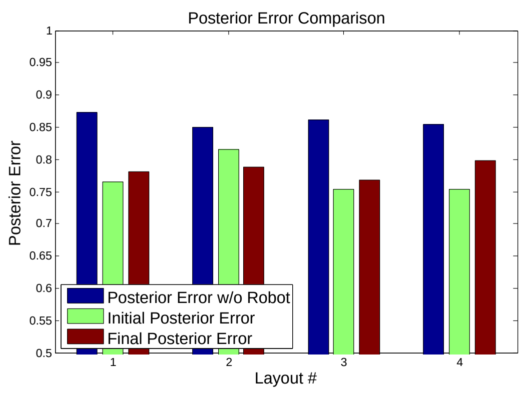
zigzagged fashion. For the same reasons as in the case of Section 5.9.7, the algorithm never converges and runs till $iterations_{max}$ is reached.

When we implemented the same set of experiments for $V = 3$, the algorithm converged soon. The initial and final node layout and posterior error comparison for the different layouts are depicted in Figure 5.47. Except the straight layout shown in Figure 5.47(a), for all other layouts the robot waypoints stay close together. Figure 5.47(b) is one of the layouts we have discussed extensively at the beginning of the section on AdaptivePath method of path planning. The posterior error values of the initial layout, the final layout and that of the final layout without the mobile robot sensing path is shown in Figure 5.49(c). In all cases the final layout has lesser posterior error value than the starting layout. For all the layouts, the algorithm converges.

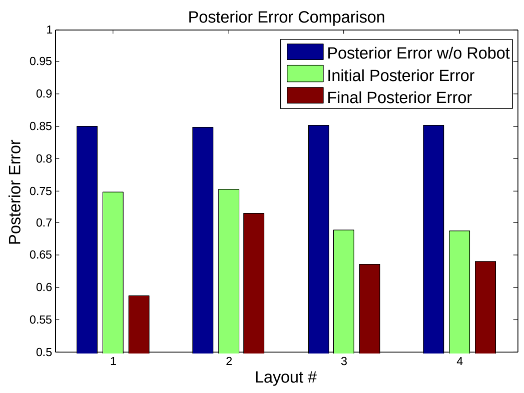
Finally, we executed the experiments for $V = 1$ and $[\sigma_s, \sigma_d] = [5, 10]$. The initial and final node layout and posterior error comparison for the different layouts are depicted in Figure 5.48. In this scenario, for all the layouts, we observe that the robot waypoints stay closer to each other even though $V = 1$. The sensors hardly move from their initial positions and the mobile robot path is greatly affected by the initial position of the first



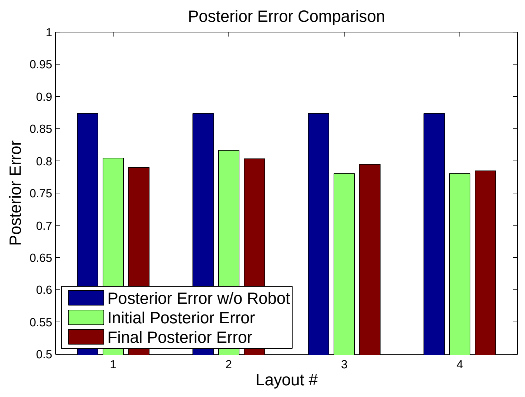
(a) Posterior Error ($iterations_{max} = 100$) for Figure 5.46



(b) Posterior Error ($iterations_{max} = 300$) for Figure 5.46



(c) Posterior Error ($iterations_{max} = 300$) for Figure 5.47



(d) Posterior Error ($iterations_{max} = 300$) for Figure 5.48

Figure 5.49: Posterior Error Comparison

robot waypoint. Since the σ_d had a high value, each node can scan greater depth along the water column. The posterior error values of the initial layout, the final layout and that of the final layout without the mobile robot sensing path are shown in Figure 5.49(d). We can observe a similar pattern as the other layouts with $V = 1$. However, the difference in posterior error values between the initial and final layouts is not significant because the nodes do not change their position very much from their initial positions. A comparison of the four different runs can be seen in Table 5.10.

Table 5.10: Comparison of Starting Positions of Robot Waypoints where $R = 10, threshold_{lim} = 10, turns_{lim} = 5, k = 4000, \alpha = 0.10, hops = 1$

Layout	**V**	**Iterations**	**Path Length**	**Posterior Error**		
				Initial	Final	Final w/o Robot
Straight	1	100	134.94	0.765	0.765	0.874
	1	300	187.72	0.765	0.781	0.873
	3	15	205.13	0.790	0.646	0.873
	1	300	187.72	0.804	0.788	0.875
Zigzag	1	100	236.14	0.816	0.788	0.848
	1	300	226.21	0.816	0.788	0.849
	3	37	742.32	0.717	0.673	0.873
	1	300	226.21	0.816	0.784	0.874
Ascending	1	100	148.84	0.754	0.771	0.859
	1	300	156.81	0.754	0.768	0.861
	3	15	142.32	0.688	0.636	0.851
	1	300	158.59	0.780	0.786	0.877
Descending	1	100	145.37	0.754	0.803	0.854
	1	300	147.51	0.754	0.798	0.854
	3	15	135.81	0.713	0.671	0.873
	1	300	149.63	0.780	0.791	0.874

5.9.9 Experiments on Number of Hops

In this section, we discuss the effect of the parameter *hops* on the running time, the mobile robot path length, and sensing efficiency of the layouts obtained after running Algorithm 4. For performing these sets of experiments, we considered a network with 10 sensors having 3 intermediate way points between each pair. All the sensors and the robot waypoints are arranged initially at 10m depth. The other parameters are set as $threshold_{lim} = 20$, $turns_{lim} = 5$, $k = 2000$ and $\alpha = [0.01, 0.10, 0.50]$ respectively. The maximum path length traversed by the mobile robot, the number of iterations required to converge, the initial and final objective function values, and the initial and final posterior error of the sensor network layout and the posterior error of the final layout without the mobile robot path are shown in Table 5.11 for comparison. We can observe that for all values of *hops*, the maximum path length traversed by the mobile robot is highest for $\alpha = 0.01$, followed by

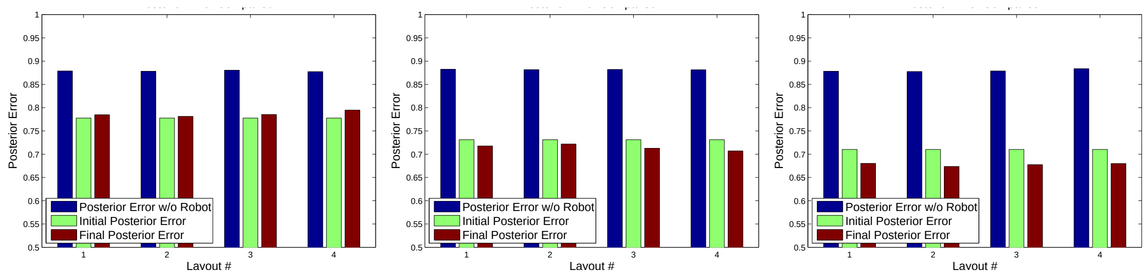
Table 5.11: Hops data for $R = 10, V = 3, iterations_{max} = 300, threshold_{lim} = 20, turns_{lim} = 5, k = 2000$ where h is *hops*, dist is *total path length*, i is *iterations to converge*

α	**h**	**dist**	**i**	**Objective Function**		**Posterior Error**		
				Initial	Final	Initial	Final	Final w/o Robot
0.01	1	189.82	25	10.4	0.005	0.748	0.592	0.851
	2	216.29	25	10.4	0.005	0.748	0.597	0.851
	3	187.85	25	10.4	0.005	0.748	0.604	0.849
	4	199.45	25	10.4	0.005	0.748	0.593	0.849
0.10	1	179.54	25	9.4	-0.013	0.748	0.586	0.851
	2	188.45	25	9.4	-0.013	0.748	0.602	0.851
	3	186.40	25	9.4	-0.013	0.748	0.607	0.850
	4	189.06	25	9.4	-0.013	0.748	0.591	0.850
0.50	1	147.18	25	5.1	-0.092	0.748	0.602	0.855
	2	156.23	30	5.1	-0.092	0.748	0.595	0.854
	3	148.00	27	5.1	-0.092	0.748	0.605	0.854
	4	168.34	25	5.1	-0.092	0.748	0.599	0.854

$\alpha = 0.10$ and then $\alpha = 0.50$, which confirms that the parameter *hops* does not change the way the algorithm functions. We expect a greater value of *hops* which implies that the algorithm will have better knowledge of the position of neighbors surrounding the particular node. As a result, more informed position will be chosen. However, the changes in the final objective function value and posterior error value for the different *hops* for same α is not very significant.

5.9.10 Experiments on a-priori deployed realistic Sensor Network

In this section, we took a network which has ten sensors all arranged at 10m depth of the water column. We sent this input to Algorithm 4 without any intermediate robot waypoints and get a layout of sensor locations which are optimized to maximize the information gain from the water column. Then, we added intermediate robot waypoints, $V = [1, 2, 3]$, to the network. These set of sensors and robot waypoints are then input to Algorithm 4. The algorithm optimizes the positions of the robot waypoints with respect to the positions of the sensors. At the same time, it optimizes the positions of the sensors with respect to the path of the mobile robot.



(a) $V = 1$ (b) $V = 2$ (c) $V = 3$

Figure 5.50: Posterior Error Comparison Section 5.9.10

Table 5.12: Comparison Table for Section 5.9.10
$R = 10, iterations_{max} = 100, threshold_{lim} = 20, turns_{lim} = 5, k = 2000$ and *dist* is *total path length*

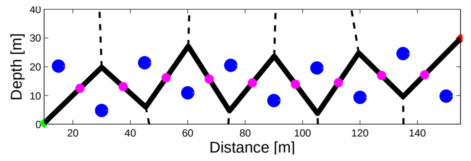
V	α	**iterations**	**Path Length**	**Objective Function**	
				Initial	Final
1	0.01	100	137.49	2.79	0.02
	0.05	100	128.03	2.68	0.02
	0.10	100	134.90	2.54	0.03
	0.50	41	129.11	1.41	0.04
2	0.01	100	138.06	3.67	0.01
	0.05	100	152.20	3.52	0.01
	0.10	100	149.43	3.34	0.01
	0.50	100	149.95	1.85	0.04
3	0.01	36	136.50	4.43	0.00
	0.05	53	159.23	4.24	0.00
	0.10	34	137.86	4.01	-0.01
	0.50	66	145.34	2.14	-0.09

For performing these sets of experiments we vary the values of α and V. The values of α chosen are $[0.01, 0.05, 0.10, 0.50]$ and that of V are $[1, 2, 3]$. The result of the experiment is represented in the Table 5.12. The Figure 5.50 shows the comparison of the posterior error values for the different layouts. Figure 5.50(a) represents the comparison of the posterior error value when $V = 1$, Figure 5.50(b) that of $V = 2$ and Figure 5.50(c) that of $V = 3$.

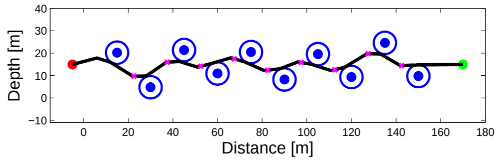
The Layout # 1 corresponds to $\alpha = 0.01$, Layout # 2 corresponds to $\alpha = 0.05$, Layout # 3 corresponds to $\alpha = 0.10$, and Layout # 4 corresponds to $\alpha = 0.50$ in all the figures. As discussed before, from the Figure 5.50(a) and the Table 5.12, we can conclude that $V = 1$ is not a stable configuration. For $V = 2$, even though the algorithm runs till maximum number of iterations for all the the layouts, the posterior error of the final Layout is much less than that of the initial layout. For this particular starting layout of sensors and robot waypoints, we can observe, the combination of V and α values that leads to good final layout are $V = 1, \alpha = 0.05$, $V = 2, \alpha = 0.10$, and $V = 3, \alpha = 0.05$, respectively.

5.10 Comparison

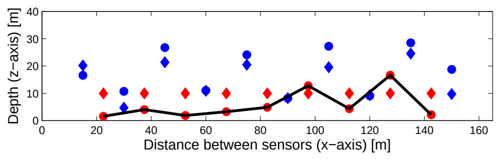
In this section we compare the three different path planning algorithms VoronoiPath, TanBugPath and AdaptivePath. We compare the algorithms with respect to one single layout of the sensors. The mobile robot will be deployed in an environment with an existing sensor network. To reproduce this scenario we assumed a network of ten sensors which we optimized with the help of Algorithm 4 with zero robot waypoints. Thus the ten sensors are distributed in the network such that the sensing is maximum in this layout. We now introduce the mobile robot in this scenario to improve the sensing even further. At first, we find out the different paths planned by the three different algorithms. Then we select horizontal locations in the water column and find out at what depth the mobile robot is positioned for the three different planned paths. Finally, we compare the algorithms with respect to the sensing



(a) VoronoiPath Method



(b) TanBugPath Method



(c) AdaptivePath Method

Figure 5.51: Compariosn with One Intermediate Robot Waypoint

efficiency which is measured by the value of the posterior error for the different layouts.

To successfully make comparison between the three algorithms we have to make sure that the conditions under which path planning is performed is same for all three methods. To make sure of this, we implement the algorithms under the following constraints and assumptions:

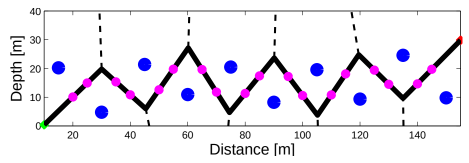
- The VoronoiPath algorithm does not have any parameter constraints since it does not depend on any input parameter that can affect the planned path. So we do not have to assume anything special for comparing the path obtained with the other algorithms.
- For TanBugPath algorithm we have two variables: the sensor view radius (r_S) and the robot view radius (r_R). For our AdaptivePath algorithm we assume identical sensor and robot waypoints. So here we assumed $r_S = r_R = 5m$.
- For the AdaptivePath algorithm we assumed $hops = 1, iterations_{max} = 300, threshold_{lim} = 20, turns_{lim} = 5, k = 2000, \alpha = 0.10$.

The path planned by AdaptivePath algorithm depends on the number of robot waypoints specified at the start of the algorithm. However, the path planned by VoronoiPath and TanBugPath algorithms have no such dependency. Therefore, we plan the path by AdaptivePath algorithm by assuming $V = [1, 2, 3]$ each and then we find out the locations in the water column which is comparable to these robot waypoints for VoronoiPath and TanBugPath algorithms. The different scenarios are discussed below.

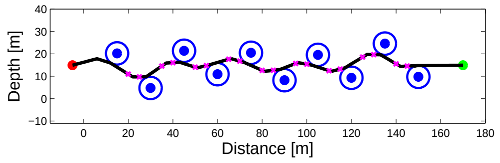
We considered a network of ten sensors which are already optimized by Algorithm 4 with $V = 0$. So the network consists of ten sensors and no robot waypoints. To be able to compare between the three different path planning algorithms, we chose the points in the mobile robot path where we have placed the robot waypoints for AdaptivePath algorithm. Wwhen $V = 1$, the positions of the robot waypoints in the water column start horizontally at 22.5m and proceed with gaps of 15m between each. The Figure 5.51 shows the initial and final layouts of the sensors and the mobile robot path for the three different algorithms.

The positions marked with a *majenta* colored circle are the robot sensing locations in case of the VoronoiPath and TanBugPath algorithms. The initial robot waypoints are represented with a red diamond and the final ones with a red circle for the AdaptivePath algorithm. Figure 5.54(a) shows the magnitudes of the posterior error calculated at the beginning and the end of the algorithm. We observe that the VoronoiPath algorithm gives us the best result, followed by TanBugPath method and finally, AdaptivePath algorithm.

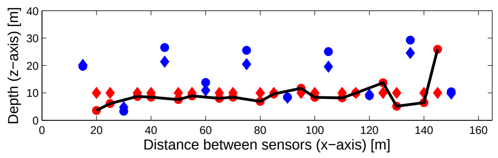
Again for $V = 2$, we considered the same network of ten sensors which are already optimized by Algorithm 4 with $V = 0$. When $V = 2$, the positions of the robot waypoints in the water column start horizontally at 20m and 25m, and then proceed increasing monotonically at gaps of 15m for each each. The Figure 5.52 shows the initial and final layouts of the sensors and the mobile robot path for the three different algorithms. Figure 5.54(b) shows the magnitudes of the posterior error calculated at the beginning and the end of the algorithm. We observe that the VoronoiPath algorithm gives us the best result, followed by TanBugPath method and finally, AdaptivePath algorithm.



(a) VoronoiPath Method



(b) TanBugPath Method



(c) AdaptivePath Method

Figure 5.52: Compariosn with Two Intermediate Robot Waypoint

For $V = 3$, once again, we considered the same network of ten sensors which we considered for $V = 1$ and $V = 2$. When $V = 2$, the positions of the robot waypoints in the water column start horizontally at 18.75m, 22.5m and 26.25m, and then proceed increasing monotonically at gaps of 15m for each each. The Figure 5.53 shows the initial and final layouts of the sensors and the mobile robot path for the three different algorithms. Figure 5.54(c) shows the magnitudes of the posterior error calculated at the beginning and the end of the algorithm. We observe that the VoronoiPath algorithm gives us the best result, followed by AdaptivePath algorithm and finally, TanBugPath method.

5.10.1 Comparison Summary

The three different algorithms has different strengths and weaknesses. Once algorithm can be better at a certain scenario compared to the others depending upon the environment, complexity of the mobile robot and other factor. In this section, we compare the algorithms based on the following factors:

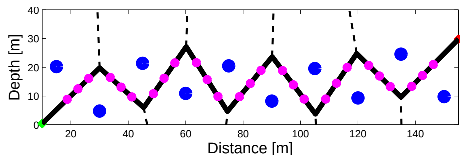
- A-Priori Network Knowledge
- Ease of Implementation
- Communication Requirement
- Energy Requirement
- Sensing Efficiency

A-Priori Network Knowledge: VoronoiPath is a global algorithm where the location of all sensors must be known at the beginning and it is computed centrally. The algorithm is run before the robot enters water, and the sensing locations are provided to the robot. Once the robot enters the water it cannot change these sensing locations. If positions of sensors change after the algorithm is run, the path of robot does not reflect this change.

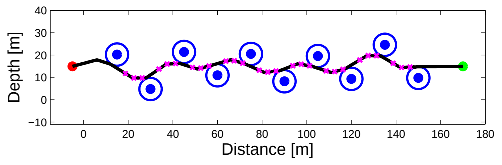
TanBugPath is a local path planning algorithm. This implies all processes for planning path of a robot through an underwater sensor network is done real-time on-board the robot. Thus this algorithm takes real-time changes in sensor locations into account while planing path. The robot should be equipped with efficient sensors and a processor for making real-time decisions.

AdaptivePath is an adaptive decentralized algorithm. Each of the sensors can independently decide their location in the water column. On introducing the robot waypoints, the sensor closest to the robot waypoints determine the best position for the robot waypoints along with its own position. Thus, the task of determining the good sensing locations is distributed between the sensors. After the robot enters the water, the nearest sensor communicates the immediate sensing locations to it through acoustic communication.

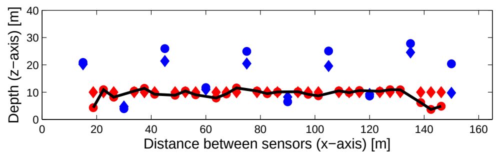
Ease of Implementation: In terms of ease of implementation, VoronoiPath planning is centralized and relatively easy to implement as long as the global knowledge of the sensors is available. When the robot enters the water, it already knows its path. So as long as the robot is able to localize itself, it will be able to traverse the planned path. Hence, it is easy to implement.



(a) VoronoiPath Method



(b) TanBugPath Method



(c) AdaptivePath Method

Figure 5.53: Compariosn with One Intermediate Robot Waypoint

This is followed by the AdaptivePath where each sensor runs an algorithm to update its depth locally. These sensors, at the same time, optimize the locations of nearest robot waypoints on either side. When the robot enters the water it should be able to communicate with the nearest sensor and move towards the specified robot waypoint. This can be done with acoustic communication and onboard localization.

TanBugPath is marginally more difficult to implement than AdaptivePath. The robot needs to communicate with the nearest sensor and needs to maintain a threshold distance with each sensor to plan the path. It also needs a method to localize itself in water. Finally, for planning a real-time path, it should have a computer with high processing power. These processes should all be tied together making it slightly difficult to implement.

Communication Requirement: VoronoiPath has the least communication requirement when the robot is inside the water column as it does not need to communicate with the sensors to learn about sensing locations. However, it needs information on all sensor locations beforehand, so in a real time environment all the sensors need to send this information to the central processing system before it departs. Therefore, we can say that it has the highest communication requirement. This is followed by TanBugPath and then AdaptivePath. For TanBugPath the robot communicates with nearest sensors to know positions of the next sensors. In AdaptivePath, the robot is dependent on sensors to know next sensing locations. Therefore, communication is crucial in this case. For all the algorithms, the robot needs to localize itself in the water column.

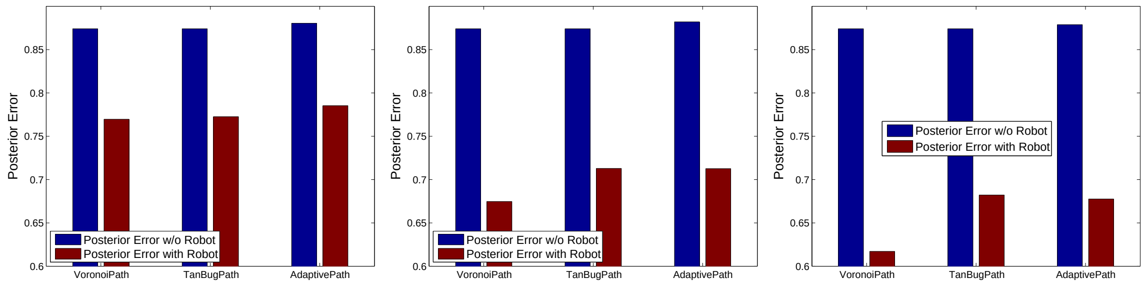
Table 5.13: Table for Comparison of maximum Path Length of Robot when it moves from one sensing location to the next one directly

V	Voronoi Path Length	Tangent Bug Path Length	Adaptive Path Length
1	120.62	125.84	134.90
2	151.19	131.32	149.43
3	166.50	134.90	137.86

Energy Requirement: An important factor in comparing the algorithms is energy efficiency of the planned robot paths. A zigzagged path is generally longer and consumes more energy than a relatively straight path. Thus, even in cases where a zigzagged path is sensing efficient, it may not be energy efficient. Often the *Voronoi* path is longer compared to *Tangent Bug* and *Adaptive* path for the same input sensor locations. As an example consider the layout shown in Figure 5.53. For VoronoiPath, the path depends on the location of the sensors only. The robot moves through the Voronoi vertices on its path and hence the total distance covered is **226.35m** for this layout. This distance can be shortened if the robot moves from one sensing location to the next and does not traverse the vertices. In TanBugPath, the distance depends on the robot start position, the sensing radii r_S and r_R, and *interval* size. In the example the distance covered by a robot is **139.77m**. For AdaptivePath, the distance depends on starting location of robot waypoints, number of intermediate robot waypoints V, the weight factor α and k value. The average distance for this layout is **140.73m**.

We find similar results for different layouts, although poor initial node configurations can cause AdaptivePath to get stuck in a sub-optimal local minima.

Sensing Efficiency: *Posterior error* is a common metric for defining how well an area is covered by sensors as discussed in Section 5.5. A lower value of posterior error signifies better sensing than a higher value. AdaptivePath is optimized with respect to an objective function. But VoronoiPath and TanBugPath algorithms do not have any objective function. So we compare the algorithms with respect to the *posterior error* of the final configuration of sensors and robot path. Figure 5.54 shows the plots of the posterior error values of the final configuration, with and without mobile robot, for three different robot sensing location configurations. It can be observed that the VoronoiPath method performs best for all V and performance of the TanBugPath algorithm is comparable to that of AdaptivePath algorithm. This was found on repeated experiments with different layouts.



(a) Posterior Error Comparison $V = 1$ (b) Posterior Error Comparison $V = 2$ (c) Posterior Error Comparison $V = 3$

Figure 5.54: Compariosn with One Intermediate Robot Waypoint

We can observe that posterior error value of the final configuration without robot is different for AdaptivePath, than VoronoiPath and TanBugPath. In AdaptivePath, the sensors present in network adapt their positions depending on robot's path. This is highly desirable. Finally, we can observe that, for all algorithms, increasing the number of sensing

Table 5.14: Comparison of Path Planning Algorithms

Measure	Voronoi Path	Tangent Bug Path	Adaptive Path
A-Priori Network Knowledge	Centralized	Centralized	Decentralized
Ease of Implementation	Easy	Complicated	Moderate
Communication Requirement	Least/Highest	Moderate	High
Energy Requirement	Longer path length	Moderate Path length	Length depends on start layout, V and α
Sensing Efficiency	Highest	Comparable to ADAPTIVEPATH	Comparable to TANBUGPATH
Change in Sensor Locations	Not taken into account	Taken into account	Taken into account
Sensor Position	Remains Static	Remains Static	Changes based on robot path

locations decreases posterior error value.

Table 5.14 summarizes these. The table also shows that since VORONOIPATH precomputes the path of the robot before its enters water, any changes in sensor locations is not taken into account after the path has been decided. For TANBUGPATH and ADAPTIVEPATH any immediate changes in sensor locations is considered. However, in case of VORONOIPATH and TANBUGPATH, the traversal of the robot does not have any effect on the positions of sensors already present in the water column. The ADAPTIVEPATH algorithm, on the other hand, rearranges the position of the sensors based on the robot's path to improve the overall sensing. This is an important feature of this algorithm.

5.11 Summary

In this Section, we simulate three different path planning algorithms for planning the path of a mobile underwater robot through a network of semi-mobile sensors. The path planning algorithms are: 1) Global Path Planning with VoronoiPath method, 2) Local Path Planning with the help of TanBugPath method, and 3) Adaptive path Planning with the AdaptivePath method. We simulate all the algorithms using the software Matlab. We perform experiments on these algorithms by varying different parameters. Finally, we compare the paths planned by the three algorithms with respect to a particular sensor network. Next, in Chapter 6, we present the contributions of this thesis, the lessons learned and the future work.

Chapter 6

Conclusion

The application of wireless sensor networks and robots to the domain of underwater exploration has huge potential for monitoring the health of river and marine environments. Monitoring these environments manually is costly and difficult. If a sensor network is deployed underwater, it could help us monitor variables such as water temperature, pressure, conductivity, turbidity, pollutant and also the behavior of underwater ecosystems.

In this thesis we describe an underwater sensor network system that consists of semi-mobile sensors and a mobile robot. Then we introduce three different path planning algorithms for planning the path of the mobile robot through the network of the semi-mobile sensors. The three algorithms are VoronoiPath, TanBugPath and AdaptivePath. VoronoiPath is a global, TanBugPath is a local and AdaptivePath is a decentralized algorithm. The mobility of the underwater robots enhances the performance of the system and results in better information gain from the area of water column. Also with the help of the mobile robot large areas can be covered more efficiently with sparse sensor networks. In TanBugPath and AdaptivePath, we show that the mobile robot does not need global information, it can travel from one node to the next to know the locations that it should be sensing next. In such cases communication is possible only when the robot and the sensors are in close proximity to each other.

We implemented the algorithms in simulation, then presented a detailed analysis and comparison.

6.1 Thesis Contributions

This thesis makes a number of theoretical contributions, including:

- Three different path planning algorithms for traversal of a mobile robot through a network of semi-mobile sensors which improve sensing through a area of water column. The algorithms are:
  - A global path planning algorithm, VoronoiPath;
  - A local path planning algorithm, TanBugPath;
  - A decentralized path planning algorithm, AdaptivePath. We prove that the algorithm is convergent.
- Analysis and comparison of the three algorithms, based on implementation method, ease of implementation, energy requirement, communication requirement and sensing efficiency.

This thesis also contributes to the field from an experiment perspective:

- Simulations verifying the performance of the VoronoiPath, TanBugPath and AdaptivePath algorithms;
- Experiments and implementation of the three algorithms showing their performance under different input parameters.

6.2 Future Work

The research presented in this thesis can be taken in different directions in the future. Firstly, since all these algorithms consider a 2D slice of the water column we can expand these algorithms so that they can be used to plan the path of a mobile robot in a 3D environment. We can do this by either extending these algorithms to include the case where more than one mobile robot can be used to sense from different slices of the water column, or by making one single mobile robot scan horizontally as well as vertically along the 3D slice of the water column to improve the overall sensing.

Another interesting direction for future work, will be to design the algorithms in such a way that the mobile robot return to the point in the water column from where it starts. Right now, the mobile robot starts from a starting point and moves towards the goal which is either a point just before or just after the last sensor in the network. While deploying this algorithm in real world, this would mean that the user has to estimate where the robot is going to end up in the water column and collect it from there. If we can design the algorithms such that the robot comes back to the starting point then the problem of deployment will become much simpler. Lastly, one of the most important applications of this thesis is to implement the algorithms discussed here in the real world, compare performances of the robots, and analyze if they match the theoretical perspective.

Finally, the goal of this thesis has been to introduce a method of underwater monitoring using semi-mobile underwater sensor networks and mobile underwater robots. We hope that the system and algorithms presented in this thesis will motivate future research in underwater robotics.

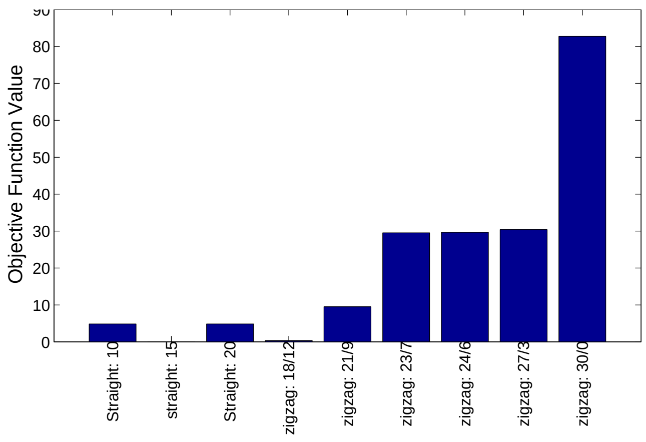
Appendix A

Figure A.1 shows the objective function comparison for the layouts shown in Figure 5.2 in Section 5.6.1.

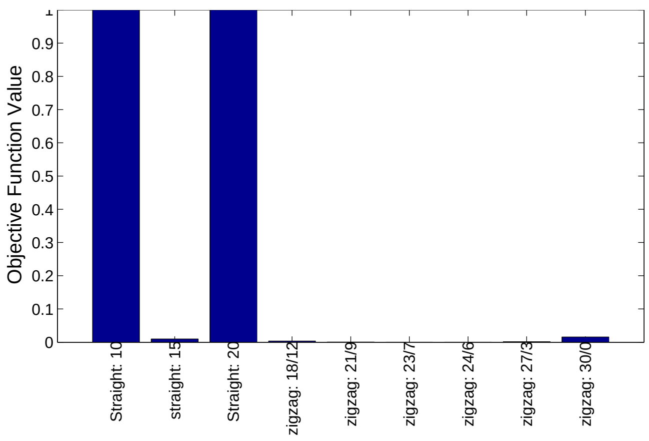
Figure A.2 shows the objective function comparison for the layouts shown in Figure 5.4 in Section 5.6.2.

Figure A.3 and A.4 show the objective function comparison for the layouts shown in Figure 5.6 in Section 5.6.3.

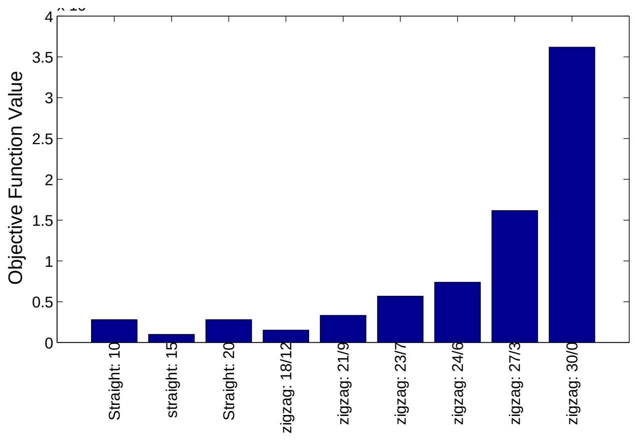
The rest of this chapter contains the remaining functions that are needed by the TanBugPath algorithm.



(a) Objective Function ($\sigma_s, \sigma_d = 5, 4$)

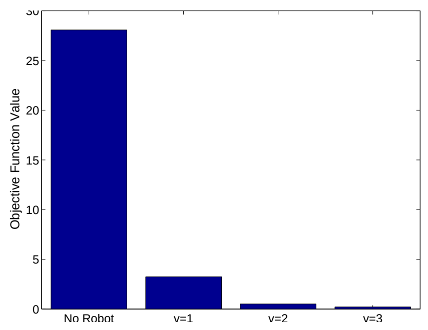


(b) Objective Function ($\sigma_s, \sigma_d = 10, 4$)

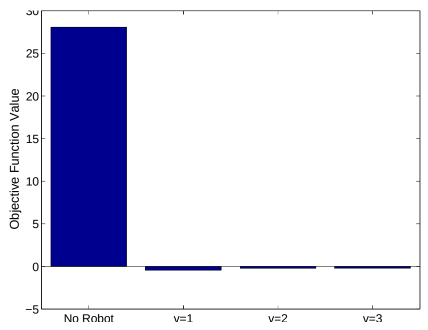


(c) Objective Function ($\sigma_s, \sigma_d = 5, 10$)

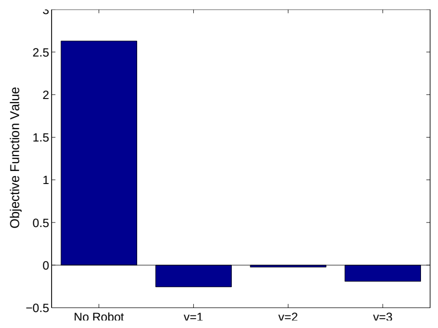
Figure A.1: Comparison of Objective Function for layouts shown in Figure 5.2



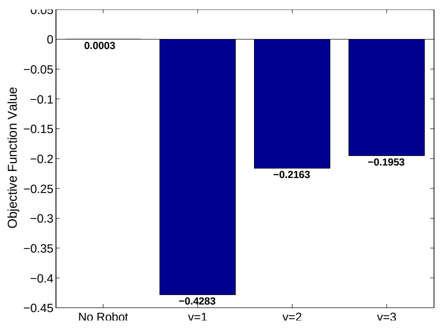
(a) $weight = 0, (\sigma_s, \sigma_d = 5, 4)$



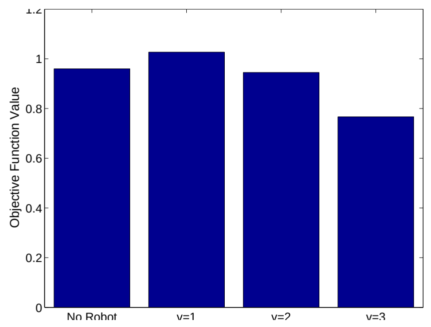
(b) $weight = 1, (\sigma_s, \sigma_d = 5, 4)$



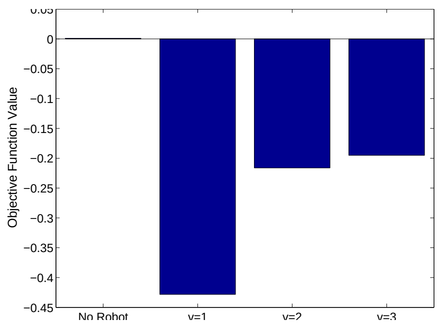
(c) $weight = 0, (\sigma_s, \sigma_d = 10, 4)$



(d) $weight = 1, (\sigma_s, \sigma_d = 10, 4)$

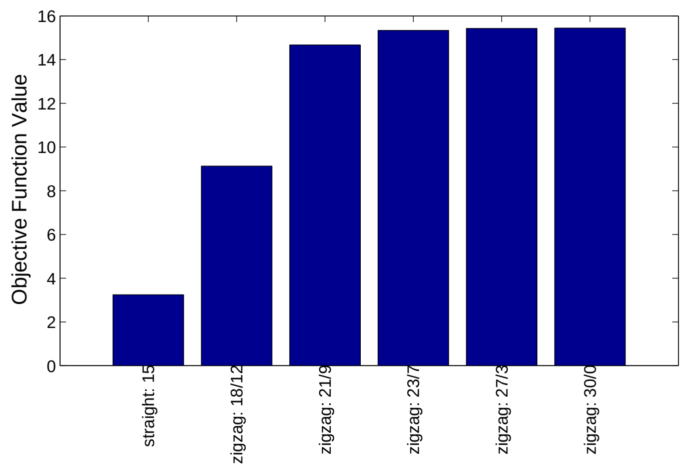


(e) $weight = 0, (\sigma_s, \sigma_d = 5, 10)$

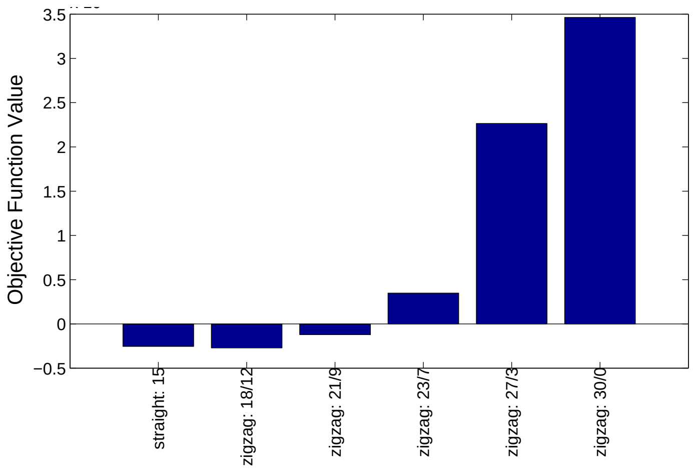


(f) $weight = 1, (\sigma_s, \sigma_d = 5, 10)$

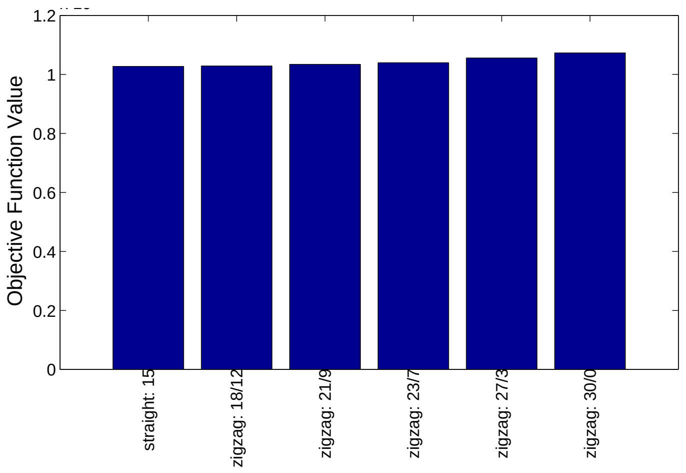
Figure A.2: Comparison of Objective Function for layouts shown in Figure 5.4



(a) $weight = 1, (\sigma_s, \sigma_d = 5, 4)$

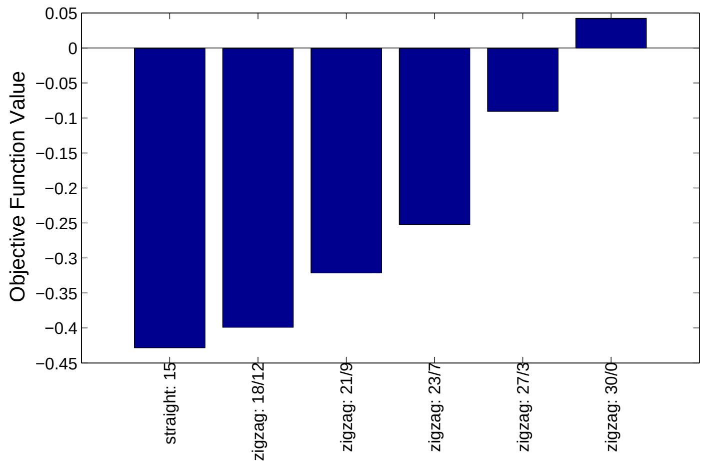


(b) $weight = 1, (\sigma_s, \sigma_d = 10, 4)$

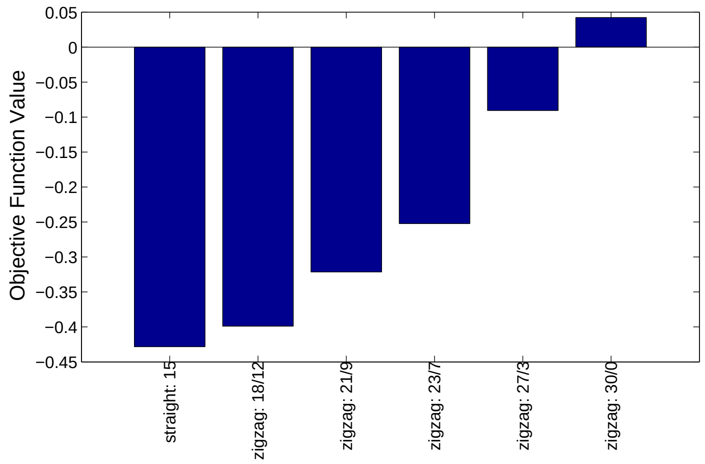


(c) $weight = 1, (\sigma_s, \sigma_d = 5, 10)$

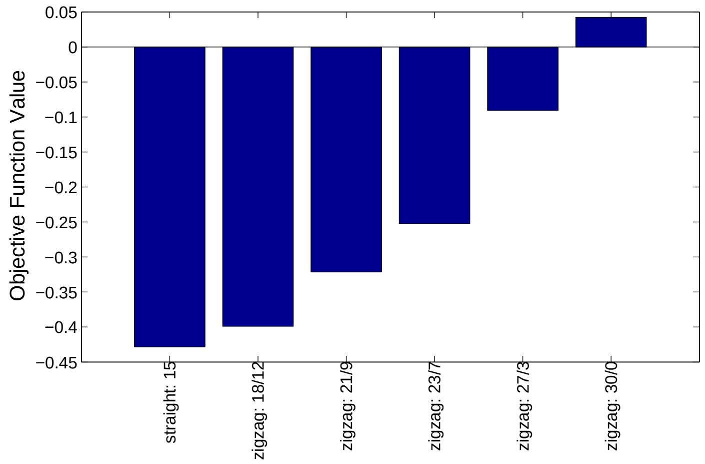
Figure A.3: Comparison of Objective Function for $\alpha = 0.00$ for layouts shown in Figure 5.6



(a) $weight = 1, (\sigma_s, \sigma_d = 5, 4)$



(b) $weight = 1, (\sigma_s, \sigma_d = 10, 4)$



(c) $weight = 1, (\sigma_s, \sigma_d = 5, 10)$

Figure A.4: Comparison of Objective Function for $\alpha = 1.00$ for layouts shown in Figure 5.6

Algorithm 5 TanBugPath: Remaining Functions

```
Require: Original Sensor layout, $sensors$; Sensor View Radius, $r_S$; Robot View Radius,$r_R$; Robot Start position, $p_{start}$; Robot Destination,$p_{end}$
Ensure: Mobile Robot Way Points, $path^F_{points}$;


procedure FindNextPos($target$, $robot$, $interval$)
    $slope \leftarrow \frac{target_Y - robot_Y}{target_X - robot_X}$
    $intercept \leftarrow robot_Y - slope * robot_X$
    $robot_{newX} \leftarrow robot_X + interval$
    $robot_{newY} \leftarrow slope * robot_{newX} + intercept$
    $robot_{new} \leftarrow [robot_{newX}, robot_{newY}]$
    return $robot_{new}$
end procedure

procedure FindTangentPoints($p_{robot}$,$p_{target}$,$r_R$,$r_S$)
    $c_1 \leftarrow [p_{target}, r_S]$
    $hyp \leftarrow$ dist($p_{robot}$,$p_{target}$)
    $r_{new} \leftarrow \sqrt{|hyp^2 - {r_S}^2|}$
    $c_2 \leftarrow [p_{robot}, r_{new}]$
    $[flag, t_1, t_2] \leftarrow$ cir2Intersection($c_1, c_2$)
    return $[flag, t_1, t_2]$
end procedure

procedure IfMoveAlongCurve($S_{near}$, $r_S$, $p_{target}$, $p_{robot}$)
    $circle \leftarrow [S_{near}, r_S]$
    $line \leftarrow [p_{robot}; p_{target}]$
    $[test, p_1, p_2] \leftarrow$ cirLineIntersection($circle, line$)
    $moveAlongCurveFlag \leftarrow$ false
    if $test == 1$ then
        if (dist($p_{robot}, p_1$) $== 0$ AND dist($p_{target}, p_1$) $>$ dist($p_{target}, p_2$)) OR (dist($p_{robot}, p_2$) $== 0$ AND dist($p_{target}, p_1$) $<$ dist($p_{target}, p_2$)) then
            $moveAlongCurveFlag \leftarrow$ true
        end if
    end if
    return $moveAlongCurveFlag$
end procedure
```

Algorithm 5 TanBugPath: Remaining Functions (cont.)

```
33:
Require: circle1($cir_{1X}, cir_{1Y}, cir_{1R}$): robot position; circle2($cir_{2X}, cir_{2Y}, cir_{2R}$): target position
Ensure: $flag, i_1, i_2$
34:
35: procedure Cir2Intersection($circle_1$, $circle_2$)
36:     $dist \leftarrow \sqrt{(cir_{1X} - cir_{2X})^2 + (cir_{1Y} - cir_{2Y})^2}$
37:     $total_R \leftarrow cir_{1R} + cir_{2R}$
38:     $diff_R \leftarrow |cir_{1R} - cir_{2R}|$
39:     if $dist > (cir_{1R} + cir_{2R})$ then        ▷ The 2 circles do not intersect; No intersection.
40:         $flag \leftarrow -3$
41:         $i_1, i_2 \leftarrow [0,0]$
42:     else if $dist < |cir_{1R} - cir_{2R}|$ then        ▷ One circle lies inside the other circle; No intersection.
43:         $flag \leftarrow -2$
44:         $i_1, i_2 \leftarrow [0,0]$
45:     else if $(dist == 0)$ AND $(cir_{1R} == cir_{2R})$ then        ▷ The 2 circles are co-incident; Infinite solutions.
46:         $flag \leftarrow -1$
47:         $i_1, i_2 \leftarrow [0,0]$
48:     else        ▷ The circles intersect at 2 points
49:         $flag \leftarrow 1$
50:         $i_1 \leftarrow$ first intersection point.
51:         $i_2 \leftarrow$ second intersection point.
52:     end if
53:     return $flag, i_1, i_2$
54: end procedure
55:
Require: Line: Connects the 2 points $(x_1, y_1, z_1)$ and $(x_2, y_2, z_2)$; Circle: Center $(x_3, y_3, z_3)$ and Radius $r$
Ensure: $flag; Point_1; Point_2$
56:
57: procedure CirLineIntersection($circle$, $line$)
58:     $a \leftarrow (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$
59:     $b \leftarrow 2((x_2 - x_1).(x_1 - x_3) + (y_2 - y_1).(y_1 - y_3) + (z_2 - z_1).(z_1 - z_3))$
60:     $c \leftarrow x_3^2 + y_3^2 + z_3^2 + x_1^2 + y_1^2 + z_1^2 - 2[x_3.x_1 + y_3.y_1 + z_3.z_1] - r^2$
61:     $t \leftarrow b^2 - 4ac$
62:     if $t < 0$ then        ▷ The line does not intersect the circle.
63:         $flag \leftarrow -1$
64:         $p_1, p_2 \leftarrow [0,0]$
65:     else if $t == 0$ then        ▷ The line is a tangent to the circle. One intersection point at $u = -b/2a$.
66:         $flag \leftarrow 0$
67:         $p_1 \leftarrow$ intersection point.
68:         $p_2 \leftarrow [0,0]$
69:     else        ▷ The line is a secant to the given circle. 2 intersection points.
70:         $flag \leftarrow 1$
71:         $p_1 \leftarrow$ first intersection point.
72:         $p_2 \leftarrow$ second intersection point.
73:     end if
74:     return $flag, p_1, p_2$
75: end procedure
```

Bibliography

[1] http://www.cs.columbia.edu/ allen/f13/notes/chap2-bug.pdf. (document), 4.3, 4.4

[2] Jerome Barraquand, Bruno Langlois, and J-C Latombe. Numerical potential field techniques for robot path planning. *Systems, Man and Cybernetics, IEEE Transactions on*, 22(2):224–241, 1992. 2.2

[3] Priyadarshi Bhattacharya and Marina L Gavrilova. Roadmap-based path planning-using the voronoi diagram for a clearance-based shortest path. *Robotics & Automation Magazine, IEEE*, 15(2):58–66, 2008. 2.5

[4] Rodney A Brooks. Solving the find-path problem by good representation of free space. *Systems, Man and Cybernetics, IEEE Transactions on*, (2):190–197, 1983. 2.2

[5] James Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, volume 3, pages 2383–2388. IEEE, 2002. 2.2

[6] Francesco Bullo, Jorge Cortés, and Sonia Martınez. Distributed control of robotic networks. *Applied Mathematics Series. Princeton University Press*, 2009. 3.3.7, 4.4.6

[7] N Buniyamin, WAJ Wan Ngah, N Sariff, and Z Mohamad. A simple local path planning algorithm for autonomous mobile robots. *International journal of systems applications, Engineering & development*, 5(2):151–159, 2011. 2.6

[8] Chenghui Cai and Silvia Ferrari. Information-driven sensor path planning by approximate cell decomposition. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 39(3):672–689, 2009. 2.2

[9] Kevin P Carroll, Stephen R McClaran, Eric L Nelson, David M Barnett, Donald K Friesen, and GN William. Auv path planning: an a* approach to path planning with consideration of variable vehicle speeds and multiple, overlapping, time-dependent exclusion zones. In *Autonomous Underwater Vehicle Technology, 1992. AUV'92., Proceedings of the 1992 Symposium on*, pages 79–84. IEEE, 1992. 2.4

[10] F. Celeste, F. Dambreville, and J.-P. Le Cadre. Optimal path planning using cross-entropy method. In *Information Fusion, 2006 9th International Conference on*, pages 1–8, July 2006. 2.3

[11] Han-Lim Choi and Jonathan P How. Continuous motion planning for information forecast. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 1721–1728. IEEE, 2008. 2.2

[12] Han-Lim Choi and Jonathan P. How. Continuous trajectory planning of mobile sensors for informative forecasting. *Automatica*, 46(8):1266–1275, August 2010. 2.2

[13] Howie Choset. *Sensor based motion planning: The hierarchical generalized Voronoi graph*. PhD thesis, Citeseer, 1996. 2.5

[14] Howie Choset and J. Burdick. Sensor based planning, part i: The generalized voronoi graph. In *Proceedings of the 1995 IEEE International Conference on Robotics and Automation (ICRA '95)*, volume 2, pages 1649 – 1655, May 1995. 2.5

[15] Howie Choset and J. Burdick. Sensor based planning, part ii: Incremental construction of the generalized voronoi graph. In *Proceedings of the 1995 IEEE International Conference on Robotics and Automation (ICRA '95)*, volume 2, pages 1643 – 1648, May 1995. 2.5

[16] Howie Choset, Marco La Civita, and Jong Chai Park. Path planning between two points for a robot experiencing localization error in known and unknown environments. 1999. 2.5

[17] Howie Choset, Sean Walker, Kunnayut Eiamsa-Ard, and Joel Burdick. Sensor-based exploration: incremental construction of the hierarchical generalized voronoi graph. *The International Journal of Robotics Research*, 19(2):126–148, 2000. 2.5

[18] Christopher I Connolly, JB Burns, and R Weiss. Path planning using laplace's equation. In *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pages 2102–2106. IEEE, 1990. 2.2

[19] Carrick Detweiler, Sreeja Banerjee, Marek Doniec, Mingshun Jiang, Francesco Peri, Robert F Chen, and Daniela Rus. Adaptive decentralized control of mobile underwater sensor networks and robots for modeling underwater phenomena. *Journal of Sensor and Actuator Networks*, 3(2):113–149, 2014. (document), 3.1

[20] Carrick Detweiler, Marek Doniec, Mingshun Jiang, Mac Schwager, Robert Chen, and Daniela Rus. Adaptive decentralized control of underwater sensor networks for modeling underwater phenomena. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 253–266. ACM, 2010. 1, 2.1, 3.1, 3.3, 3.3.4, 3.3.7, 4.4.6, 5.1, 5.5

[21] Carrick Detweiler, Marek Doniec, Iuliu Vasilescu, Elizabeth Basha, and Daniela Rus. Autonomous depth adjustment for underwater sensor networks. In *Proceedings of the Fifth ACM International Workshop on UnderWater Networks*, page 12. ACM, 2010. 3.2

[22] Carrick Detweiler, Marek Doniec, Iuliu Vasilescu, and Daniela Rus. Autonomous depth adjustment for underwater sensor networks: design and applications. *Mechatronics, IEEE/ASME Transactions on*, 17(1):16–24, 2012. 3.2, 3.2

[23] Carrick Detweiler, John Leonard, Daniela Rus, and Seth Teller. *Passive mobile robot localization within a fixed beacon field*. Springer, 2008. 3.2

[24] Carrick Detweiler, Stefan Sosnowski, Iuliu Vasilescu, and Daniela Rus. Saving energy with buoyancy and balance control for underwater robots with dynamic payloads. In *Experimental Robotics*, pages 429–438. Springer, 2009. 3.2

[25] Carrick Carrick James Detweiler. *Decentralized sensor placement and mobile localization on an underwater sensor network with depth adjustment capabilities*. PhD thesis, Massachusetts Institute of Technology, 2010. 1, 2.1, 3.1, 3.3, 3.3.4, 3.3.7, 4.4.6, 5.1

[26] Carrick Detweiller, Iuliu Vasilescu, and Daniela Rus. An underwater sensor network with dual communications, sensing, and mobility. In *OCEANS 2007-Europe*, pages 1–6. IEEE, 2007. 3.2, 3.2

[27] Marek Doniec, Carrick Detweiler, and Daniela Rus. Estimation of thruster configurations for reconfigurable modular underwater robots. In *Experimental Robotics*, pages 655–666. Springer, 2014. 3.2

[28] G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Computational Principles of Mobile Robotics. Cambridge University Press, 2010. 4.3

[29] Kikuo Fujimura and Hanan Samet. A hierarchical strategy for path planning among moving obstacles [mobile robot]. *Robotics and Automation, IEEE Transactions on*, 5(1):61–69, 1989. 2.2

[30] S. Garrido, L. Moreno, M. Abderrahim, and F. Martin. Path planning for mobile robot navigation using voronoi diagram and fast marching. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 2376–2381, Oct 2006. 2.5

[31] Shuzhi Sam Ge and Yan Juan Cui. New potential functions for mobile robot path planning. *IEEE Transactions on robotics and automation*, 16(5):615–620, 2000. 2.2

[32] Shuzhi Sam Ge, Xuecheng Lai, and A.A. Mamun. Boundary following and globally convergent path planning using instant goals. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 35(2):240–254, April 2005. 2.6

[33] S.S. Ge, Y. Cui, and Cishen Zhang. Instant-goal-driven methods for behavior-based mobile robot navigation. In *Intelligent Control. 2003 IEEE International Symposium on*, pages 269–274, Oct 2003. 2.6

[34] Elmer G Gilbert and Daniel W Johnson. Distance functions and their application to robot path planning in the presence of obstacles. *Robotics and Automation, IEEE Journal of*, 1(1):21–30, 1985. 2.2

[35] Carlos Guestrin, Andreas Krause, and Ajit Paul Singh. Near-optimal sensor placements in gaussian processes. In *Proceedings of the 22nd international conference on Machine learning*, pages 265–272. ACM, 2005. 2.1, 5.5

[36] III Hoff, K., T. Culver, J. Keyser, M.C. Lin, and D. Manocha. Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 3, pages 2931–2937 vol.3, 2000. 2.5

[37] Yong K Hwang and Narendra Ahuja. A potential field approach to path planning. *Robotics and Automation, IEEE Transactions on*, 8(1):23–32, 1992. 2.2

[38] B.J. Julian, M. Angermann, and D. Rus. Non-parametric inference and coordination for distributed robotics. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 2787–2794, Dec 2012. 2.1

[39] Brian J. Julian, M. Angermann, M. Schwager, and D. Rus. A scalable information theoretic approach to distributed robot coordination. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 5187–5194, Sept 2011. 2.1

[40] Brian J Julian, Michael Angermann, Mac Schwager, and Daniela Rus. Distributed robotic sensor networks: An information-theoretic approach. *The International Journal of Robotics Research*, 31(10):1134–1154, 2012. 2.1

[41] I. Kamon, E. Rivlin, and E. Rimon. A new range-sensor based globally convergent navigation algorithm for mobile robots. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 429–435 vol.1, Apr 1996. 2.6

[42] Lydia Kavraki and J-C Latombe. Randomized preprocessing of configuration for fast path planning. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 2138–2145. IEEE, 1994. 2.2

[43] Thomas Kollar and Nicholas Roy. Efficient optimization of information-theoretic exploration in slam. In *AAAI*, volume 8, pages 1369–1375, 2008. 2.2

[44] Andreas Krause and Carlos Guestrin. Near-optimal observation selection using submodular functions. In *AAAI*, volume 7, pages 1650–1654, 2007. 2.1

[45] J. LaSalle. Some extensions of liapunov's second method. *Circuit Theory, IRE Transactions on*, 7(4):520–527, Dec 1960. 3.3.7, 4.4.6

[46] S.L. Laubach, J. Burdick, and L. Matthies. An autonomous path planner implemented on the rocky 7 prototype microrover. In *Robotics and Automation, 1998. Proceedings. 1998 IEEE International Conference on*, volume 1, pages 292–297 vol.1, May 1998. 2.6

[47] Naomi Ehrich Leonard, Derek A Paley, Francois Lekien, Rodolphe Sepulchre, David M Fratantoni, and Russ E Davis. Collective motion, sensor networks, and ocean sampling. *Proceedings of the IEEE*, 95(1):48–74, 2007. 2.4

[48] Juan Liu, Feng Zhao, and D. Petrovic. Information-directed routing in ad hoc sensor networks. *Selected Areas in Communications, IEEE Journal on*, 23(4):851–861, April 2005. 2.3

[49] Kian Hsiang Low, John M Dolan, and Pradeep Khosla. Adaptive multi-robot wide-area exploration and mapping. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1*, pages 23–30. International Foundation for Autonomous Agents and Multiagent Systems, 2008. 2.2

[50] Kian Hsiang Low, John M Dolan, and Pradeep K Khosla. Information-theoretic approach to efficient adaptive path planning for mobile robotic environmental sensing. In *ICAPS*, 2009. 2.2

[51] Daniel R. Lynch and Dennis J. McGillicuddy Jr. Objective analysis for coastal regimes. *Continental Shelf Research*, 21(1112):1299 – 1315, 2001. 1, 3.3.5

[52] Keiji Nagatani, Yosuke Iwai, and Yutaka Tanaka. Sensor based navigation for car-like mobile robots using generalized voronoi graph. In *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, volume 2, pages 1017–1022. IEEE, 2001. 2.5

[53] Hiroshi Noborio, Ryo Nogami, and Satoru Hirao. A new sensor-based path-planning algorithm whose path length is shorter on the average. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on*, volume 3, pages 2832–2839. IEEE, 2004. 2.2

[54] Yvan Petillot, I Tena Ruiz, and David M Lane. Underwater vehicle obstacle avoidance and path planning using a multi-beam forward looking sonar. *Oceanic Engineering, IEEE Journal of*, 26(2):240–251, 2001. 2.4

[55] Clement Petres, Yan Pailhas, Pedro Patron, Yvan Petillot, Jonathan Evans, and David Lane. Path planning for autonomous underwater vehicles. *Robotics, IEEE Transactions on*, 23(2):331–341, 2007. 2.4

[56] M. Schwager. *A Gradient Optimization Approach to Adaptive Multi-Robot Control*. PhD thesis, Massachusetts Institute of Technology, September 2009. 3.3.7, 4.4.6

[57] Ryan N Smith, Mac Schwager, Stephen L Smith, Burton H Jones, Daniela Rus, and Gaurav S Sukhatme. Persistent ocean monitoring with underwater gliders: Adapting sampling resolution. *Journal of Field Robotics*, 28(5):714–741, 2011. 2.4

[58] Anthony Stentz. Optimal and efficient path planning for partially known environments. In *Intelligent Unmanned Ground Vehicles*, pages 203–220. Springer, 1997. 2.2

[59] Ruben Stranders, Alessandro Farinelli, Alex Rogers, and Nicholas R Jennings. Decentralised coordination of mobile sensors using the max-sum algorithm. In *IJCAI*, volume 9, pages 299–304, 2009. 2.3

[60] O. Takahashi and R.J. Schilling. Motion planning in a plane using generalized voronoi diagrams. *Robotics and Automation, IEEE Transactions on*, 5(2):143–150, Apr 1989. 2.5

[61] Sebastian Thrun and Arno Bücken. Integrating grid-based and topological maps for mobile robot navigation. In *Proceedings of the National Conference on Artificial Intelligence*, pages 944–951, 1996. 2.2

[62] John Tisdale, Zuwhan Kim, and J Karl Hedrick. Autonomous uav path planning and estimation. *Robotics & Automation Magazine, IEEE*, 16(2):35–42, 2009. 2.1

[63] Iuliu Vasilescu, Carrick Detweiler, Marek Doniec, Daniel Gurdan, Stefan Sosnowski, Jan Stumpf, and Daniela Rus. Amour v: A hovering energy efficient underwater robot capable of dynamic payloads. *The International Journal of Robotics Research*, 29(5):547–570, 2010. 3.2

[64] Iuliu Vasilescu, Carrick Detweiler, and Daniela Rus. Aquanodes: an underwater sensor network. In *Proceedings of the second workshop on Underwater networks*, pages 85–88. ACM, 2007. 3.2

[65] C Vasudevan and K Ganesan. Case-based path planning for autonomous underwater vehicles. *Autonomous Robots*, 3(2-3):79–89, 1996. 2.4

[66] Yunfeng Wang and Gregory S Chirikjian. A new potential field method for robot path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 977–982. IEEE, 2000. 2.2

[67] Charles W Warren. A technique for autonomous underwater vehicle route planning. *Oceanic Engineering, IEEE Journal of*, 15(3):199–204, 1990. 2.4

[68] C.W. Warren. Global path planning using artificial potential fields. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 316–321 vol.1, May 1989. 2.2

[69] Shangming Wei. Smooth path planning and control for mobile robots. 2005. 2.6

[70] Namik Kemal Yilmaz, Constantinos Evangelinos, Pierre FJ Lermusiaux, and Nicholas M Patrikalakis. Path planning of autonomous underwater vehicles for adaptive sampling using mixed integer linear programming. *Oceanic Engineering, IEEE Journal of*, 33(4):522–537, 2008. 2.4

[71] Xiaoping Yun and Ko-Cheng Tan. A wall-following method for escaping local minima in potential field based motion planning. In *Advanced Robotics, 1997. ICAR'97. Proceedings., 8th International Conference on*, pages 421–426. IEEE, 1997. 2.2

[72] Guoxian Zhang, Silvia Ferrari, and M Qian. An information roadmap method for robotic sensor path planning. *Journal of Intelligent and Robotic Systems*, 56(1-2):69–98, 2009. 2.2

[73] Jie Zhao, Zi-wei Zhou, Ge Li, He Zhang, Wang-bao Xu, et al. The apposite way path planning algorithm based on local message. In *Mechatronics and Automation (ICMA), 2012 International Conference on*, pages 1563–1568. IEEE, 2012. 2.2