Fiducial Marker Navigation for Mobile Robots

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Science (Honours)

of Rhodes University

Luke Ross

Grahamstown, South Africa
November 2012

Abstract

Regarding mobile robots, navigation using fiducial markers has already been achieved and obstacle avoidance using search algorithms is common. However, the combination of these two ideas is relatively uncommon, and therefore, is the focus of this research project. Owing to the mobility of these robots, good performance of the system is of utmost importance. The relevant options available for the creation of this hybrid system are investigated, and an efficient system, capable of meeting its objectives, is designed and implemented on the WifibotLab LIDAR mobile robot. Under testing, the system showed good performance and proved accurate in terms of marker detection and the routes determined. Several variables were identified that can be adjusted to increase accuracy and/or performance even further. Although various setbacks were encountered, the creation of a system usable in practical scenarios was successful, and therefore viable for integration and use on other similarly equipped mobile robots.

ACM Computing Classification System Classification

Thesis classification under ACM Computing Classification System (1998 version, valid through 2012)

I.2.9 [Robotics]: Autonomous vehicles, Sensors
I.2.8 [Problem Solving, Control Methods, and Search]: Heuristic methods
I.4.8 [Scene Analysis]: Tracking
I.5.4 [Applications]: Computer vision

General-Terms: Algorithms, Performance

Acknowledgements

I would like to acknowledge the financial and technical support of Telkom, Tellabs, Stortech, Eastel, Bright Ideas Project 39, and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

I would also like to acknowledge the financial contribution made to me by the National Research Foundation (NRF) towards this research. While the ideas expressed and conclusions reached in this thesis are my own, the NRF's assistance has been greatly appreciated and has made this thesis possible.

A special thank you to my parents, Allan and Nicky Ross, for their support and funding, not only for this year, but throughout my academic career. Your guidance and help is much appreciated and without you, this thesis could never have been started.

To my fiancée, Melinda Dicks. To you I owe the greatest thanks. Your support and advice throughout this year has been greatly appreciated. I would also like to thank you for aiding with the thesis where you could and for volunteering to proofread. Any grammatical errors found within this thesis are not owing to her, but are faults of my own.

Lastly, I would like to thank my supervisor, Karen Bradshaw, for the guidance and encouragement throughout the year. Thank you for making this project such an enjoyable experience. Without you, this thesis would not be what it is.

Contents

1 Introduction **1**

1.1 Background 1

1.2 Research Objectives 1

1.3 Thesis Organisation 2

2 Background **3**

2.1 Introduction 3

2.2 WifibotLab LIDAR 4

2.3 Fiducial Marker Detection 5

2.3.1 ARToolKit 6

2.3.2 ArUco - Based on OpenCv 8

2.4 Search Algorithms 9

2.4.1 A* 11

2.4.2 D* (Dynamic A*) 14

2.4.3 Generalized Fringe-Retrieving A* 16

2.4.4 Moving Target D* Lite 16

2.5 Summary 17

3 System Design 19

3.1 Introduction . . . 19

3.2 Fiducial Marker Tracking . . . 19

3.3 Search Algorithms . . . 22

3.4 Robot Movement . . . 23

3.4.1 Accessing Robot Functionality . . . 23

3.4.2 Follow Marker Design . . . 24

3.4.3 Obstacle Avoidance . . . 26

3.4.4 Searching . . . 31

3.5 Putting it all together . . . 31

3.6 Summary . . . 34

4 System Implementation 35

4.1 Introduction . . . 35

4.2 Accessing Robot Functionality . . . 35

4.3 Detection and Following . . . 40

4.4 Obstacle Avoidance . . . 42

4.5 Searching . . . 50

4.6 Using the System . . . 51

4.7 Summary . . . 52

5 Results and Analysis **53**

5.1 Introduction 53

5.2 Marker Detection using ArUco 53

5.2.1 Marker Size and Camera Resolution 54

5.2.2 Detection Performance 56

5.2.3 Automatic Lighting 58

5.3 Occupancy Grid Design 59

5.3.1 Area Represented 59

5.3.2 Occupancy Grid Resolution 63

5.4 Path Calculation 67

5.4.1 Comparison of A* Implementations 67

5.4.2 Number of Adjacent Cells Considered and Heuristics 68

5.5 Final Decisions and Limitations 70

5.6 Summary 74

6 Conclusions **75**

6.1 Project Summary 75

6.2 Revisiting the Objectives 75

6.3 Future Work 76

A Code Listings **81**

A.1 Methods Used in Issuing Move Commands 81

A.2 A* Implementation 82

List of Figures

3.1 Markers created with ArUco 20

3.2 Detection with ArUco 24

3.3 LIDAR capabilities 27

3.4 Rectangle used for checking path ahead 27

3.5 Occupancy grid showing blocking obstacles 29

3.6 State diagram of mode switching 32

3.7 Information flow between looping threads 33

4.1 Detailed representation of rectangle 43

4.2 Conversion process 46

5.1 Distance versus marker size for different resolutions 55

5.2 Detection at different orientations 55

5.3 Standard layout used for experiments 60

5.4 Exaggerated maps using different scan ranges 62

5.5 Exaggerated maps using different cell sizes - scenario 1 65

5.6 Exaggerated maps using different cell sizes - scenario 2 65

5.7 Invalid path calculated when using a cell size of 440 mm and considering diagonal movement 66

5.8 Paths deviate when movement is restricted to four directions 69

5.9 Different paths calculated when four and eight cells are considered 70

5.10 Example layout for demonstration 72

5.11 System in action 72

5.12 Unsuccessful layouts 73

List of Tables

5.1 Resolution effect on performance 57

5.2 Scan range effect on performance 61

5.3 Grid resolution effect on performance 64

5.4 Performance comparison between A* implementations 67

5.5 Effect on performance based on the number of cells considered 69

Listings

4.1 Move robot forward continuously 36

4.2 The *robotmove* thread 37

4.3 Pan/tilt controls 38

4.4 Request and retrieval of LIDAR data 39

4.5 Dectection using ArUco 40

4.6 Implementation of various decisions 41

4.7 Rectangle array value calculation 43

Chapter 1

Introduction

1.1 Background

Fiducial marker navigation on mobile robots has been implemented successfully by researchers in various applications such as monitoring underwater domains using amphibious robots [24]. However, many implementations of these kinds of systems do not optimize the paths that the mobile robots travel. In most robot navigation systems in which minimal paths are calculated whilst avoiding obstacles, target tracking is not done using fiducial markers; instead methods based on GPS coordinates are more commonly used.

This project aims to add to the areas of mobile robot navigation and obstacle avoidance techniques. Since most mobile robots have low to moderate processing power owing to battery life constraints, significant emphasis is placed on the efficiency of the system built.

1.2 Research Objectives

First, an investigation into the tools available for marker detection and the search algorithms suited to this project is conducted. Using this contextual information, decisions regarding the design of the system are made, after which the system is implemented. This system must satisfy certain criteria and therefore the objectives for this research are as follows:

- Detect fiducial markers accurately using a standard web camera.

- Navigate a mobile robot using a fiducial marker in a clear and obstacle-filled environment.
- Ensure that the system performs well, and is therefore suitable for less powerful robots.

1.3 Thesis Organisation

The subsequent chapters of this thesis are organised as follows:

Chapter 2 discusses literature on fiducial marker detection and obstacle avoidance. This literature consists of previous research conducted in these areas as well as the various options available for designing the proposed system.

Chapter 3 covers the design of the system. The reasoning behind the design decisions made are explained, and the structure of the system is described.

Chapter 4 covers the implementation of the system and presents code listings of the fundamental sections.

Chapter 5 compares the accuracy of the system with its performance when varying certain parameters regarding marker detection, occupancy grid design and search algorithm design. Furthermore, it describes the major setbacks encountered during this project.

Chapter 6 gives a brief summary of this thesis, revisits the goals of this project and, lastly, mentions the possibilities for future work.

Chapter 2

Background

2.1 Introduction

This chapter reviews literature on the WifibotLab mobile robot, fiducial markers, the toolkits available for fiducial marker detection and various search algorithms that can be used for robot path planning. Current systems that implement fiducial markers for navigational purposes as well as systems that implement efficient path planning are also discussed.

The chosen toolkit for marker detection needs to be reasonably accurate and, owing to the processing and memory constraints on many mobile robots, not too computationally demanding. In addition, the chosen search algorithm needs to have the following properties:

- It must be able to calculate the optimal path from the robot to the marker, avoiding obstacles.
- It must be able to function effectively in known, partially known and unknown environments.
- It must be able to function effectively with a moving target.
- It must be sufficiently computationally efficient for use on mobile robots.

2.2 WifibotLab LIDAR

WifibotLab LIDAR is a mobile robot intended for educational purposes as well as the development of affordable robotic systems[1]. The robot's specifications are as follows:

- Commell LE-376 Embedded Intel Atom Miniboard
- Intel Atom D510, 1.66 GHz, 1 MB cache
- 1 GB DDR2-667 memory
- Logitech QuickCam ® Sphere AF
- 5.6 m range LIDAR (URG-04LX-UG01)
- 2 x Infrared (IR) sensors
- Atheros AR5413 a/b/g WiFi card
- 4 x 12 V motors
- 2 x Hall encoders
- 12 V NiMh, 9000 mAH Battery with charger

As listed above, the robot is equipped with a LIDAR (Light Detection And Ranging) sensor. This sensor uses light (most often laser pulses) to measure the distances to objects in its surroundings[2]. Since the Wifibot robot is well equipped and has high processing capabilities, it was the robot chosen for testing the system.

As this high-end mobile robot is being used for experimentation in this project, more common, lower specification robots must be considered when choosing between the different libraries and algorithms available. Wifibot has been used successfully in various systems in the past, two of which are mentioned. The first is the PotPet robot [10]. This robot is a mobile flowerpot that allows users to grow plants in a more effective and enjoyable way. It was built using a Wifibot platform owing to its good mobility. PotPet is autonomous and automatically moves into areas with more sunlight as well as towards people when it requires more water. Demetriou et al. [5] created a second system, which

[1]http://www.wifibot.com/wifibot-wifibotlab.html
[2]http://forsys.cfr.washington.edu/JFSP06/lidar_technology.htm

allows for mobile robot localization using WiFi signal strength measurements from a number of access points. This system provides a low-cost localization method for robotic applications in indoor environments where GPS is unavailable.

2.3 Fiducial Marker Detection

Fiducial Markers

Fiducial, or fiduciary markers are image like objects, which are designed to be detectable and which usually contain an interpretable meaning [37]. Fiducials are used in many fields, including augmented reality, robotics and medicine. In robot navigation systems, an autonomous robot can follow a set path with the aid of fiducial markers [18]. Such markers offer performance, identification and localization improvements, and are more cost-effective when compared to other techniques used for path-planning in unknown environments [18]. In the medical field, fiducial markers are used when treating prostate cancer. During the treatment, doctors need to have an accurate view of the prostate gland and since this view is often not sufficient, gold fiducial markers are placed in the prostate to enhance vision[3].

Fiducial markers can vary from small dots to complicated bar-code images and can be of various shapes [20]. According to Owen et al. [20] and verified by Li et al. [16], two-dimensional, bar-coded, square fiducial markers are the best and most popular form of marker to use. Since these are the types of markers used for augmented reality or robot navigation systems [8], they seem appropriate. Examples of these types of markers are ARToolKit or ArUco markers. ARToolKit and ArUco markers consist of a black border and a black and white, uniquely patterned interior. The border of such a marker aids in the initial detection of the marker, whilst the interior pattern is used for identification of the marker [20]. The pattern must be unique [20] and as stated above, may have meaning encoded into its graphical representation. Technology such as a robot with an attached camera and running the required software can easily interpret such a pattern. Owing to the square nature of the marker, its position and orientation can be calculated accurately with respect to a calibrated camera [20].

Important Design Criteria

[3]http://clinicaltrials.gov/ct2/show/NCT00061347

Owen et al. [20] state that, in order for fiducial markers to be most accurately recognized, various factors should be considered when designing the markers. Firstly, the shape of the fiducial should emit at least four points. The simplest of these shapes is a square and owing to its simplicity, computational advantages are noted. In addition, the fiducial image can contain most colours, but it is best if a monochrome colour is used. This is because a monochrome image is easier to recognize against bright, contrasting backgrounds. Using a monochrome colour also gives computational advantages as the algorithms are simplified. Thirdly, the size of the marker is dependent on the resolution of the camera being used, with the border of the marker constituting at least 13% of the marker's width. Other important and more technical criteria concerning effective marker design, according to Owen et al. [20] are as follows:

- There should be no ambiguity when determining the marker's position and orientation relative to a camera.
- Markers should not favour certain orientations.
- If multiple markers are used in the system, they must all be unique.
- Simple algorithms that are not intensive should be used to locate and identify the marker quickly. For these algorithms to be used, the marker itself needs to be designed with this intent in mind.

Libraries used for Detecting Fiducial Markers

Libraries that can be used for marker detection include ARTag[4], ARToolKit[5] and ArUco[6]. According to ARTag's homepage, it is currently unavailable as a research aid and therefore is not discussed further for use in this project.

2.3.1 ARToolKit

ARToolkit was first released in 1999 by Dr Hirokazo Kato, after which its further development was maintained by the Human Interface Technology Laboratory [1].

ARToolKit is a successful, robust marker based system, commonly used in the augmented

[4] http://www.artag.net/index.html
[5] http://www.hitl.washington.edu/artoolkit/
[6] http://www.uco.es/investiga/grupos/ava/node/26

reality field [7]. It is C and C++ language oriented, although Java and Matlab are also supported. ARToolKit can be used to detect pre-programmed ARToolKit markers in images and augment three-dimensional virtual objects onto these markers [1]. However, the marker detection functionality of the software can be used on its own without implementing any of the augmented reality functionality. Once a marker is identified, ARToolKit returns the marker ID as well as the location of the four corners of the marker relative to the camera device [6]. ARToolKit markers are two-dimensional and planar, and consist of a unique, black and white, patterned interior which is surrounded by a black border [7]. The large contrast between the black and white colours on the markers aids in ARToolKit's robustness [7].

Advantages

Two of ARToolKit's advantages with greatest relevance to this project are that it is open source for non-commercial use and widely used [6]. Since it is widely used, there is extensive research material available. Furthermore, according to the feature list on the ARToolKit home page [1], ARToolKit supports multiple platforms including Windows, Linux, Mac OS X and SGI; it has real-time detection for two-dimensional markers; and there is sufficient documentation. The documentation available includes installation instructions, a simple calibration procedure as well as executable sample code. ARToolKit has precise detection, is easy to use and is well suited for the implementation of low-cost tracking systems, as the only added hardware requirement is a camera [2].

Disadvantages

Hornecker and Psik [9] point out some disadvantages of using ARToolKit, the first of which being that it does not recognize a marker if the full marker is not in the camera's view. Secondly, owing to its large black border, when the marker is printed on certain laser printers, light may be reflected causing inconsistencies in the video feed. The lighting issue can be resolved by printing the markers on an ink-jet printer [9]. Moreover, since ARToolKit is threshold based [8], a chosen threshold may not detect markers in different lighting conditions. Hirzer [8] further claims that ARToolKit has a high false positive rate.

History of Use

ARToolKit has been used extensively in marker tracking and augmented reality and, according to the "Projects" tab on the ARToolKit homepage, is used globally by over 300 researchers for a wide variety of purposes. Past applications that use ARToolKit

vary from creating virtual environments in which ancient artifacts can be restored and viewed [25], to tracking human body movements [23].

2.3.2 ArUco - Based on OpenCv

The search for literature on ArUco yielded few results, perhaps owing to its fairly recent release. As a result, a large section of the literature presented below was found in online documents such as the ArUco homepage[7].

ArUco was developed by Rafael Munoz-Salinas from the University of Cordoba and released in November 2010 under a BSD license[8]. It is a basic C++ library used for the detection of fiducial markers and intended for augmented reality purposes [3]. It is based on OpenCv (Open Source Computer Vision), a vision based library, which is the most popular library in the computer vision field [3][4].

Features

According to ArUco's homepage, some of the features that ArUco offers are listed below:

- Markers can be detected using one line of C++ code.
- Use can be made of AR boards (a grid of markers) to increase detection accuracy.
- There are 1024 different ArUco markers available.
- Since it is based on OpenCv, ArUco detects markers quickly and reliably and is cross-platform.
- Examples and sample code are available.
- It is low-cost owing to possession of a BSD license.

ArUco also provides various applications with the library, two of which are relevant to this project [3]. The first application creates a marker, given an identification number, and saves this as a jpg file for printing. The second application, which is the main application, detects markers in a live video feed or pre-recorded video. Camera calibration is also

[7] http://www.uco.es/investiga/grupos/ava/node/26
[8] http://softwaredd.net/softs/linux/games/aruco.html

possible using OpenCv.

History of Use

Speers et al. [24] reported using an amphibious, autonomous robot to monitor underwater sensors. In this system, fiducial markers are displayed on sessile sensors and are used to communicate with the robot using the ArUco library. ArUco has also been used in various unpublished projects, some of which are listed on the ArUco homepage. These projects include: the Soldamatic Project[9], the purpose of which is to aid in the training of welders by using augmented reality to create welding simulations, as well as OpenSpace3D[10], which is used for developing interactive, real-time 3D projects.

2.4 Search Algorithms

Requirements for Project

The robot in this project needs to be able to travel towards the fiducial marker in an environment that may contain obstacles. Since the robot should travel along the shortest path, a suitable search algorithm needs to be implemented such that the optimal path, avoiding obstacles, from the robot's current position to the marker is chosen. Furthermore, an efficient search algorithm needs to be implemented to keep computation to a minimum. Incremental search algorithms, some of which are presented below, are search algorithms that use previous search information when searching for minimal traversals and therefore can perform faster than searching from scratch [32]. These algorithms are popular for robot path planning[11].

Most research literature concerning path planning for mobile robots does not take navigation in unknown environments into account but rather, it is assumed that the entire map is initially known [27]. In this project, the robot will be placed in an environment which may be partially known or completely unknown as well and will therefore need to acquire map information using its LIDAR sensor. An algorithm that functions in unknown environments is therefore required. Since the marker will be moving whilst the robot attempts to travel towards it, the moving target search problem needs to be taken into account as well. The moving target search problem is a path planning problem, often

[9] http://seabery.es/simulador-de-soldadura-soldamatic/?lang=en
[10] http://www.openspace3d.com/
[11] http://www-scf.usc.edu/ xiaoxuns/research.html

encountered in computer games, where a hunter attempts to catch a moving target [32].

Representing the Environment

A specific robot configuration can be represented using a state, and the distance between two states can be represented using an arc. Therefore, a graph of these states, joined together by arcs, can represent a robot's environment. The search algorithms discussed use this representation [32][28]. The robot's environment, or map information, needs to be stored internally in a structure on the robot. This will allow the robot to update its map information whenever it is acquired from its sensors, plan an optimal path and know where to move. Two common structures that can be used to store map information are metric and topological maps.

Metric maps are usually implemented using occupancy grids. According to Thrun and Bucken [35], in the area of mobile robotics, occupancy grids are the most successful environmental representations. This is verified by Kraetzschmar et al. [13]. Although two-dimensional occupancy grids are most common, three-dimensional grids have also been implemented [33]. These grids represent the robot's environment using a matrix of cells, each of which contains an occupancy value [33][35]. This occupancy value is the probability that the cell is occupied [35]. Kuffner [14] states that many systems use grid-based maps to internally represent a robot's environment and then search for optimal paths from start to goal states using the grid's embedded graph. Occupancy maps are easily implemented and simple to use [13][34]. There is also no ambiguity in determining the robots position in its grid from its actual position in the environment [34]. Occupancy maps have the disadvatage of generating maps that are often inaccurate and do not represent sections of the environment correctly [33]. These inconsistencies generally occur in busy environments, unlike the environment used in this project. Another disadvantage of using grid-based maps such as occupancy grids is that, when a high resolution grid is required, memory requirements can become an issue [22][13].

Topological maps represent the environment using a graph [12]. This graph consists of a collection of nodes connected by arcs. The nodes correspond to distinctive landmarks and the arcs correspond to the distances between these landmarks. Unlike occupancy grids, the resolution of a topological map is purely dependent on the complexity of the environment. This allows for faster planning and less memory requirements than when using an occupancy grid [34]. Topological maps have disadvantages in that they are difficult to implement and maintain in large evironments and that similar landmarks are

often ambiguously recognized, resulting in inaccurate map information [34][35].

Both grid-based and topological maps have their unique advantages and disadvantages. Therefore, researchers have integrated the two types of maps by generating topological maps on top of grid-based maps [35][22]. Firstly, the grid-based map is partitioned into smaller regions. These regions are separated by criticial lines. An algorithm is then used to map this partitioned map into a topological graph with each region corresponding to a node and each critical line corresponding to an arc. This hybrid allows for accurate map representations as well as efficient planning. [35]

History of Use

Search algorithms have been used extensively in the fields of Computer Science and Robotics in a wide variety of applications. Examples are: computer games [19] such as computer chess[12], Google's famous search engine[13], and robot path planning. Further applications are discussed below.

Principal Search Algorithms

Although some other search algorithms are mentioned, the main algorithms that I have investigated for use in the robot's navigation are A*, the D* family of algorithms, Generalized Fringe-Retrieving A*, and Moving Target D* Lite.

2.4.1 A*

The A* search algorithm, which is an extension of Dijkstra's algorithm [19], is one of the most widely used search algorithms in the field of Artificial Intelligence [32], and forms the basis for the other search algorithms discussed in this section. It was first introduced by Hart, Nilsson and Raphael in 1968 [19].

A* performs faster than Dijkstra's algorithm by using heuristics [19]. It is a best-first search algorithm and is used to calculate a minimal path from a start state to a goal state [19]. Heuristics are methods used to estimate the shortest path from any location in its map representation to the goal state and therefore improve performance by sacrificing optimality [17][21]. This means that A* does not guarantee that it will find the optimal

[12]http://verhelst.home.xs4all.nl/chess/search.html
[13]http://www.techi.com/2012/03/googles-search-algorithm-changes-1998-2012/

path from source to goal since it uses a heuristic. However, if a path exists, it will be found. This is often the optimal path, but when it is not, it is often close to being optimal. Heuristics are used in scenarios where determining a path that is close to the shortest path is suitable and where better performance is a requirement. [17]

Algorithm Description

A simple, but concise overview of the A* algorithm according to [15] is as follows. Two lists, referred to as the "open" and "closed" lists, are maintained. The open list keeps track of the cells in the occupancy grid that are being considered at the current time and therefore, still need to be examined. Cells that are considered are those directly surrounding (a choice can be made regarding whether to include the diagonal cells) the current cell, traversable (do not contain obstacles) and are not in the closed list. The closed list contains cells that have already been examined. Starting from the source, the potential cells are those directly surrounding the source cell that are traversable, and so these are added to the open list. These new cells under examination point back to the source cell and therefore the source cell is their parent. The F value, which is the sum of the cell's G and H values, of each of the cells in the open list is then calculated. The G value is the cost of moving from the source to the cell along the path calculated while the H value is the estimated cost from the cell to the goal and is based on the chosen heuristic. The cell with the lowest F value in the open list is then chosen, removed from the open list and added to the closed list. This process is then repeated from the selected cell. When surrounding cells are being considered, if they are already in the open list, they are either updated with a new parent cell as well as a new F and G score if the new path from the source to the cell is shorter, or they are ignored if the new path is longer or the same distance. This process is repeated until the target cell is added to the closed list (a path has been found) or the open list is empty and the target has not been found (no path to from the source to the goal exists). The path is then determined by following the path of parent pointers from the goal back to the source. [15]

Heuristics

There are different heuristics that can be used with A*, some of which are discussed below [21]. A heuristic must be chosen according to the needs of the programmer and the hardware available. If a heuristic, h(n), is such that h(n) = 0, then the heuristic is not used in the calculation of f(n), and A* becomes Dijkstra's algorithm. When this heuristic is used, the shortest path is guaranteed, but the worst performance will be achieved. As the value of h(n) increases, the accuracy of the algorithm decreases and the performance

of the algorithm increases and so a trade-off between the two must be decided on. Three well-known heuristics for use with grid maps are the Manhatten, Diagonal and Euclidean distance heuristics. [21]

The Manhattan distance heuristic is the standard heuristic for use on square grids and if movement is restricted to four directions, it is preferable [21]. This heuristic can be calculated using the formula in *Equation 2.1.*

$$h(n) = d \times (|n.x - goal.x| + |n.y - goal.y|) \tag{2.1}$$

In *Equation 2.1*, n represents the grid cell that the heuristic to the goal is being calculated from while the constant d is used to scale the heuristic and is usually set to equal the lowest cost to move between adjacent grid squares. On the contrary, the Diagonal distance heuristic, also known as the Chebyshev distance, is useful when movement includes the diagonal directions, thus allowing movement in all eight directions. Its formula is shown in *Equation 2.2*.

$$h(n) = d \times max(|n.x - goal.x|, |n.y - goal.y|) \tag{2.2}$$

The Euclidean distance heuristic is mainly used when movement is not restricted to the usual grid directions, but is allowed in any direction. This heuristic can be calculated using *Equation 2.3*.

$$h(n) = d \times \sqrt{(n.x - goal.x)^2 + (n.y - goal.y)^2} \tag{2.3}$$

Although this heuristic is still usable on grid maps, and although optimal, or close to optimal paths will be calculated, performance is poor. [21]

A* calculates paths most accurately when all the map information is known. When not all the map information is known, D* is recommended[14].

Moving Target Search Problem

According to Sun et al. [32], the moving target search problem can be solved using A* in a dynamic environment to calculate the path with the lowest cost between the current start and goal states whenever a change in the environment or a deviation of the goal

[14]http://theory.stanford.edu/ amitp/GameProgramming/Variations.html

state from the current path occurs. This is computationally expensive as the new path has to be planned from scratch each time a change occurs [32].

2.4.2 D* (Dynamic A*)

D* refers to any of the three incremental search algorithms: Original D*, Focused D* and D* Lite [19]. Stentz first described the original D* algorithm in 1994 [27] and further developed it in 1995, giving rise to the Focused D* algorithm [26]. D* (Dynamic A*) is a generalization of A* for partially unknown, completely unknown or dynamic environments [27][26]. D* Lite differs algorithmically from the other D* algorithms, but is used to solve the same path-planning problems [19][11]. D* Lite was introduced by Koenig and Likhachev in 2002 and is based on the authors' Lifelong Planning A* (LPA*) [11].

Algorithm Description

All three of the D* algorithms can be used to solve the path-planning problem in which a robot traverses from a start state to a goal state in an unknown environment [19]. Since this is possible in an unknown environment, the three algorithms are also successful in this task in partially unknown and known environments. Nosrati et al. [19] gives a brief overview of how these algorithms function.

Initially, assumptions regarding the unknown map information are made and the minimal path from start state to goal state is calculated. The robot then traverses the planned path until a discrepancy in the map information is found. This new map information (a previously unknown obstacle, for example) is added to the robot's map information. If this new information conflicts with the current path, a new minimal path from the robot's current position to the goal state is planned. This is repeated until either the goal state is reached, resulting in success, or no path can be calculated to the goal state, resulting in failure.

Map Strategies for Unknown Terrain

When calculating the lowest cost path from the start state to the goal state using partial map information, three map strategies can be used to estimate the cost values of these unknown cells [28]. These strategies are known as the optimistic, pessimistic and average values strategies. For the optimistic strategy, unknown sections of the map are assumed to be easily traversable areas. For the pessimistic strategy, these sections are assumed to

be the hardest areas to traverse, yet are still traversable. For the average value strategy, unknown cells are assumed to be similar to the known cells in their surrounding area and so an average of these cost values is calculated. [28]

Comparison of Original D*, Focused D* and D* Lite

If D* is implemented in an unknown environment, replanning occurs often. As discussed in the A* review, this replanning calculation can be computationally expensive. D* solves the computational issue that algorithms such as A* suffer from by using incremental graph theory techniques [28]. Stentz [28] states that, in environments with many states, D* can perform hundreds of times faster than brute force replanning algorithms such as repeated A* searches and, as stated in [19], provided that the goal state is static, the D* algorithms all outperform repeated A* searches. Stentz [28] further states that D* can guide a robot in real-time through unknown and dynamic environments. Compared to the original D*, Focused D* reduces computation by only updating states that are still relevant to the robot's traversal using a heuristic [19]. The Focused D* algorithm is algorithmically complicated and is therefore difficult to understand and implement [11][29].

Unlike Focused D*, D* Lite is simple to understand and therefore easily analysable and extendable [19][11]. Koenig and Likhachev [11] state that its efficiency is greater than or equal to that of Focused D*. This is confirmed by Nosrati et al. [19]. Although, researchers have extended D* Lite in a straightforward manner in order to solve moving target search problems in changing environments, it is slow [32].

History of Use

The D* algorithm is viewed as a landmark in the area of mobile robot navigation [29] and has been implemented extensively on mobile robots [11]. Two of these implementations include indoor Nomad robots and outdoor High Mobility Multipurpose Wheeled Vehicles (HMMWVs) [11]. It seems that current systems generally implement D* Lite rather than the older D* algorithms. Even researchers in Stentz's lab, from where the D* algorithm originated, now often use D* Lite instead of D* for their current projects [36]. On Koenig's research page[15], some applications of D* Lite are listed. These include the prototype system tested on the Mars rovers "Spirit" and "Opportunity", in which elements of D* Lite were used. Another was Carnegie Mellon University's winning DARPA Urban Challenge vehicle, where elements of D* Lite were used for maneuvers such as navigating through parking lots and complicated U-turns.

[15] http://idm-lab.org/research.html

2.4.3 Generalized Fringe-Retrieving A*

Generalized Fringe-Retrieving A* (G-FRA*) was introduced in 2010 by Xiaoxun Sun, William Yeoh and Sven Koenig [31]. G-FRA* is an incremental search algorithm, which is a generalization of Fringe-Retrieving A* (FRA*). It allows for moving target searches to be solved on arbitrary graphs instead of using two-dimensional gridworlds, which are not realistic when working with robotic systems. This algorithm was designed for moving target search problems in static known environments. The hunter traverses along the optimal path from its current state to the current state of the target and whenever the target deviates from the path, a new optimal path is calculated. This new path is calculated using A*, but instead of replanning from scratch, previous search information is used. This process is repeated until the hunter catches the target. [32][31]

According to Sun et al. [31], FRA* is the fastest algorithm for moving target search problems using two-dimensional grids and, according to experimentation, G-FRA* proved to be the fastest algorithm for moving target search problems using arbitrary graphs. It performed better than Generalized Adaptive A* (GAA*) which was the previous fastest algorithm for the task using arbitrary graphs, by up to one order of magnitude. These experiments were performed on known state lattices, used for UGV navigation [31]. Although GAA* was tested in a static known environment [31], it can also be used in dynamic environments [30].

2.4.4 Moving Target D* Lite

The search for literature on this algorithm yielded few results, perhaps owing to its recent publication. Moving Target D* Lite (MT-D* Lite) as the name suggests, is an algorithm used for solving moving target search problems [32]. Specifically, it is an extension of D* Lite that makes use of the principle behind G-FRA* for recalculation of optimal traversals from the hunter to the target in dynamic environments [32]. MT-D* Lite is a recent algorithm introduced by Sun, Yeoh and Koenig in 2010 [32].

Sun et al. [32] state that D* Lite is not suitable for moving target search problems as it executes slowly. This is because it shifts the map in order to maintain a stationary start state. Much of the information obtained from previous searches is not reusable owing to this shift and therefore D* Lite can perform slower than when A* searches from scratch [32]. Sun et al. [32] go on to state that many other incremental search algorithms

including D*, FRA* and G-FRA*, are not suitable for solving moving target search problems in dynamic environments. This is because these algorithms were designed for use in static environments or systems in which the start state is kept stationary [32]. These performance issues led to the development of the two incremental search algorithms, Basic MT-D* Lite and MT-D* Lite, which are both extensions of D* Lite, but do not have to transform the map and solve moving target search problems in dynamic environments [32]. Under experimentation, MT-D* Lite performed four to five times faster than Generalized Adaptive A* (GAA*) [30], the previous fastest incremental search algorithm for solving the moving target search problem in dynamic environments [32]. GAA* was also introduced by Sun, Yeoh and Koenig [30].

Basic MT-D* Lite and MT-D* Lite Performance

For moving target search problems in dynamic environments, D* Lite can be used by calculating the optimal path from the hunter to the target whenever a change in the environment is observed or the target deviates from the known path. Basic MT-D* Lite increases the performance of this algorithm by calling its *BasicDeletion()* method instead of problematically shifting the map [32]. MT-D* Lite is an optimization of Basic MT-D* Lite which calls an optimized method, *OptimizedDeletion()*, instead of *BasicDeletion()*, and therefore, also does not require shifting of the map [32]. Pseudocode can be found in [32] for both the Basic MT-D* Lite and the MT-D* Lite algorithms. MT-D* Lite achieves improved performance compared to Basic MT-D* Lite by using the principle behind G-FRA* [32]. Under test conditions, Basic MT-D* Lite and MT-D* Lite performed better than GAA* in both static and dynamic environments. This is due to the fact that Basic MT-D* Lite and MT-D* Lite reuse previous search information whereas GAA* searches from scratch [32]. MT-D* Lite also performed better than Basic MT-D* Lite in both static and dynamic environments [32]. These experiments were performed in both static and dynamic, known environments.

The search for literature on the performance of MT-D* Lite in partially unknown and completely unknown environments yielded no results.

2.5 Summary

The discussed literature indicates that systems with a similar purpose to this project do exist, but are implemented differently. Although some literature could not be found,

it seems plausible that, using the toolkits and algorithms discussed in this chapter, the proposed robot navigation system can be successfully implemented on mobile robots, meeting all the project requirements. By implementing this system using a different approach, this project provides an opportunity to add to the areas of robot navigation using fiducial markers and robotic path planning.

Chapter 3

System Design

3.1 Introduction

The aim of this project is to design and implement a system that allows for the navigation of the Wifibot robot in open and busy environments using a fiducial marker. In *Chapter 2*, the criteria for this system were outlined and various options available for the construction of this system were investigated. This chapter focuses on the design of the system and therefore, the design decisions made concerning these options and the reasoning for these decisions are discussed.

The first section of this chapter covers the detection and tracking of the marker, after which algorithms for avoiding obstacles are discussed. Details regarding the robot's movement in the different modes of the system are then explained. Finally, the overall picture detailing how the different parts of the system fit together, is presented.

3.2 Fiducial Marker Tracking

The two libraries that were considered for tracking fiducial markers were ARToolKit and ArUco. Both of these libraries seemed likely candidates as they are both open source, cross-platform and, even though little information was found on the use of ArUco, both libraries are well documented. Owing to ARToolKit's robustness, popularity and widespread use, it was initially chosen. Unfortunately technical issues were experienced

when attempting to set up ARToolKit and it was therefore abandoned. It seems that much tweaking is required in order for ARToolKit to function correctly with Ubuntu 10.04 LTS which is perhaps owing to ARToolKit being out-dated. Since ArUco seemed just as competent, offers many great features as discussed in *Chapter 2*, and is up-to-date, it was chosen instead. Version 1.2.4 of Aruco was used along with OpenCv 2.4.2.

As noted previously, ArUco ships along with various sample applications, making understanding the library significantly simpler. Since ArUco was chosen, creating the marker could be done using one of the sample applications. This application allows for the production of up to 1024 different fiducial markers and saves the marker as a jpg image file of specified resolution. The marker can then be printed out and immediately used for tracking. Three different markers created using ArUco are shown in *Figure 3.1.* A marker size of length 160 mm was initially used for testing the system and after the system had been completed, the marker length was varied and a size that proved accurate whilst not too visually distracting was chosen.

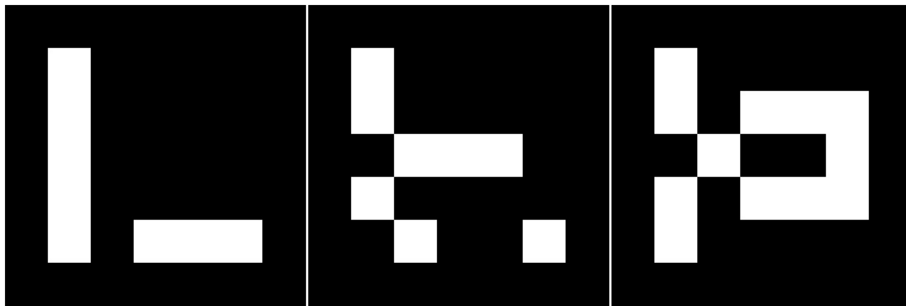


Figure 3.1: Markers created with ArUco

Most inexpensive web cameras with panning capability do not have the ability to pan in a full 360 degree circle. The web camera that the robot is equipped with is the Logitech QuickCam ® Sphere AF[1], and this camera, with its motorized tracking, can only view within 189 degrees of its surroundings. Note that this means that the camera's maximum pan capabilities are less than 189 degrees. Since the entire 360 degree view of the robot's surroundings cannot be observed, when the robot turns to avoid an obstacle, the marker needs to be kept within the camera's panning limits. Since ArUco, like ARToolKit, can only detect the marker if the entire marker is in the camera's feed, this means that the entire marker must be kept within 94.5 degrees either way from the robot's forward direction.

[1]http://www.logitech.com/en-in/webcam-communications/webcams/quickcam-sphere-af

Owing to the nature in which the search algorithm is used, which is discussed in *Section 3.3*, it is not therefore possible for paths calculated avoiding certain arrangements of obstacles to be traversed. To work around this problem, either a camera that can rotate a full 360 degrees is required or several cameras would need to be set up, thus capturing the robot's entire surroundings. If several cameras are used, they would need to overlap each other's feeds since, as mentioned, ArUco requires that the full marker be visible in the frame currently being processed. A problem that would then arise is that as soon as the marker is brought too close to the web cameras, the cameras would need to overlap further. Another possibility would be to stitch adjacent frames to form a single frame for analysis, but this would be overly complicated. If the robot's entire environment could be viewed, the angle that the robot could turn from the marker would then be limited by the LIDAR, which captures data in a 240 degree arc.

Owing to limited equipment and time constraints, these solutions were left for future work, and the single camera system was kept and certain environmental layouts ignored.

To access the web camera's controls, MJPG-Streamer[2] could have been used by issuing HTTP GET requests to the server, but this would mean that more than one process would be accessing the webcam since MJPG-Streamer would also stream video from the webcam. Only a single process can access the web cameras streaming functionality at a time without tweaking and so if OpenCv is to be used for the video feed, this would not be easily achievable. Instead, the web camera controls were accessed directly in the same way that they are accessed by guvcview[3], an interface for viewing and capturing video from Linux UVC compatible devices as well as changing camera settings and controls. This direct access is done using ioctl[4] system calls along with the V4L2 I/O[5] controls. Guvcview can also be used to disable automatic exposure and other automatic lighting enhancements of the camera as these hinder frame rates severely, and thus make the system seem clumsy. Since the system is intended for use in near perfect lighting conditions, disabling these settings should not affect accuracy.

[2]http://sourceforge.net/projects/mjpg-streamer/
[3]http://guvcview.sourceforge.net/
[4]http://pubs.opengroup.org/onlinepubs/009695399/functions/ioctl.html
[5]http://en.wikipedia.org/wiki/Video4Linux

3.3 Search Algorithms

As mentioned in *Chapter 2*, the search algorithm to be implemented needs to produce optimal paths, use minimal resources so that it is viable on low specification robots, and function in known, partially known and unknown environments. These requirements are satisfied by the replanning A* algorithm, the D* algorithms, the Generalized Adaptive A* (GAA*) algorithm and the Moving Target D* Lite (MT-D* Lite) algorithm. Since the system must further cater for a moving target, the GAA* and the Moving Target D* Lite algorithms are more suitable as they should perform better. According to the experimentation mentioned in *Chapter 2*, MT-D* Lite performs four to five times faster than GAA* and therefore MT-D* Lite was initially chosen for the path calculation.

Unfortunately, it was soon realized that the odometer readings from the robot's motors were inaccurate and not repeatable. Owing to the poor accuracy, using this algorithm efficiently was not possible because the odometer readings were to be used to indicate the position of the robot on the environment's map representation when adding data to the map acquired from the robot's LIDAR. It was not possible to localize the robot using odometry or other means such as vision-based localization, as this was out of the scope of the project. As a result, a map representation could not be constructed accurately. Inaccurate odometer readings also meant that it was not possible to move the robot forward by a certain distance, find out how far the robot had moved in a certain direction or find out how much the robot had turned in a certain direction, and therefore a path could not be calculated and then followed until a discrepancy in the map was found. This meant that even if a map representation could be built, none of the search algorithms mentioned thus far could be used in their usual manner to aid in traversing the robot from source to goal. The search algorithm could be used to calculate an optimal path, but that path could not be followed. If another robot with accurate odometer readings were used, the map representation could be constructed as discussed above and then MT-D* Lite could be implemented successfully. Thus, owing to this issue being common with many inexpensive robots, this project's aim was changed to implementing a system that would allow the Wifibot robot to follow a fiducial marker, avoiding obstacles optimally, whilst not relying on odometry at all.

Two algorithms were designed that could traverse past obstacles without having to know how far the robot had moved or turned, or that did not require the robot to move or turn by a certain amount. The first algorithm does not make use of a search algorithm,

but instead uses simple mathematics on the data captured from the LIDAR to make decisions about the direction to move in. This algorithm only considers obstacles that are close to the robot - within 1 m - and is therefore prone to making incorrect decisions. While the algorithm was successful in avoiding obstacles in simple scenarios, there were many restrictions on the layout of these obstacles and the algorithm was unsuccessful in more complicated environments. Such restrictions include that obstacles needed to be spaced far apart from each other, especially when the robot was not facing the opening between the two obstacles directly. Another issue was that, when only considering obstacles within 1 m of the robot, determining optimal paths from the robot to the goal correctly was highly unlikely. Owing to the algorithm's shortcomings, it is not discussed further.

A new algorithm was therefore needed, which led to the design of the second algorithm. This algorithm uses A* in a similar manner to when it is used in a dynamic environment as discussed in *Chapter 2*. The new algorithm differs, however, as the A* path is replanned every few hundred milliseconds instead of whenever a discrepancy in the map is found or the goal moves. Therefore a path is calculated only to determine in which direction the robot should move next. The robot is then issued the appropriate command and a new path is calculated. Since no information regarding the previous path calculated is reusable, using an incremental search algorithm would no longer result in improved performance. This is the reason that the A* search algorithm was chosen. This algorithm is discussed in detail in *Chapter 4*. As mentioned in *Chapter 2*, two-dimensional occupancy maps are the most successful environmental representations in the area of mobile robotics. For this reason, and owing to their simplicity, which makes for easy understanding and implementation, this representation was chosen over the more complicated topological map representation to represent the robot's environment in this project.

3.4 Robot Movement

3.4.1 Accessing Robot Functionality

Wifibot's motors are accessible via the RS232 port on the robot chassis. This is the port that the robot's IR sensors and odometer and speed readings are read from as well. In order to move the robot in a continuous manner, the specific move command needs to be given every x ms, with x having a value of at most 250 ms. Therefore, since an interval of

250 ms would give us reasonable accuracy with regard to movement and would be least straining on system performance, this interval was chosen. Accessing the LIDAR and the web camera on the robot is done in their respective manners as these devices have their own manufacturers and are not part of the robot. This is explained in a later section.

3.4.2 Follow Marker Design

ArUco is used to detect the marker in the current frame of the web camera video feed. The resolution of the video feed is set manually using OpenCv to 640x480. This resolution was chosen initially and was used until the system had been implemented. After implementation, the resolution was varied and tested according to the performance of ArUco's detect method and the marker detection accuracy. These results along with the final decision with regard to the best resolution are discussed in *Chapter 5*.

To keep track of the marker, the robot's web camera must pan, keeping the marker in its view. Therefore, after the marker has been detected, the center needs to be determined. If the center of the marker is not within two vertical thresholds, the camera must pan to keep the marker's center within these thresholds. These two thresholds should be set to 40% and 60% of the resolution width. Therefore, if a resolution of 640x480 is used, the thresholds will be vertical lines placed at x=256 and x=384. *Figure 3.2* shows the marker being detected using ArUco with the thresholds edited in.

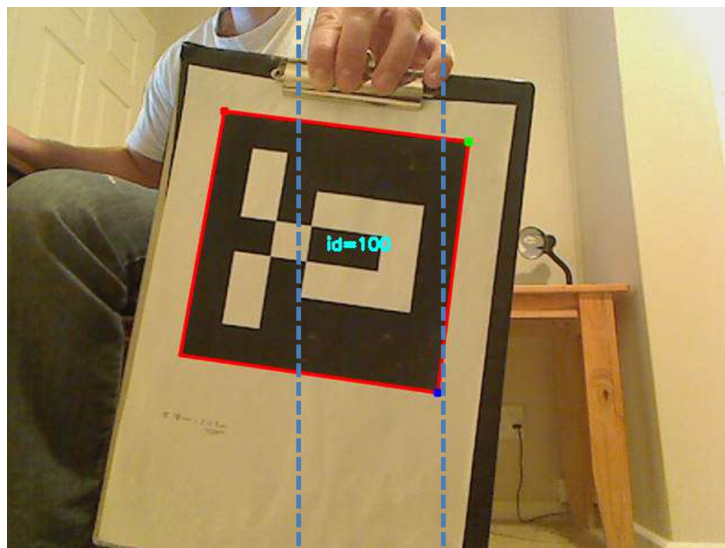


Figure 3.2: Detection with ArUco

Since the robot needs to follow the marker and not just the camera, once the camera has panned a certain amount the robot should turn towards the marker and while the robot is turning, the camera should remain centered on the marker. The robot should then keep turning until its front has lined up with the camera's front. Therefore, once the robot has finished turning, the robot will be facing in the direction of the marker, and so will the camera. Since the robot needs to turn once the camera has panned past a certain amount, the amount that the camera has panned must be stored and when this amount exceeds either of the two thresholds, the appropriate move commands can be issued to the robot motors. Since the maximum amount that the webcam can pan in either direction is 4000, which is roughly 65 degrees from the camera's default position, these two thresholds are set at -1000 and 1000, roughly -16.2 and 16.2 degrees from the camera's default position, respectively.

Having covered the turning of the robot, now we explain how the robot follows the marker successfully. Forward movement commands are issued to the robot when the marker is detected and the robot is not turning. This means that if the marker is moved to the left or right in the robot's view, the robot continues moving forward while the web camera pans. When the camera pans past one of the web camera thresholds, however, the robot stops moving forward and turns in the correct direction until it is facing the marker directly. The marker following process then continues. All this is done in a loop that iterates as fast as it can process frames, and therefore, a lower resolution may be crucial to ensure acceptable responsiveness of the system as a whole.

While following the marker, at some stage the robot could catch up to the marker. If this happens, a stop command is sent to the robot when the distance between it and the marker is less than a certain threshold, which was set to 750 mm as this distance seemed appropriate. The process for calculating the distance from the robot to the marker is explained below. First, a few calculations are done using the (x,y)-coordinates of the detected marker. The greatest length in pixels from one corner to another of the marker needs to be determined and in the case of the diagonal lengths, such as the length from the marker's top-left corner to bottom-right corner, the length along the x-axis (or y-axis, since these are the same) must be estimated by calculation for comparison. These lengths can be easily determined using trigonometry, since whenever a marker is detected, the x and y pixel coordinates for each of its corners are stored in the marker's *Marker* variable, which is mentioned later. When the marker is a certain distance from the robot, the value of its greatest length will be similar for all orientations of the marker, and thus this distance can be robustly determined. In order for the distance from the robot to the

marker to be calculated, a scaling constant is needed. To calculate this scaling constant, a few distances from the robot's web camera to the marker were measured and the longest length between the marker's corners at that distance was recorded. The scaling constant was then calculated by finding the average of the longest length in pixels multiplied by the measured distance from the robot's camera to the marker, which was found to be 37. Therefore, the distance between the robot and the marker, given the longest length in pixels, can be calculated using the formula presented in *Equation 3.1*.

$$distance(mm) = \frac{37}{longestlength} \times 1000 \quad (3.1)$$

Note that this is an extremely rough estimate for the actual distance from the robot to the marker and its accuracy can be affected by the height that the marker is held above the ground as the vertical dimension was ignored in the calculations for simplicity. In practice however, this does not cause any issues as very little accuracy is required for this distance. Originally, to estimate the distance between the robot's camera and the marker, the ArUco area or perimeter methods were used on the detected marker, but this was not as robust as the above method under certain orientations of the marker. This calculated distance to the marker is not only used to issue a stop command to the robot when the marker is too close, but is also used in placing the goal for the A* algorithm in the occupancy grid as explained in the next section.

3.4.3 Obstacle Avoidance

The URG-04LX-UG01 LIDAR[6] from Hokuyo is the LIDAR attached to the robot used in this research. It can be accessed by making use of the URG library that is provided by the manufacturer. There are two modes to choose from when reading from the LIDAR, GD scan and MD scan. The GD scan, which is the read once mode, was chosen for this system. The MD scan is the mode for continuously reading from the LIDAR, but, if delays occur between consecutive reads, errors can occur.

The data returned from the LIDAR is stored in an array containing the distances (in mm) to obstacles from the LIDAR. Since this LIDAR has a scan range of 240 degrees and the gap between each scan is 0.35 degrees, the array contains 685 valid distance values. Note that the LIDAR also has a dead zone as shown in *Figure 3.3*. Several measurements recorded in this dead zone are also present in the returned array, but are not valid, and

[6] http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx_ug01.html

therefore, must be ignored. As the scans occur from right to left in an anti-clockwise direction, the distances are indexed in the array with the rightmost distances starting at the beginning of the array and the leftmost distances contained towards the end of the array.

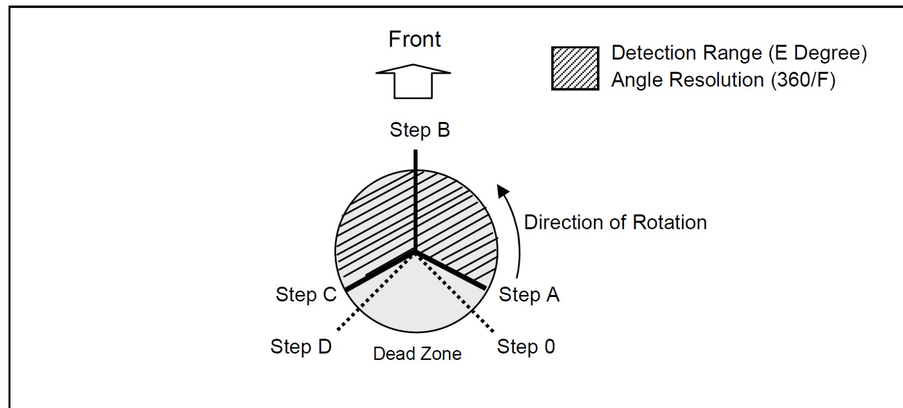


Figure 3.3: LIDAR capabilities

In the various scenarios that are mentioned later, a method is needed for determining whether obstacles exist in the robot's near path. Since the robot's width is 0.38 m, a rectangle of 0.44 m (adding 30 mm to each side for certainty) by 1 m directly ahead of the robot as shown in *Figure 3.4*, represents the robot's near path ahead. If any obstacles are detected within this rectangle from the LIDAR scan, the robot's path is said to be obstructed, else the path ahead is clear.

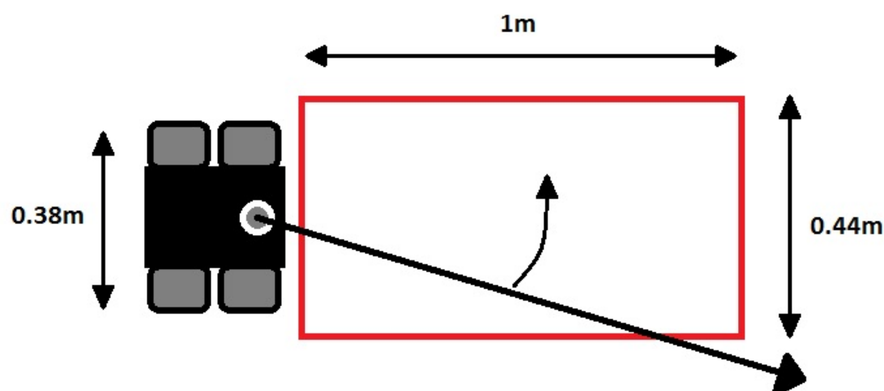


Figure 3.4: Rectangle used for checking path ahead

To construct the occupancy map, after the LIDAR has returned the array containing the distances from the robot to the nearby obstacles, the data in the array needs to be

converted into (x,y)-coordinates and then into occupancy grid coordinates. In *Chapter 4*, more detail regarding these conversions are presented. The obstacles are then placed in the occupancy grid. After a path has been calculated from the robot to the goal, avoiding the obstacles, and the robot has been issued a move command, the above process is repeated continuously to acquire updated map data, thereby updating the path to the goal. Note that for the first algorithm mentioned in *Section 3.3* that does not make use of a search algorithm, only the array returned by the LIDAR is needed. No conversions need to take place as no map is built.

Since the LIDAR can measure up to 5.6 m in its 240 degree scan range, a decision was made regarding how much of the surrounding environment needs to be considered when using A* to calculate the path. If the entire scanned area is represented, the occupancy map will be larger and therefore A* will perform more slowly. When less scanned area is represented, although performance is increased, the chance that incorrect or less optimal paths are chosen increases. Initially, the entire scan range was represented as the robot used in this system is reasonably powerful. However, after performance testing on the implemented prototype, a suitable scan range was found.

Another decision made regarding the occupancy grid, was the size of the area that each of the grid cells represent. The greater the area that each cell covers, the fewer cells needed in the grid, meaning better performance can be achieved. On the other hand, if each cell represents a smaller area, more cells are needed, resulting in a more accurate representation. This decision could only be made after performance testing, and so, initially each cell represented a ground area of 7744 mm^2, with the length of each side being 88 mm. For convenience hereafter, the distance represented by the sides of each cell is referred to as the cell size. The reason that a length of 88 mm was chosen is that it is a fifth of 440 mm, which is the robot's width (note that the width of the robot is greater than the length). This means that whenever an obstacle is inserted into the occupancy grid, additional obstacles are also inserted into all cells that are within two cells of the obstacle - meaning that 24 additional obstacles are inserted. This eliminates the possibility of calculating paths that are too narrow for the robot to traverse. Since a cell size of 88 mm is used, the scan area is limited to 5588 mm.

An extra two rows and columns are needed surrounding the grid for reasons explained below and therefore, the dimension of the occupancy grid is 131x131. The actual valid area, ignoring the area behind the robot, makes up a grid with dimensions 127x127. The two rows and the two columns on each side of the grid give an extra four rows and four

columns, thus giving the final dimensions of 131x131. Since the fiducial marker will often be held by a person, the data returned from the LIDAR includes the detection of the person's feet or lower legs. This data is then exaggerated by placing an extra 24 obstacles surrounding each cell in which the person is detected. The goal cell is therefore often surrounded by obstacles, eliminating the chance of a path to the goal being found. To prevent this from happening, obstacles (actual or exaggerated) surrounding the goal cell (all cells within four cells of the goal cell) are not placed in the occupancy grid. A similar issue is experienced surrounding the robot's cell, where obstacles near to the robot exaggerate and cause the cell to become completely surrounded. This too, eliminates the chance of a path being calculated. To solve this issue, obstacles close to the robot cannot be completely ignored, as this would result in the robot making contact with the obstacles. Instead a simple algorithm, which is discussed in *Chapter 4*, was designed, to solve this issue.

Since the LIDAR scans in an arc shape and the occupancy grid is a square, a ring of obstacles, two obstacles deep, is placed surrounding the robot's scan area. This is the reasoning for the addition of the extra two rows and columns mentioned above. Two lines of obstacles, once again two obstacles deep, are also placed from directly behind the robot to the ring of obstacles along the 120 degree and -120 degree scan line from the robot's LIDAR. The ring and the two lines are placed such that paths that are out of the scan area specified by the user or in the dead zone of the LIDAR cannot be calculated. The reason that both the ring and the two lines are two obstacles deep is that if all eight surrounding cells on the occupancy grid are considered for movement, and when a single line of obstacles is placed diagonally, paths can be calculated that get through into the area that the ring and lines are trying to block off. *Figure 3.5* illustrates the occupancy grid with the ring and two lines of obstacles placed in it.

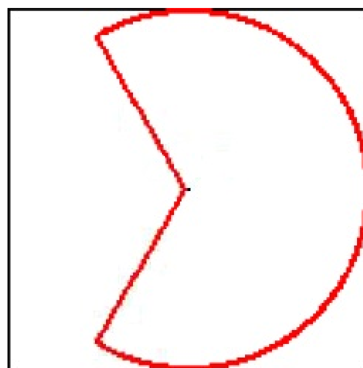


Figure 3.5: Occupancy grid showing blocking obstacles

On the occupancy grid, the robot is placed in the cell at row and column 65 - the center of the grid. If the LIDAR scan range is limited or a different cell size is used, this will affect the dimension of the grid and therefore it is likely that the center of the grid will have a different row and column number. In order to place the goal on the grid, the angle between and the distance from the robot to the marker need to be calculated or estimated.

Estimating the distance has already been discussed in *Section 3.4.2*. Estimating the angle is done by dividing the stored pan position, discussed earlier, by 61.5. The reason for using 61.5 is that when the camera pans by a value of 100, the measured rotation of the camera is 1.626 degrees and 100/61.5=1.626. The goal cell is then determined using these two values; a detailed process of how this is done is given in *Chapter 4*. It should be noted that these two values are estimated and only reasonable accuracy is required since there is no need to pinpoint the exact location of the marker on the map because we are working with a 88 mm resolution and, even if the goal cell is off by one or two cells, the system should not function any differently.

Another decision that needed to be made concerned the number of directions in which the robot can possibly move on the occupancy map, or rather, which of the eight surrounding cells are eligible. For this project the choice was between four (no diagonal movement) and eight (includes diagonal movement) directions. If the robot can only move in four directions, fewer cells are considered for movement and thus the A* algorithm performs faster, but the paths calculated will not be as optimal as if eight possible directions are used. When eight directions are considered the performance is reduced. After implementation, the difference in performance was reviewed and a final decision made. Initially, all eight possible directions were considered for movement. The Diagonal (Chebyshev) heuristic was selected for use with the A* algorithm for reasons discussed in *Chapter 2*. If the final decision were changed to consider only the four adjacent directions, the Manhattan heuristic would be used along with the A* algorithm.

Every time the LIDAR scans the surrounding area, a map is constructed and a new path from the robot to the marker is calculated. Therefore, two OpenCv *Mat* images showing the path avoiding the obstacles are created and displayed each time the LIDAR scans. The first of these images contains the robot's surrounding environment as it is (without any extra obstacles added or ignored), while the second shows the exaggerated map (with the 24 extra obstacles added and the obstacles surrounding the goal cell removed) so that the user can obtain a clear picture of what is happening.

3.4.4 Searching

Owing to the circumstances concerning the robot's poor odometer accuracy, it is not possible to implement an elegant manner in which the robot can wander. Under different circumstances, when the marker is lost, the most recent location of the marker could be kept as the goal and then the robot could traverse towards it in an attempt to find it again. Once that location has been reached, a wandering process could begin. An example of a wandering algorithm that could be used if the robot's odometry were accurate would be to randomly choose a cell in the occupancy map to set as the goal and then traverse towards the goal while panning the webcam searching for the marker. Cells could also no longer be traversable (by setting the cell in the occupancy map appropriately) once they have been traversed, provided that placing an obstacle in that cell would not close off any area that had not yet been searched. This is done to avoid searching in areas that have already been searched in.

Since this approach was not possible, when the marker is lost, the robot and its camera remain stationary for 5 s, thus allowing sufficient time for the user to display the marker in the camera's view. If the marker has not been detected by the time that this period has elapsed, the marker is searched for by panning the web camera all the way in one direction and then all the way in the other direction. The direction in which the web camera initially pans is the direction in which the marker was last detected. If the marker has still not been detected, the robot performs a turn of roughly 360 degrees (roughly, since the odometry cannot be measured). The direction of this turn is towards the direction in which the marker was last detected. Thus, after the 5 s time period has elapsed, the searching mode includes three stages - panning the camera in the most likely direction, panning the camera in the other direction and, finally, the robot performing a full 360 degree turn. If the marker has not been detected after these three stages have been performed, another 5 s delay occurs, followed by a repeat of the search process.

3.5 Putting it all together

As discussed in *Sections 3.4.2 - 3.4.4*, the state of the robot at any time is either: following the marker normally when no obstacles are in its path, avoiding obstacles in an attempt to follow the marker normally, or searching in an attempt to find a lost marker. This means that the robot can be in one of three different modes at any time. After all the

devices have been initialized and all the startup processes performed, the system enters into the main loop. When entering the loop, mode 3 - the searching mode - is the default and thus this is the robot's initial mode. The searching process as described in *Section 3.4.4* is carried out until the marker is detected. On detection of the marker, the robot switches to mode 1 - following the marker normally - by default, and begins following the marker. The robot remains in mode 1 provided that no obstacles are detected in the robot's path, meaning that no obstacles are detected within the rectangle described in *Section 3.4.3*. If an obstacle is detected inside this rectangle, the system switches into mode 2, obstacle avoidance mode. For a transformation from mode 2 back to mode 1, the robot needs to have successfully avoided any obstacles in its path to the marker. Then, the system changes back into mode 1 if either the marker is detected directly ahead of the robot and the path ahead of the robot is clear, or the marker is detected and its distance from the robot is 750 mm or less. If the marker is lost in either mode 1 or mode 2, the robot switches to mode 3. Whenever a switch to mode 3 occurs, the mode that is being switched from is recorded so that when the marker is detected again, the previous mode can be reinstated. The switching between modes is represented visually using a state diagram in *Figure 3.6*.

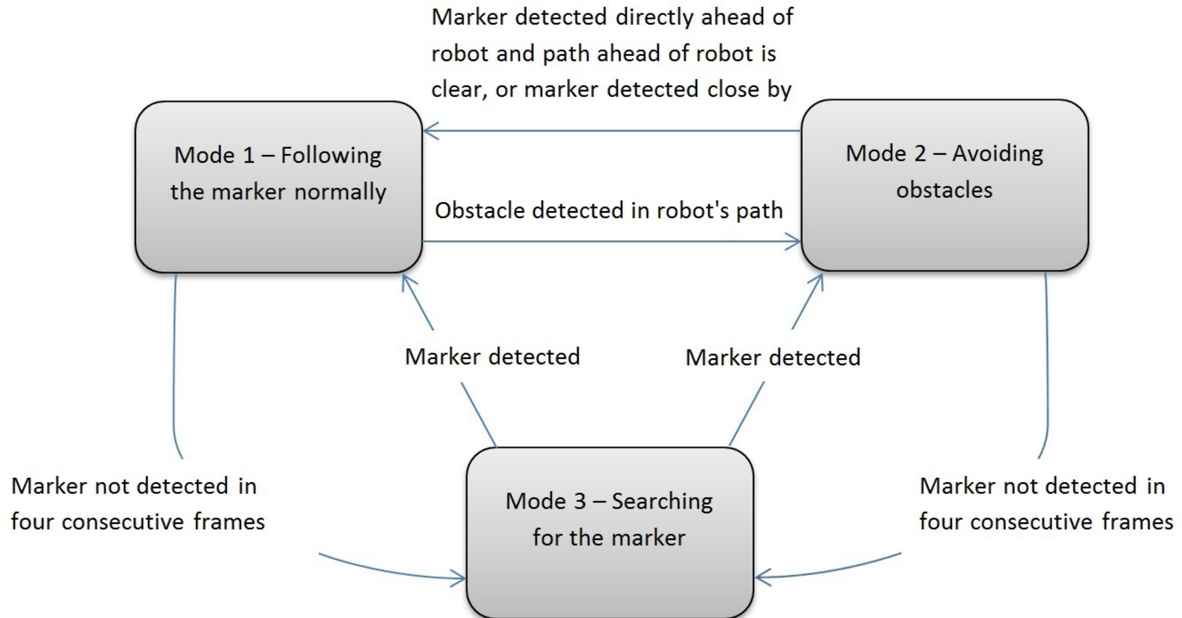


Figure 3.6: State diagram of mode switching

In order for this system to be implemented successfully, three loops are required, two of which are run in separate threads that are spawned at startup. The first of these is the main loop for the system. This loop receives frames from the camera, detects the marker

in the frames, pans the web camera, sets a flag to indicate that the robot's motors should be issued specific commands provided the system is in mode 1 or mode 3, and displays the video feed showing the detected marker. Note that in this loop, a flag indicating that the robot should move in a certain direction is set. This flag is checked in the second loop, which is where the actual commands to the robot's motors are issued. If the flag is set to move forward and is not changed for the next few seconds, the move forward command is continuously issued to the motors and, for this reason, this process needs to be implemented using a simultaneously executing loop. This loop runs in its own thread, the *robotmove* thread.

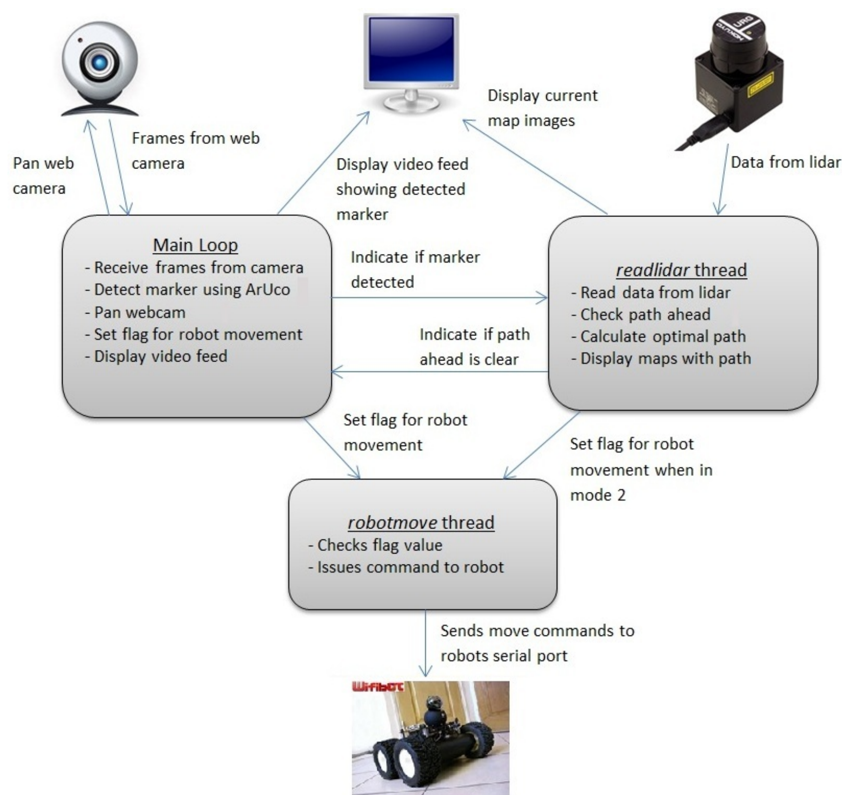


Figure 3.7: Information flow between looping threads

The third loop is used to read data from the LIDAR and indicate to the main loop whether the path ahead of the robot is clear. When the system is in mode 2, this loop also handles the obstacle avoidance process and the display of the optimal path, calculated to avoid the obstacles. When calculating the optimal path avoiding the obstacles in mode 2, this third loop, like the main loop, sets a flag instructing the robot's next move. The search algorithm is therefore called from this loop. Since data from the LIDAR must be read continuously, the loop has been implemented to run in its own thread, which is called the

readlidar thread.

These three loops, along with some of the system's main functionality, are shown in *Figure 3.7*. When the system starts up, all the devices are initialized and various startup processes are performed. After this, the two threads, *robotmove* and *readlidar*, are created and execution of these two loops begins. After these threads have been spawned, the third loop, where the marker detection takes place, is started in the *main()* method itself.

Since this system requires quite a few processes running in parallel, implementation needs to be carried out using good coding practice and must be as efficient and optimal as possible. Even though the robot used in this system has high specifications, as discussed in *Chapter 2*, the system was designed in such a way as to require as few resources as possible since many other mobile robots have much lower harware specifications.

3.6 Summary

In this chapter, design decisions regarding the options investigated in *Chapter 2* were made. Some of these decisions proved impractical owing to hardware issues experienced with the Wifibot robot, and therefore, some of these options were reconsidered. ArUco was finally chosen for the detection of fiducial markers and the A* search algorithm, used in a repetitive manner, for obstacle avoidance. Owing to their popularity and simplicity, occupancy grids were chosen to represent the robot's environment. The manner in which the robot would function as well as how the system would operate were described in detail.

Chapter 4

System Implementation

4.1 Introduction

Whereas *Chapter 3* covered the design of the system, this chapter deals with the implementation of the system. As mentioned previously, implementation was done in C++ on the robot itself.

The first section of this chapter shows how the robot's different devices were accessed. Marker detection using ArUco and the logic behind the robot's navigation when following the marker normally are then discussed. Next, the process of constructing the occupancy grid, calculating an optimal path to the marker and determining the direction in which the robot should move, is detailed. Finally, implementation-specific details concerning the searching process are presented.

4.2 Accessing Robot Functionality

The Robot's Motors

Issuing commands as well as accessing information such as the odometer readings and the current speed of Wifibot's motors were done through the robot's RS232 serial port as mentioned in *Chapter 3*. If required, other information such as information from the IR sensors could also be obtained through this port. When issuing a command, data is sent to the serial port, which then sends the command to the DSPIC33F, the microcontroller

that controls the robot's motors. Reading is done in a similar manner. Before sending or receiving data from the serial port, the port needs to be opened and initialised at a baud rate of 19200. Once the port has been opened and initialised, data can be sent to or read from the robot, in the form of a buffer. For setting the robot's motors to a certain speed in a certain direction, the RS232 serial port expects nine 8-bit characters to be sent in a buffer in a particular format. The important indices of this buffer are indices three to seven. The third and fourth characters make up the desired speed of the robot's left motor while the fifth and sixth make up the speed of the right motor. According to the "Protocol Sheet" document available for download from the manufacturer's website[1], the seventh 8-bit character is decomposed as follows:

Bit 7 (128) - Left side closed loop speed control :: 1 -> ON / 0 -> OFF
Bit 6 (64) - Left side forward/backward speed flag :: 1 -> Forward / 0 -> Reverse
Bit 5 (32) - Right side closed loop speed control :: 1 -> ON / 0 -> OFF
Bit 4 (16) - Right side forward/backward speed flag :: 1 -> Forward / 0 -> Reverse
Bit 3 (8) - PID speed :: 1 -> 10 ms / 0 -> 50 ms (50 ms is recommended)
Bit 2 (4) - Not used, set to 0 (future option)
Bit 1 (2) - Not used, set to 0 (future option)
Bit 0 (1) - Not used, set to 0

For the robot to move smoothly, move commands must be issued periodically, at least every 250 ms. If data is not sent after 250 ms, a timeout occurs and the robot's speed is set to zero, therefore causing movement to become jerky. To send a move command to the robot the three methods discussed in *Appendix A.1* are used.

To move the robot forward using both motors at a speed of 130 tics, the *setRS232Motor33f()* method is called in the manner shown in *Listing 4.1*.

```
1 while (true) {
2     setRS232Motor33f(hUSB, 130, 130, 80);
3     usleep(250000); //sleep in microseconds
4 }
```

Listing 4.1: Move robot forward continuously

According to *Appendix A.1* and *Listing 4.1*, to send a command, a buffer needs to be filled with the appropriate hexadecimal values, and then sent one byte at a time to the

[1] http://www.wifibot.com/page5.php

RS232 port. Note that the characters in the buffer's second to seventh indices are used to calculate the cyclic redundancy check (CRC) value, which is then stored in the buffer's eighth and ninth indices. To give the robot a turn command at the same speed as in the example in *Listing 4.1*, the seventh character in the buffer must be changed from 80 (64+16) to 16 for a left turn or 64 for a right turn. These values can be easily calculated using the information on the seventh character, decomposed as mentioned earlier. As mentioned in *Chapter 3*, robot commands are issued from a seperate thread owing to the command having to be resent every 250 ms. The main section of this thread, the *robotmove* thread, is shown in *Listing 4.2*.

```
1  void* robotmove(void*) {
2
3      //allows for termination from main()
4      pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
5
6      //move robot
7      //speeds: min=70, max=240
8      while(1) {
9          if (move_cmd == 0) { //move forward
10             setRS232Motor33f(hUSB, 130, 130, 80);
11             usleep(250000);
12         }
13         else if (move_cmd == 1) { //turn left
14             setRS232Motor33f(hUSB, 160, 160, 16);
15             usleep(250000);
16         }
17         else if (move_cmd == 2) { //turn right
18             setRS232Motor33f(hUSB, 160, 160, 64);
19             usleep(250000);
20         }
21         else
22             usleep(250000);
23
24         //...
25     }
26 }
```

Listing 4.2: The *robotmove* thread

Here, it can be seen how the flag, *move_cmd*, is set to instruct the robot to move in a particular manner. Since the thread loops until the user ends the program, the thread's cancel type has been set, as seen in line 4, so that it can be terminated easily from the

main loop when the user requests the program to shut down.

To read data from the robot, the process is reversed. The DSPIC33F sends 21 characters every 10 ms to the RS232 port. This data is accessed by first reading from the serial port and then extracting the data from the buffer. Reading from the robot's chassis was successful but, as mentioned before, the accuracy of the odometer readings was poor, and since this was the only reason for data to be read, this functionality was not required.

Web Camera Control

Two methods for panning and tilting the web camera were implemented successfully. The first method involved using MJPG-Streamer, as described in *Chapter 3*. The GET requests were given to the server by making use of the cURL[2] library. Since the camera's feed could not be accessed by OpenCv at the same time as the MJPG-Streamer server was running, another method for accessing the camera controls was needed. Panning and tilting were eventually implemented using ioctl system calls along with the V4L2 I/O controls as discussed in *Chapter 3*. *Listing 4.3* shows the function called to pan or tilt the web camera and in line 31 an example is given of how to pan the camera to the left by a small amount.

```
1  void pantilt(int dev, int pan, int tilt, int reset) {
2      struct v4l2_ext_control xctrls[2];
3      struct v4l2_ext_controls ctrls;
4  
5      if (reset > 0) {
6          switch (reset) {
7              case 1:
8                  xctrls[0].id = V4L2_CID_PAN_RESET;
9                  xctrls[0].value = 1;
10                 break;
11             case 2:
12                 xctrls[0].id = V4L2_CID_TILT_RESET;
13                 xctrls[0].value = 1;
14                 break;
15         }
16         ctrls.count = 1;
17         ctrls.controls = xctrls;
18     }
19     else {
```

[2] http://curl.haxx.se/

```
20          xctrls[0].id = V4L2_CID_PAN_RELATIVE;
21          xctrls[0].value = pan;
22          xctrls[1].id = V4L2_CID_TILT_RELATIVE;
23          xctrls[1].value = tilt;
24          ctrls.count = 2;
25          ctrls.controls = xctrls;
26      }
27      if ( ioctl(dev, VIDIOC_S_EXT_CTRLS, &ctrls) < 0 )
28          perror("VIDIOC_S_EXT_CTRLS - Pan/Tilt error. Controls are not
                available\n");
29  }
30
31  pantilt(dev,100,0,0); //pans webcam to the left
```

Listing 4.3: Pan/tilt controls

Tilting or resetting the camera can be done by setting the third and fourth parameters of the function appropriately. The only time that the camera is tilted in this system is immediately after it is reset. When the system starts up, the camera is reset to its default position and therefore tilted up slightly to account for the height of a person holding the marker. The camera is also reset after it pans, searching for the marker in mode 3.

Reading Data from the LIDAR

In *Chapter 3*, two modes for reading data from the LIDAR - the GD scan and the MD - were mentioned. Since the GD scan was found to be more suitable, it was chosen for implementation as explained below. After the LIDAR device's setup and variable reservation processes have been performed, a request for the GD data is sent to the LIDAR device. After a successful response, the data is retrieved into the data array mentioned in *Chapter 3*. *Listing 4.4* shows the process of requesting, retrieving and then accessing the data from the LIDAR.

```
1 int min_length = urg_minDistance(&urg);
2 int max_length = urg_maxDistance(&urg);
3
4 //request for LIDAR GD data - single read mode
5 int ret = urg_requestData(&urg, URG_GD, URG_FIRST, URG_LAST);
6 if (ret < 0)
7     urg_exit(&urg, "urg_requestData()");
8
9 //reception of data
```

```
10 int max = urg_receiveData(&urg, data, data_max);
11 if (max < 0)
12    urg_exit(&urg, "urg_receiveData()");
13
14 for (i = 0; i < n; i++) {
15    if ((data[i] <= min_length) || (data[i] >= max_length))
16       continue; //ignore out of range values
17    printf("%d: %ld\n", i, data[i]);
18 }
```

Listing 4.4: Request and retrieval of LIDAR data

4.3 Detection and Following

By making use of OpenCv, streaming from the web camera and processing each frame received from it is simple and neat. Once a *VideoCapturer* object has been instantiated and opened, frames can be read from it in a loop and processed in the same iteration before the next frame is obtained. Decisions on what actions the robot should perform in mode 1 and mode 3 are all made in this loop. ArUco allows the detection of markers in an image using a single line of code as shown in *Listing 4.5*. This listing shows an example of using OpenCv and ArUco to: stream images from a web camera, detect a marker in the image, draw a box around the marker, calculate the x-coordinate of the marker's center and output the marker's details.

```
1 int key = 0;
2 int centerx;
3 float marker_size = 0.159;
4 MarkerDetector marker_detector;
5 CameraParameters cam_params;
6 VideoCapture video_capturer;
7 vector<Marker> markers;
8 Mat input_img, input_img_copy;
9
10 //capture until ESC
11 while (key!=27 && video_capturer.grab()) {
12    centerx = -1; //also to check if marker detected
13    video_capturer.retrieve(input_img);
14    marker_detector.detect(input_img, markers, cam_params, marker_size);
15    input_img.copyTo(input_img_copy);
```

```
16      for (unsigned int i = 0; i < markers.size(); i++) {
17          markers[i].draw(input_img_copy, Scalar(0,0,255), 2); //outline marker
18          centerx = markers[i].getCenter().x; //get x coord of center of marker
19      }
20      imshow("Live feed", input_img_copy); //show output
21      key=waitKey(2);
22  }
```

Listing 4.5: Dectection using ArUco

As seen in this listing, when ArUco's detect method is called, the *markers* vector contains information on as many markers, stored as *Marker* variables, as are detected in the current frame. For this system, since only one marker was needed, the vector only contains up to one marker at a time.

To make decisions regarding whether the robot should pan its camera in a certain direction, or turn itself in a certain direction *etc.*, conditional statements containing the criteria for such decisions discussed in *Chapter 3* were implemented. Two examples showing these kinds of decisions are given in *Listing 4.6*. Note that these are just examples and although they do exist in the system as shown, the scope in which they are used is different.

```
1  //example 1
2  if ((centerx != -1) && (centerx < 128) && (pan_pos <= 4000)) {
3      pantilt(dev, 100, 0, 0); //pans webcam to the left
4      pan_pos += 100;
5      cam_pan = true;
6      //...
7  }
8  else if ((centerx != -1) && (centerx > 192) && (pan_pos >= -4000)) {
9      pantilt(dev, -100, 0, 0); //pans webcam to the right
10     pan_pos -= 100;
11     cam_pan = true;
12     //...
13 }
14
15 //example 2
16 if ((centerx != -1) && (pan_pos > 1000)) {
17     move_cmd = 1; //robot turns left
18     turn_marker = 1; //used to indicate that robot is turning left towards
           the marker
19     not_detected = 0;
20     //...
```

```
21 }
22 else if ((centerx != −1) && (pan_pos < −1000)) {
23     move_cmd = 2; //robot turns right
24     turn_marker = 2; //used to indicate that robot is turning right towards
           the marker
25     not_detected = 0;
26     //...
27 }
```

Listing 4.6: Implementation of various decisions

The actual panning of the camera is done from within these if statements whilst the movement of the robot is done by setting the global variable *move_cmd* appropriately. This variable is then checked in the loop in the *robotmove* thread and according to its value the appropriate move command is sent to the robot's motors.

An issue was experienced during implementation where, owing to the marker being moved quickly around in the camera's view, individual frames became blurred, causing ArUco not to be able to detect the marker. This in turn would caused the robot to change to mode 3, and subsequently stop. The marker would again be detected soon after the motion had stabilized causing the robot to switch back to mode 1 and continue following. This caused the robot to move in a jerky manner. To solve the problem, the variable, *not_detected* - used in *Listing 4.6*, was introduced in an attempt to smooth the robot's movement. The variable is a simple counter that counts from zero to four and is incremented by one whenever the marker is not detected. When it reaches four, the marker is considered lost and the system changes to mode 3, but whenever the marker is detected, *not_detected* is reset to zero. This allows for much smoother movement when following the marker. Note that this functionality is crucial to the smoothness of the robot's movement, especially when the frame rate of the camera is low.

4.4 Obstacle Avoidance

Path Ahead Rectangle Implementation

In order for the system to switch from mode 1 to mode 2, at least one obstacle needs to be detected in the path ahead of the robot. This path ahead, as mentioned in *Chapter 3*, is defined to be the rectangular area ahead of the robot, which is represented using an array

with the indices representing the scan number of the LIDAR and the distance from the LIDAR to the edge of the rectangle being stored in the appropriate indices. *Figure 4.1* shows a more detailed representation of this rectangle.

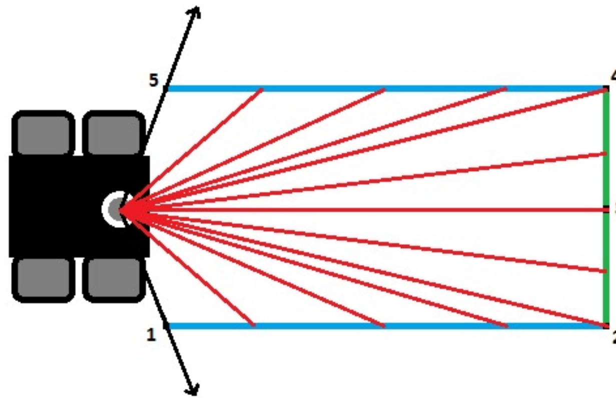


Figure 4.1: Detailed representation of rectangle

It can be seen in this figure that the "rectangle" only has three straight sides. This is indeed the case as the side from corner 1 to corner 5 is actually triangular because the scan range of the LIDAR is limited to create this fourth "edge". Nevertheless, the shape is still referred to as a rectangle.

Since the size of the arc that the LIDAR needs to check to see if obstacles are in its direct path is roughly 119 degrees, and since the LIDAR scans with a 0.35 degree resolution, the array has 340 elements. The distances from the location of the LIDAR along their respective LIDAR scan lines of the first 137 elements as well as the last 137 elements of the array are represented by the two blue lines from corners 1 to 2 and corners 5 to 4, respectively, as shown in *Figure 4.1*. The values for the first blue line are the same as those for the second and are calculated using formula A given in *Listing 4.7*. The distances for elements 137 to 169 and 170 to 202 are represented by the two green lines from corners 2 to 3 and corners 4 to 3, respectively. The values for these lines are also equal and are calculated using formula B. The red lines in the figure represent the distances along their respective scan lines. Therefore, if the LIDAR scans an obstacle on a certain scan line and the distance to the obstacle is less than the distance specified in the array at that scan line's index, the obstacle is said to be in the robot's direct path. *Listing 4.7* shows how the values for this array are calculated.

```
1 int lengths[340]; //contains distances in 1m by 0.44m box for LIDAR
```

```
2 float angle = 30;
3 int temp;
4
5 for (int i = 0; i < 170; i++) {
6     if (i < 137){
7         temp = (220/(cos((angle*3.14)/180)))+0.5; //formula A
8         lengths[i] = temp;
9         lengths[339-i] = temp;
10    }
11    else if (i < 170){
12        temp = (1000/(cos(((90-angle)*3.14)/180)))+0.5; //formula B
13        lengths[i] = temp;
14        lengths[202-(i-137)] = temp;
15    }
16    angle += 0.35;
17 }
```

Listing 4.7: Rectangle array value calculation

Note that the corresponding values of the array are offset to those of the data array returned by the LIDAR by a value of 215. This is because the size of the arc needed to check the robot path is 120 degrees whereas the arc that the LIDAR scans is 240 degrees. In both formulas presented, a value of 0.5 is added to the final distance - the reason being that the distances are stored as integers and therefore, are automatically rounded down. By adding a value of 0.5, the lengths are rounded off to the closest whole number.

Obstacle Avoidance Algorithm

As mentioned in *Chapter 3*, an algorithm was designed to calculate optimal routes from the robot to the marker avoiding all obstacles in its path by making use of the A* search algorithm. The pseudo code for this algorithm is as follows:

```
    while true:
1.      if marker detected directly in front of robot AND path ahead of
        robot is clear:
            break out of loop and follow marker normally
        else if marker lost:
            break out of loop and search for marker
2.      lidar scans for obstacles up to 5.6 m away in 240 degree arc
3.      obstacle data received from lidar is converted to occupancy map
```

```
        coordinates through conversion process described later
4.      obstacles are placed in the occupancy map using these coordinates
5.      fixed position of robot is placed in occupancy map
6.      goal coordinates are calculated using distance from the robot to
        marker and angle between direction that robot faces and marker
7.      goal placed in occupancy map using goal coordinates
8.      A* search algorithm used to calculate optimal path from robot to
        marker
9.      path calculated is analysed to determine robot action
10.     robot given the appropriate command
```

According to this pseudo code, after A* is used to calculate the path, the path is analysed to determine which move command the robot should be given. This is done by looking at the first cell of the occupancy map that the calculated path occupies - the cell that the robot needs to move to first. The robot is then issued a turn left, turn right, or move forward command based on this cell before the process is repeated.

This algorithm, although more complex and resource intensive, solves all the issues that were experienced with the first algorithm and can calculate paths past obstacles in a range of up to 5.6 m, limited only by the LIDAR's scan range, versus the 1 m limit of the first algorithm. Although this algorithm is more complicated because it involves a search algorithm and requires building a map, it provides more control and is easier to understand and visualize when compared to the first algorithm, discussed in *Section 3.3*.

Occupancy Grid Construction

To construct the occupancy grid each time the LIDAR is read, the following steps need to be carried out. Firstly, since it would be wasteful to recalculate the grid cells for the ring and the two lines of obstacles each time new data is received from the LIDAR, these cells are determined once and then stored in an array. Each time that the LIDAR is scanned, the contents of the array are copied into the new occupancy map before the actual obstacles detected by the LIDAR are inserted. Note that this initial calculation is also done for the map images that are displayed and the initial data is stored in a *Mat* variable. The data that is read from the LIDAR then needs to go through a conversion process so that coordinates for the occupancy map's two-dimensional array can be determined. *Figure 4.2* details this process.

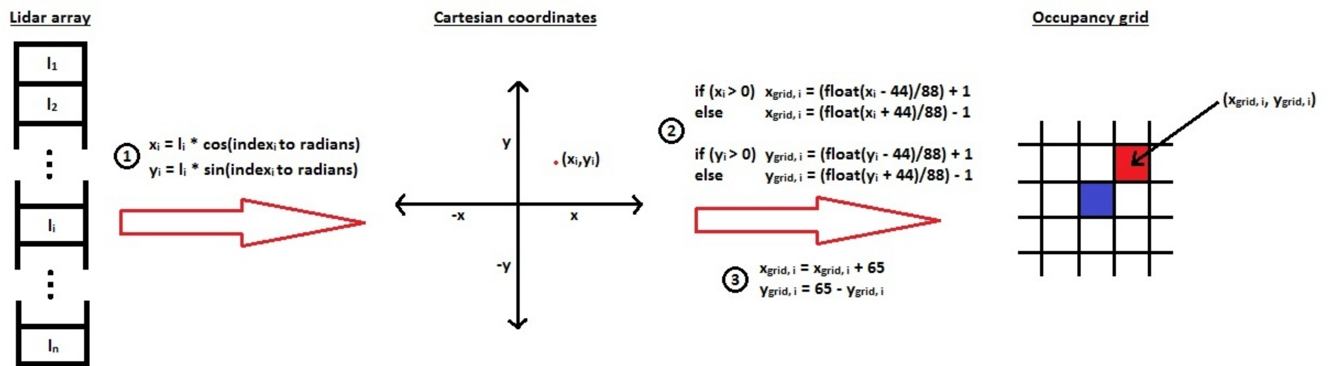


Figure 4.2: Conversion process

As shown in this figure, the LIDAR returns an array with each of its elements containing the appropriate distance values illustrated by l_1 to l_n. Since the LIDAR scans with a 0.35 degree resolution, each index of the LIDAR array is a scan 0.35 degrees greater than the previous scan and therefore can be associated with an angle. Therefore, since the LIDAR array holds distance and angle information, the Cartesian coordinates for each element of the array can be calculated. These coordinates are calculated using the first formula given in *Figure 4.2*.

Once the (x, y)-coordinates have been calculated, the coordinates for the occupancy grid can be calculated using the second set of formulas demarcated by 2 in *Figure 4.2*. The value 88 in these formulas is the size of the cells in the map and therefore, if a different cell size is used, this value would change accordingly. The value 44 is half the cell size and therefore would also change. The reason that half the cell size is added or subtracted is so that the cell that the robot, or rather the LIDAR, is present in is accounted for as all distances to obstacles are measured from the center of this cell. Note that a value of one is either added if x_i is positive, or subtracted if x_i is negative. This is so that when converting from a floating point value to an integer, the result is rounded up if x_i is positive, or rounded down if x_i is negative.

The results from those calculations are then translated according to the third set of formulas, so that (0, 0) is the center of the occupancy map. The red cell is the example cell that has been determined from the formulas discussed, whilst the blue cell is the center of the occupancy map containing the robot. The value 65 in these formulas depicts the row and the column number of the center of the occupancy map, since the map is represented using a two-dimensional array and so row and column zero represent its top left corner. If a different cell size or limit to the LIDAR scan range is used, the value 65

would also change. The two-dimensional array's columns, or x-coordinates, increase from left to right, and since this is the manner in which it should remain, the x-coordinate is translated by adding the value of 65. The rows of the array increase from top to bottom. Since the y-coordinates should increase from the bottom to the top, the additive inverse of the y-coordinate (i.e., negative y) is first calculated, after which it is translated by adding the value of 65.

After this conversion process, these coordinates are used for the occupancy grid as well as the two map images mentioned earlier. Since these images are stored in *Mat* variables, which have a similar structure to a two-dimensional array, this is possible. These images are illustrated in *Figure 5.11* in *Chapter 5*. The robot, as well as the goal cell, which is discussed later, are represented in these images as blue pixels; the obstacles as red pixels; and the path from the robot to the goal, also discussed later, as green pixels. Whereas, the robot, goal, obstacles and the path are represented as differently coloured pixels in the map images, in the occupancy grid itself, different integers are used for these representations.

While explaining how the occupancy map is built, the center cell was said to contain the robot itself. In fact, this cell actually contains the LIDAR, but will still be referred to as containing the robot. This is mentioned merely for completeness.

As mentioned earlier, the exaggeration map contains exaggerated obstacles and no obstacles in the near surroundings of the goal cell. There is also an algorithm that limits obstacle placement surrounding the robot, as mentioned in *Chapter 3*. Therefore, each time an obstruction is detected, there is slightly more logic and calculation involved than that showed in *Figure 4.2*. The pseudo code below describes the simplified process that takes place for each obstruction that the LIDAR detects.

```
   for each length in lidar array:
1.    calculate cartesian coordinates for obstruction
2.    calculate coordinates for occupancy map
3.    place obstacle in normal map image
      for the actual and each of the 24 extra obstacles:
4.       if obstacle being placed surrounding goal cell:
            continue
5.       else if obstacle NOT being placed surrounding source cell:
            place obstacle in exaggerated map
```

```
            place obstacle in occupancy grid

         //therefore obstacle being placed surrounding robot
6.       else if actual obstacle being placed
         OR actual obstacle not in surrounding cells
         OR actual obstacle in surrounding cells, but extra obstacle being
         placed 1 cell away (any direction) from the actual obstacle:
            place obstacle in exaggerated map
            place obstacle in occupancy grid

         //else don't place the obstacle!
```

The logic used in placing obstacles surrounding the robot's cell ensures that any extra obstacles that need to be added to a particular side of the robot, but which are exaggerations of an obstacle detected on a different side, are not inserted. This logic is shown in part 6 of the pseudocode. Therefore obstacles placed near to the robot do not block off traversable routes unnecessarily.

In order for the path from the start state to the goal state to be calculated, the goal coordinates on the occupancy grid need to be calculated. As discussed in *Chapter 3*, the goal cell is calculated using the distance from the robot to the marker and the angle between the front of the robot and the marker. The (x, y)-coordinates for the goal are calculated using *Equations 4.1* and *4.2*.

$$x = distance \times cos(\frac{angle \times \pi}{180}) \tag{4.1}$$

$$y = distance \times sin(\frac{angle \times \pi}{180}) \tag{4.2}$$

These coordinates are then scaled and translated to a cell on the occupancy map as shown in the process above. If the distance from the robot to the marker exceeds the distance to the outer ring, which is determined by the maximum area for the LIDAR to scan, the obstacle is placed at that limited distance. As in mode 1, in mode 2 a counter, *not_detected*, is kept to smooth out the detection process and therefore if the marker is not detected for a few frames and then is detected again, the obstacle avoidance process does not halt, but instead the last location of the goal is used. There is no need to estimate where the marker will be when it is not detected as the time period is so small owing to the high frame rate of the camera. Therefore, it is sufficient to use the last known goal coordinate.

A* Implementation

When first implementing the A* algorithm, the Boost C++ libraries[3] were used as they have their own built in A* search. Although the A* search itself performed reasonably, the paths it calculated were often not optimal and this was probably owing to the Euclidean heuristic that was used. Other than this, when using the Boost library methods, there was little control over decisions such as whether the robot should consider diagonal cells on the grid when calculating the path or only adjacent cells, and the manner in which the implementation was carried out seemed overly complicated for the purpose of this project.

Therefore, A* was implemented manually, closely following the method obtained from the Web[4]. This simpler implementation, once integrated into the system, performed much faster than when using the Boost library and was not overly complicated. Moreover, additional functionality could be added to the system with ease. Changing between different heuristics and considering diagonal cells for movement were achieved easily and changing to a different heuristic solved the issue of incorrect paths being calculated. Since performance was improved significantly and much more functionality was gained, this version of the A* algorithm was implemented. The exact differences in terms of performance are given in *Chapter 5*.

Appendix A.2 shows the code for the chosen A* implementation along with comments. Some of the less important sections of the implementation have been removed and replaced with comments to keep the listing compact. A brief explanation of this algorithm can be found in *Chapter 2*. In order to calculate the optimal path from one cell on the map to another, the *pathFind()* method in the code is called with the appropriate (x, y)-coordinates of the two cells. The function returns a string containing the directions to follow cell by cell.

As seen in the implementation, each cell or node in the occupancy map, when under consideration, consists of an x- and y-coordinate on the map, a level g(n), and a priority f(n). These attributes are stored using the *node* class. All the maps, namely, the occupancy map itself, the open list map representation, the closed list map representation and the map containing the back pointers, are represented using integer arrays as seen in lines 1-4. The open list is represented using a priority queue of *node* objects. This

[3]http://www.boost.org/

[4]http://code.activestate.com/recipes/577457-a-star-shortest-path-algorithm/

representation is used so that the attributes can be retrieved easily from each *node* object and so that the *nodes* are sorted by highest priority. The *nodes* in the open list need to be kept sorted by highest priority because the *node* with the lowest F value (highest priority) is chosen next - from where the process is repeated. In order for the priority queue to be kept sorted with the highest priority, which is the lowest priority value in this implementation, at the "top" of the queue, the '<' operator was overloaded so that when two nodes were compared, the priority of each node could be compared and the opposite Boolean result returned.

In this case, all eight cells surrounding the cell currently being evaluated were considered as neighbours and therefore in the *node* class's *estimate()* method, the Chebyshev heuristic was chosen as the heuristic of choice. If the number of plausible directions for movement in the code is changed from eight to four then the Manhattan heuristic, which has been commented out in line 48, would be preferable. If movement is not restricted and therefore all eight directions are valid, the cost of moving diagonally is higher than that of moving in a straight direction. This can be seen in the *node* class's *nextLevel()* method in line 43 where, if the movement is diagonal, the cost of moving is 14, whereas the cost is 10 when the movement is straight.

4.5 Searching

As mentioned in *Chapter 3*, when the marker is lost (mode 3), the process is paused for 5 s before searching for the marker begins. In order to do this, the ctime library was used to determine when 5 s had elapsed. Until these 5 s have elapsed, the camera is kept facing in the direction in which it lost the marker and the searching process is not begun. Once the time elapses, the first stage of the process is executed.

This first stage involves the camera panning from where the marker was last detected, all the way to the left or right, depending on which direction the marker was last detected in. The second stage is then panning the camera all the way in the opposite direction. It is panned by a value of 100 every third iteration of the main loop and therefore the speed at which the camera pans is proportional to the performance of the system as a whole. If the system's performance is poor, frames are either skipped, if the robots hardware is the cause, or only a few frames are analysed if the camera is at fault. Either way, if the system's performance is poor, detection will be difficult if the camera is panning too

quickly and on the other hand, if the systems performance is good, it is unecessary to pan the camera slowly. The manner is which this is implemented is therefore ideal.

In the third stage of the process, the robot turns in a 360 degree circle. This is achieved by using another counter, which is declared globally and thus can be accessed by both the *main* loop as well as the *robotmove* thread. When this third stage begins, the robot is issued a turn command in the direction of the last marker detection, and every 250 ms, the command is reissued. In the *robotmove* thread the counter is incremented by one every 250 ms after the turn command is given. In the *main* loop the counter is checked and once it exceeds a value of 37, which is a figure determined through experimentation, the robot's motors are no longer issued turn commands. This value of 37 is suitable only when the robot is on a tiled surface and is turned with a speed of 160 tics. If a different surface is used or the robot is turned at a different speed, a different value will need to be determined.

As mentioned in *Chapter 3*, if the marker has still not been detected, another 5 s delay is imposed, after which the process is repeated.

4.6 Using the System

To start, view and stop the system, the user must have access to the robot's desktop using a form of remote login such as VNC from either another machine connected via Ethernet or Wi-Fi, or from a mobile device such as an Android based tablet PC or smart phone, provided that the operating system running on the device supports a form of remote login and is Wi-Fi capable. An Android based mobile device that is Wi-Fi capable is recommended since VNC applications, such as androidVNC[5], are available freely from the Android Market, and more freedom is achieved when using a wireless mobile device as opposed to when using a wired, immobile device. To stop execution of the system, the user presses the escape key. This stops the robot immediately, disconnects from and closes all the devices such as the web camera, the RS232 port, and the LIDAR, and exits from the looping threads.

[5]http://www.freewarelovers.com/android/app/androidvnc

4.7 Summary

This chapter covered the implementation of the system and included several code listings to help clarify important sections. The system consists of three loops that run in parallel. A brief recap of the main tasks for which each loop is responsible for, is given below:

***main()* loop:** Handles marker detection, the following of the marker when in mode 1, as well as the searching process.

***robotmove* loop:** Responsible for the actual issuing of move commands to the robot based on decisions made in the other two loops.

***readlidar* loop:** Responsible for constructing the occupancy grid, calculating an optimal path to the marker and determining in which direction the robot should move.

Chapter 5

Results and Analysis

5.1 Introduction

In this chapter, experiments measuring the accuracy and the performance of the system are discussed, the results presented and an analysis of these results given. Variables concerning marker detection, occupancy grid design and the A* search algorithm itself are then varied and these experiments repeated for the different options. In each of these experiments, all variables currently not being varied are given their default values. These values result from the decisions that were originally made. For example, when the amount of environment that the occupancy grid represents is varied, the cell size of the grid is kept at 88 mm, the number of cells considered remains 8 and the Chebyshev heuristic is used. The code used in obtaining the results in this chapter has been left in the final code as comments.

Decisions made regarding the final system configuration are then listed, some of the system's capabilities illustrated and the limitations of the project discussed.

5.2 Marker Detection using ArUco

To increase detection accuracy of the markers or to increase the system's performance, various options regarding fiducial marker design as well as the configuration of the system were experimented with. The following subsections focus on these experiments and their results. All these experiments were carried out in a well, artificially lit environment.

5.2.1 Marker Size and Camera Resolution

Regarding the design of the fiducial marker, the size of the marker was varied in an attempt to gain accuracy, by increasing the size, or to reduce the amount of visual distraction, by reducing the size. Accuracy can also be gained by increasing the resolution of the web camera, or performance increased by decreasing this resolution. Since trade-offs between accuracy and the amount of visual distraction or performance are important to the success of the project, these experiments were performed.

Setup and Conduction of Experiment

Three different sized markers were printed out for testing. The side lengths of these markers measured 130 mm, 160 mm and 200 mm. Regarding camera resolution, three resolutions, 160x120, 320x240, and 640x480 were tested.

The robot was then set up with its web camera facing slightly upward, as is the case when the system is in use. The system was then started, with the restriction that no commands could be sent to the robot's motors or the web camera, to ensure that the robot and camera remained stationary, and the mode was manually set to mode 1 - following the marker normally. Therefore, the robot was limited to only detecting markers directly in front of it. The three markers were then, one at a time, held in front of the robot, at a height of 1.5 m above the ground. Once they were detected, the markers were moved further away from the robot's camera until they were no longer detected robustly. They were then edged towards the robot to find the maximum distance away from the robot that the marker was detected perfectly, without being lost in any frames. The distance along the ground, from the web camera to the marker, was then measured using a tape measure. Once, distances for all three markers had been recorded, the resolution of the camera was changed and the process repeated. Note that these distances measured are slightly less than the actual distance from the robot's camera to the marker, as the vertical distance between them has been ignored.

Results and Analysis

A graph showing the results obtained from this procedure is given in *Figure 5.1.* This graph clearly shows that by increasing the marker size, the maximum distance at which the marker can be detected, increases in a reasonably linear manner. Furthermore, it can be seen that when setting the camera to a resolution of 320x240, as compared to 160x120, a large increase in accuracy is noted. However, when further increasing the resolution to

640x480, accuracy increases only marginally. The rate at which the maximum distance increases as the resolution increases does not seem to be linear, but rather logarithmic.

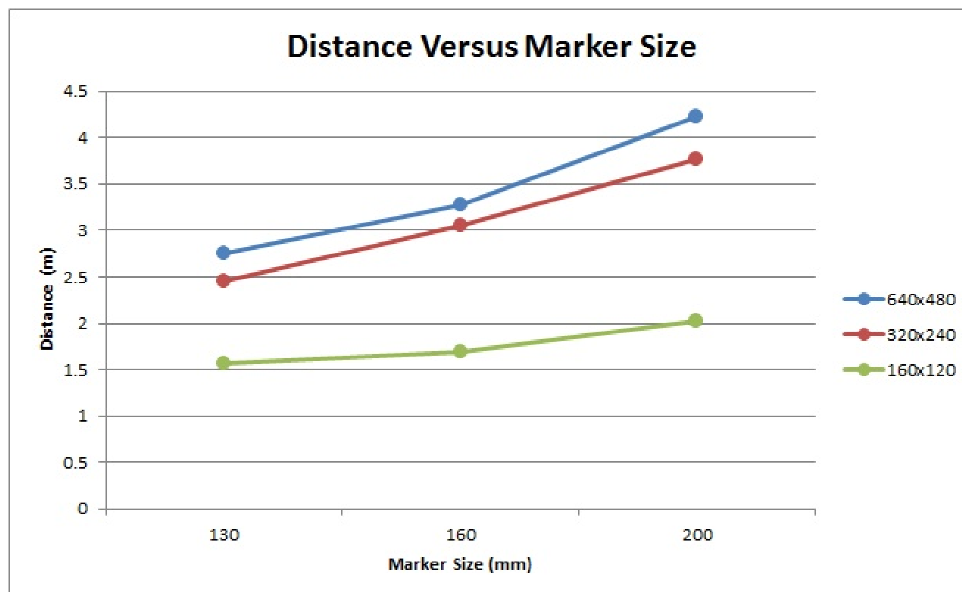


Figure 5.1: Distance versus marker size for different resolutions

During this experiment, the marker was also held at different orientations, at a distance slightly less than each marker's maximum distance. Some of these orientations, held close to the camera for illustration purposes, are shown in *Figure 5.2*. Variations in orientation did not affect detection at all.

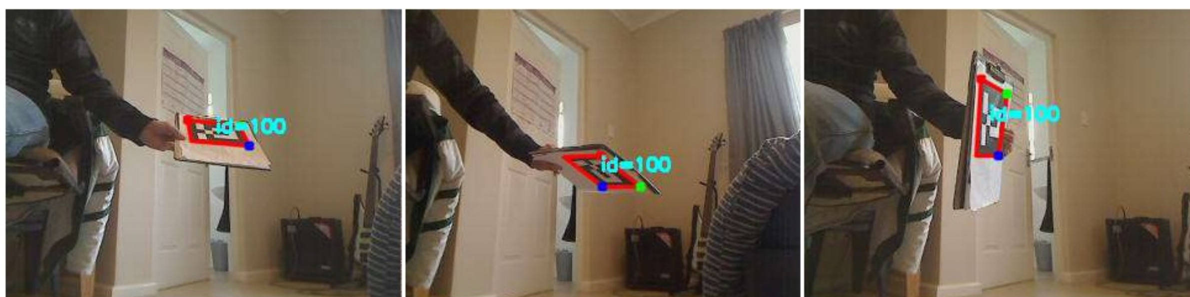


Figure 5.2: Detection at different orientations

Only when bright light is present in the environment, such as direct sunlight, does the orientation of the marker affect its chances of detection.

If the system is set up in a small environment, a smaller marker is often more appropriate as detection issues occur when larger markers are close to the camera. These issues

result owing to ArUco requiring the entire marker to be present in the camera frames and when a large marker is brought close to the camera, sections of the marker are not viewed. The opposite applies when the system is used in a large environment, as detection may need to occur from larger distances and therefore, a larger marker is more suitable.

The decisions arrived at in this project are for environments that are small to medium sized and therefore a maximum detection distance of around 3 m proved sufficient. This is mainly owing to the venue that the system was developed in. Detecting at a resolution of 160x120 is therefore not appropriate, as the maximum distance at which markers could be detected is well less than 3 m. Since a marker with a side length of 160 mm was detectable at 3 m when using either of the other two resolutions tested, and since greater distances were not a requirement, this size was chosen for the marker. If larger environments are used, when using this particular equipment, markers with side lengths greater than 200 mm, although still not too visually distracting, are not recommended as the LIDAR can only scan up to a range of 5.6 m, and better paths are calculated when the marker is kept within this range.

5.2.2 Detection Performance

As shown, when the resolution of the camera is increased, detection accuracy increases. However, when the resolution is increased, the performance of the system decreases. There are two reasons for this performance decrease. The first is that when using a high resolution on many inexpensive web cameras, such as the camera used in this system, even with all automatic lighting disabled, low frame rates are noted. This is without any additional processing, except merely the running of the camera in programs such as guvcview. The second reason is that ArUco takes longer to process larger frames, and so fewer frames from the web camera are processed. Either way, if fewer frames are processed, the usability of the system is affected.

Setup and Conduction of Experiment

Firstly, to check the performance of the camera itself, guvcview was used. The desired resolution was set and the frame rate of the camera was then observed.

Secondly, the performance of the system was tested. This was done by recording the time taken for the system to process 1000 frames from the camera, at the three different

resolutions experimented with in the previous section. This was achieved by incrementing a counter in the system's main loop from zero to 1000. This time was measured accurately by making use of the ctime library as follows. When the main loop was first entered, the system time was recorded, and once the counter had reached 1000, it was recorded again. Subtracting these two times gives the time taken. From these times, the frames processed per second for each resolution was calculated by dividing the value of 1000 by the recorded time.

Results and Analysis

The camera's performance proved excellent, achieving 30 frames per second, at resolutions up to 640x480. When the resolution was set higher though, to 960x720, the frames per seconds dropped to 14.5. Since, increasing the resolution above 640x480 seems unnecessary, as discussed earlier, this resolution was not considered further.

The results from testing the system's performance are given in *Table 5.1.*

Resolution	**Time (*seconds*)**	**Frames Per Second**
640x480	90.51	11.05
320x240	33.34	29.99
160x120	33.32	30.01

Table 5.1: Resolution effect on performance

At a resolution of 640x480, only 11.05 frames are processed per second. This large hit to the performance is due to ArUco's *detect()* method taking considerably longer to process the larger frames. However, at resolutions of 320x240 and 160x120, the system performs remarkably well, allowing frames to be processed at the camera's maximum frame rate. Even though the time taken to process 1000 frames, when lowing the resolution from 320x240 to 160x120, remains the same, there is less strain on the system. The time remains constant because the iteration speed of the system's main loop is capped at the maximum rate that frames can be received from the camera. If the system was implemented on a robot with less processing power, a resolution of 160x120 may be crucial to the performance of the system.

On this robot, when using a resolution of 320x240, frames were processed at their maximum speed. Owing to this 171.4% increase in performance from that when using a resolution of 640x480, and since as discussed in *Section 5.2.1*, the detection accuracy at this resolution is only marginally worse than when a resolution of 640x480 is used, this

resolution was the final choice for use in this project.

Detection from further distances, as well as increased performance could have been achieved by tracking a brightly coloured, simply shaped object - such as a bright yellow ball, instead of a fiducial marker. Detection distances could be increased because the camera would need to capture less detail, since no unique patterns like those present on fiducial markers would need to be detected. Since less detail is required, if the detection distance were decreased, the resolution of the camera could be lowered further, thus increasing performance. Performance could be increased further since a less demanding detection algorithm would be required. An example of such an algorithm is the Camshift[1] algorithm.

5.2.3 Automatic Lighting

In all the testing so far, the web camera's automatic lighting settings were manually disabled. When enabling the automatic exposure lighting setting, although detection accuracy is not increased when the system is being tested in a well-lit environment, when the environment is even slightly dark, these settings increase the accuracy significantly. Unfortunately using this setting comes at a huge performance cost as mentioned in *Chapter 3*, as the camera's frame rate is often reduced to single digits, causing the main loop to iterate slowly, thus causing the robot's movement to be jerky and the system unusable.

A single experiment, consisting of the same two tests performed in *Section 5.2.2*, was carried out to verify whether automatic lighting is plausible when using a low resolution (160x120). This was done in a reasonably lit environment, since when a darker environment is used, more automatic lighting is needed, which decreases the camera's performance.

Even at this low resolution, the camera's performance resulted in a frame rate of 14.5 frames per second, noted using guvcview, while the system's performance was capped at 14.33 frames per second. Owing to this poor performance and the detection accuracy lost when using a small resolution, using these settings on this particular camera does not seem plausible.

[1]Continuously Adaptive Mean Shift: an object tracking method based on colour

5.3 Occupancy Grid Design

As with the marker design, there were several options concerning the design of the occupancy map. The choice of option affects the optimality of the paths calculated and the time taken to calculate them. It is important that the paths calculated are as close to optimal as possible as unnecessary distance traversed is wasteful when considering that most mobile robots operate on battery power. Owing to the nature in which the A* search algorithm is used, these paths need to be calculated quickly, otherwise move commands may be given to the robot when they are no longer appropriate, causing the robot to behave erratically and possibly make contact with the obstacles.

The following subsections detail the experiments performed, their results and an analysis of these results.

5.3.1 Area Represented

The first choice regarding the design of the occupancy grid, is how much of the environment surrounding the robot should be represented. This is easily done by ignoring data received from the LIDAR that is further away than a specified distance, when building up the occupancy grid. Consider the example where the goal is placed directly in front of the robot, at a distance of 5.6 m (the maximum range that the LIDAR can scan). Therefore, when the LIDAR range is limited, the goal is contained in the cell directly in front of the robot, but at a distance further than that which the occupancy map can represent. When the robot begins to move towards the goal, it remains on the outskirts of the grid until it is closer than the range to which the LIDAR data is limited. Keeping this example in mind, limiting how much environmental data is represented by the grid, results in the occupancy grid having fewer cells. If the grid has fewer cells, when the A* algorithm is used to calculate a path from the robot to the goal, fewer cells need to be considered, thus boosting performance. This comes at a cost though. Not only might the path that the robot takes to the goal be unnecessarily longer than the optimal path, but incorrect decisions regarding which direction the robot should move in to avoid obstacles could be made, perhaps causing the robot to traverse into a dead-end.

Setup and Conduction of Experiment

For this experiment, the size of the environment that the occupancy map represents was varied. Three different sized occupancy grids were tested. These grids represent the area

within 1 m, 3 m and 5.6 m, respectively, from the LIDAR at all angles that the LIDAR is capable of scanning. These distances are hereafter referred to as the LIDAR's scan range. Tests on each grid were performed and the system's performance measured.

For each run of the experiment, the layout, described earlier in this section, was used. This layout is shown in *Figure 5.3*. Since the performance of the system differs according to different environmental layouts, all runs were performed keeping this layout unchanged.

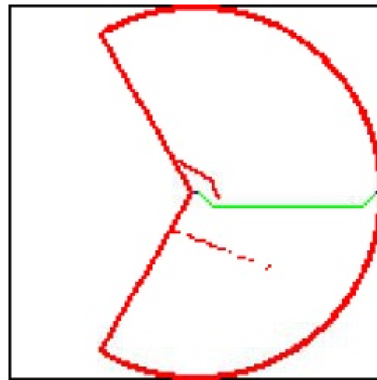


Figure 5.3: Standard layout used for experiments

As mentioned in the earlier example, the goal was placed 5.6 m away from the robot and therefore would be located in the cell on the border of the occupancy grid, directly in front of the robot, for each of the three experiments. *Figure 5.3* shows the setup of obstacles around the robot, where the goal has been placed, and the path that is calculated to the goal avoiding obstacles, in the case where the occupancy grid represents a scan range of 5.6 m. It is important to note that the obstacles placed to the left and in front of the robot are within 1 m of the robot. This was intentional, so that similar paths could be calculated in each experiment. Another point to note is that, the obstacles used throughout these experiments were reasonably low, not higher than 40 cm. This is because if obstacles are placed between the robot and the marker, the marker still needs to be in the camera's line of sight in order for the goal to be set.

To record the times taken for calculating paths on different sized grids, the robot needed to be kept stationary and so commands to the robot's motors were restricted. For each of the three tests, the time taken for the path, from the robot to the manually placed goal avoiding obstacles, to be calculated was recorded. This is referred to as test 1. This was done by recording the current system time just before the *pathFind()* function shown in *Appendix A.2*, was called, and then recording the system time again when the program

returned from this function. The time taken to calculate the path is therefore the difference of these two times. The time taken for the entire process - reading from the LIDAR, building the occupancy grid, updating the map images with obstacle data, calculating the path using the *pathFind()* function, updating the grid and images with path information, and deciding in which direction to move the robot - was also measured. We refer to this as test 2.

The three occupancy grids were then tested and the two timings recorded for each grid. These tests were repeated 20 times to achieve more accurate results for each of the occupancy grids, and the average times for the two timings calculated. Note that all performance results presented in the next sections are averages of 20 tests.

Results and Analysis

The results obtained from these tests are presented in *Table 5.2*.

Distance (mm)	**Total Cells**	**Test 1 Ave (ms)**	**Test 2 Ave (ms)**
5588	131x131=17161	145.1	248.8
3036	73x73=5329	33.8	135.9
1012	27x27=729	1.6	102.6

Table 5.2: Scan range effect on performance

Owing to the cell size of 88 mm used in this experiment, the LIDAR scan ranges were not exactly 1 m, 3 m and 5.6 m, but rather 1012 mm, 3036 mm and 5588 mm, respectively.

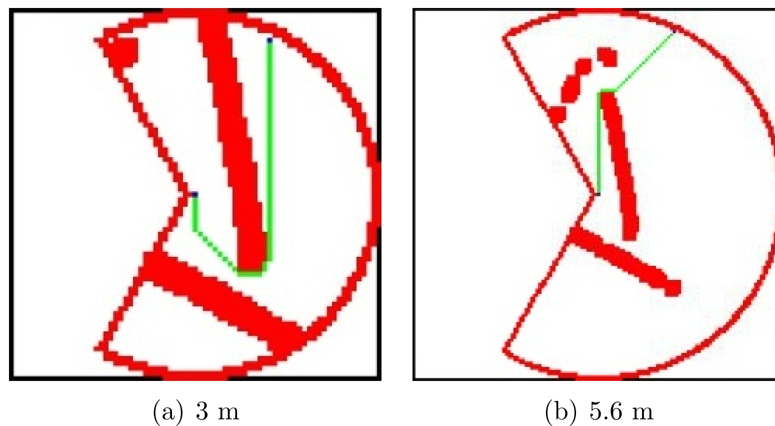
Since the amount of area that the grid represents is proportional to the total number of cells in the grid, these are included in the table. Even though many of the cells in the grid are never considered owing to the ring and two lines of obstacles placed on the grid, this should not affect the relationship between the number of cells in the grid and the performance of the two tests. As can be seen in the table, the relationship between the total number of cells and the time taken for the path to be calculated (test 1) is far from linear, but rather as the number of cells increase, the performance decreases at a much faster rate.

The times recorded for test 2 are roughly 100 ms more than those recorded for test 1. Since the path calculated in test 1 is also calculated in test 2, the overhead from the other tasks performed in this test adds just over 100 ms to the overall process, regardless of the number of cells in the grid. This constant time is due to the LIDAR, which is capable of scanning 10 times per second, which means that it takes 100 ms to scan once.

Since the time taken for test 2 is always roughly 100 ms more than that for test 1, this test was ignored in the remaining experiments.

Although, all the results in *Table 5.2* could indicate good performance, these results denote the performance associated with this particular layout only. In the majority of scenarios, the time taken for the tasks performed in test 2 to complete, was between 100 and 400 ms when the scan range was set to 5.6 m. However, when more complicated paths were calculated, the time taken increased to 800 ms using this particular configuration of variables, causing the system to seem slightly unresponsive, but still usable. When the scan range was decreased to 3 m, the worst performance noted was around 200 ms for the completion of the tasks in test 2.

Although the performance is significantly reduced when a larger scan range is used, the overall paths calculated are closer to being optimal. This is because more of the environment is considered each time paths are calculated and therefore, more informed routes are chosen. Under testing, this proved to be the case. In certain scenarios, shorter paths are chosen when larger scan ranges are used. One can see this by considering the scenario illustrated in *Figures 5.4(a)* and *5.4(b)*. In this scenario the goal has been set to the left of the robot, 5.6 m distant.



(a) 3 m (b) 5.6 m

Figure 5.4: Exaggerated maps using different scan ranges

In *Figure 5.4(a)*, the scan range has been limited to 3 m. Here an incorrect path to the goal has been calculated since crucial information regarding the environment has been ignored. In *Figure 5.4(b)*, the scan range is set to 5.6 m, and therefore more information regarding the robot's surroundings is taken into consideration. This extra information proves useful and allows an optimal path to the goal to be calculated.

Although the above scenario shows a case where it is preferable to have a larger scan range, in practice, these scenarios are not common and it is most often unnecessary. This is because the LIDAR cannot detect obstacles behind obstacles, causing many openings and blocked off areas further away, to remain undetected. Therefore, in most scenarios, where the area contains a reasonable number of obstacles, a smaller scan range, most often gives the same result as a larger one.

5.3.2 Occupancy Grid Resolution

The second decision regarding the occupancy grid's design, is how fine the resolution of the grid should be. This is done by varying how much of the environment each cell in the grid represents. By increasing the resolution of the grid, the number of cells in the grid increases, similar to varying the LIDAR's scan range, but the size of the surrounding environment represented by the grid remains the same, whereas this amount is increased when increasing the scan range. The process of determining in which cells of the occupancy map a detected obstacle should be placed, was explained in *Chapter 3*. *Figure 5.9* shows the formulas used for this process with a cell size of 88 mm. For different cell sizes, the values in these formulas need to be changed accordingly.

As the resolution of the occupancy grid is increased, the number of cells in the grid increase (cell size reduces), thus representing the environment more accurately. This allows for smaller openings to be traversed and tighter corners around obstacles to be taken, thus reducing the distance to the goal. However, as when increasing the scan range, when increasing the resolution, performance decreases. When the resolution decreases, the opposite occurs - an increase in performance is noted, but the representation of the robot's environment is less detailed. Openings in the environment that could be traversed are often not detected and when traversing around obstacles, extra distance is travelled.

Setup and Conduction of Experiment

For this experiment, the side length (cell size) of each cell in the occupancy grid was set to represent distances of 88 mm, 148 mm and 440 mm. Note that these sizes are all close multiples of 440 mm. This is so that extra obstacles can be placed surrounding the actual obstacles, filling up an area of 440 mm by 440 mm, and eliminating the chance of paths that are too narrow, from being calculated. When adjusting the cell size, the scan range

must also be adjusted slightly, and therefore scan ranges of 5588 mm, 5550 mm and 5500 mm for the respective cell sizes, were used. These slight adjustments are negligible in terms of performance.

To measure how the system performs when the cell size is varied, the first performance test, measuring the time taken for the *pathFind()* function to complete, performed in *Section 5.3.1* when varying the scan range, was repeated, but this time with the LIDAR scan range set back to 5.6 m and the cell size varied. The same environmental layout, as shown in *Figure 5.3* was used. Since the same layout was used and since varying the cell size affects the number of cells in the occupancy grid, which affects the performance, the results from these experiments are comparable with those in *Section 5.3.1*.

Using different cell sizes affects how many extra obstacles are placed around each actual obstacle. In the case of a cell size of 88 mm, obstacles are placed in all the cells within two cells from the actual obstacle's cell, as previously mentioned. When a cell size of 148 mm is used, obstacles are placed in all the cells one cell away from the actual obstacle's cell, and when using a cell size of 440 mm, no extra obstacles need to be placed. Adjusting the cell size also affects how many cells surrounding the goal cell should be kept traversable. However, the placement of obstacles surrounding the robot is not affected.

Results and Analysis

Table 5.3 shows the results obtained from this experiment.

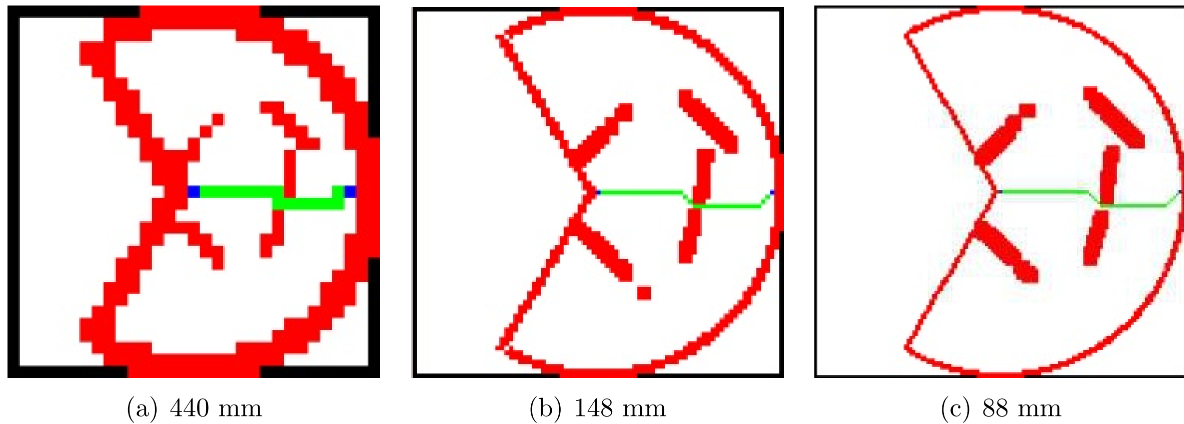
Cell Size (mm)	**Total Cells**	**Test 1 Ave (ms)**
88	131x131=17161	145.1
148	79x79=6241	34.4
440	29x29=841	1.6

Table 5.3: Grid resolution effect on performance

Once again, as the number of cells in the grid increases, the performance decreases rapidly. Since the number of cells in the grid depends on the size of the cell, making adjustments to the scan range or the cell size has the same impact on performance.

To show the differences in grid accuracy when adjusting the cell size, and therefore the optimality of the paths calculated, two scenarios are described. In the first scenario, which is illustrated in *Figures 5.5(a)*, *5.5(b)* and *5.5(c)*, two long obstacles were placed close to one another, creating an opening that the robot could fit through easily. The goal was

then set 5.6 m ahead of the robot as before and the path to the goal calculated. This was done for each of the three cell sizes. In this scenario, all three occupancy grid resolutions were successful, since the opening between the two obstacles was fairly wide.

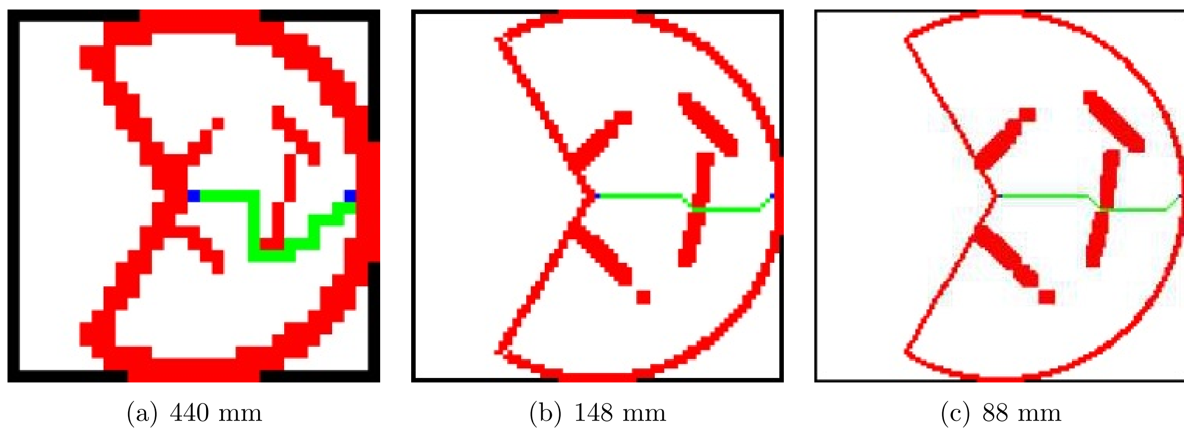


(a) 440 mm (b) 148 mm (c) 88 mm

Figure 5.5: Exaggerated maps using different cell sizes - scenario 1

Although all the resolutions were successful in this scenario, when using a cell size of 88 mm, shown in *Figure 5.5(c)*, the opening between the two obstacles was two cells wide, whereas when a block size of 148 mm or 440 mm was used, there was only one traversable cell between these obstacles. This shows the advantage of using a finer resolution.

The second scenario is similar to the first, but this time, the two obstacles were moved slightly closer together, but still leaving plenty of room for the robot to fit through. This scenario is shown in *Figures 5.6(a)*, *5.6(b)* and *5.6(c)*.



(a) 440 mm (b) 148 mm (c) 88 mm

Figure 5.6: Exaggerated maps using different cell sizes - scenario 2

This time, when using a cell size of 440 mm, the opening was not detected and an alter-

nate route was calculated. The opening was still detected when using the other, higher resolution occupancy grids. However, in most scenarios, if openings are reasonably wide, the same paths are calculated as when using finer resolutions. The reason for this is that when a higher resolution is used, each obstacle detected is exaggerated to represent an area of 440 mm by 440 mm of the environment, and therefore the only difference when using a higher resolution grid is that obstacles are pin pointed onto the map with more precision than when a lower resolution grid is used.

Since no extra obstacles are placed when using a cell size of 440 mm, when diagonal movement is allowed, the issue shown in *Figure 5.7* can occur. This is the same layout used in the second scenario discussed above and although the path calculated in this particular scenario is valid, as a wide enough opening does exist, in other scenarios this may not be the case. To solve this issue, when a cell size of 440 mm is used, diagonal movement is prohibited. In the results shown in *Table 5.3*, diagonal movement was not prohibited for this case and so the timings shown are therefore comparable with the other results.

This is, however, not an issue when extra obstacles are placed surrounding the actual obstacles. In the worst case, when a cell size of 148 mm is used, if two exaggerated obstacles are placed diagonally from one another and a route passing in-between them is calculated, then the minimum ground distance between these two obstacles is 418.6 mm, which is more than enough space for the robot to fit through.

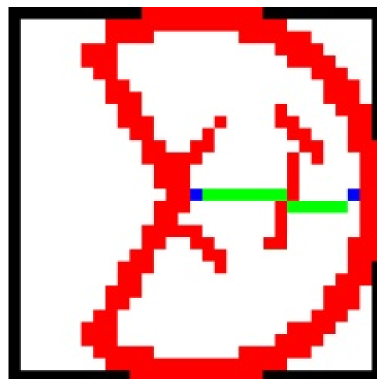


Figure 5.7: Invalid path calculated when using a cell size of 440 mm and considering diagonal movement

5.4 Path Calculation

In the previous section, the experiments that were performed dealt with the design of the occupancy grid. This section deals with the design of the A* search algorithm itself. As with the design of the occupancy grid, when adjusting the search algorithm's design, a trade-off exists between the optimality of the paths calculated and the time taken to calculate them.

The following subsections detail the experiments performed, their results and an analysis of these results, in order to find the balance between performance and optimality of the path.

5.4.1 Comparison of A* Implementations

First a decision was made regarding which implementation of the A* search algorithm, to use. As mentioned in *Chapter 4*, the A* algorithm was first implemented using the Boost C++ libraries. This was found to be over-complicated, making it difficult to add or change functionality, and the performance was inadequate. A second algorithm was then implemented, this time by coding the algorithm manually without the use of libraries. Since this implementation was simple in comparison, meaning that functionality could be added or adjusted easily, and it performed faster, it was chosen over the Boost implementation.

A similar experiment to that described in *Section 5.3.1*, using the same layout shown in *Figure 5.3*, was performed for each of the A* implementations. Since, with the Boost implementation, diagonal movement was not allowed and the Euclidian heuristic was used, and because this could not be changed easily, the chosen A* implementation was adjusted to match these settings. *Table 5.4* shows the results obtained when varying the maximum scan range for each of the implementations.

Distance (mm)	Boost A* (ms)	Manual A* (ms)
5588	253.3	5.6
3036	83.1	2.5
1012	26.9	0.6

Table 5.4: Performance comparison between A* implementations

As can be seen, the performance when using the Boost implementation is considerably worse, and at best performed 3224% slower than when using the manual implementation. This performance is even worse than that when diagonal movement is allowed with the manual A* implementation, as seen in the earlier results given in *Table 5.2*. These results also show the performance gained when diagonal movement is not considered. This is discussed next.

5.4.2 Number of Adjacent Cells Considered and Heuristics

Next, a decision whether to restrict movement to only four directions (horizontal and vertical) or to allow it in all eight directions (diagonals movement allowed) was made. When diagonal movement is allowed, more cells are considered when calculating the path and therefore more F values need to be calculated, reducing performance severely. However, as with all of these design decisions, there is a trade-off. When movement is restricted to four directions, paths calculated are not optimal and are most often far from being optimal. As mentioned in *Chapter 2*, when only horizontal and vertical movement is allowed, the Manhattan heuristic is preferred and when diagonal movement is allowed, the Chebyshev heuristic is more suited.

Setup and Conduction of Experiment

To benchmark the performance of the system, and to ensure that the results obtained are comparable with previous results, the same experiment performed in *Section 5.3.1*, measuring the time taken for the *pathFind()* function to complete, using the layout shown in *Figure 5.3*, was repeated. This test was carried out for each of the two cases - movement in four directions only, and movement in all eight directions. When testing, the appropriate heuristic in each case was used. The Euclidian heuristic was also tested in the case of movement in four directions to see how this heuristic affected the optimality of the paths calculated and the performance of these calculations.

Results and Analysis

When movement is limited to four directions using the Manhattan heuristic, paths calculated are not consistent. Each time a new path is calculated, while keeping the robot, goal and all the obstacles stationary, a different route to the marker is determined. On investigation, the reason for this was found to be due to the larger value that this heuristic calculates. When using the Euclidean or Chebyshev heuristics, or even by halving the

value that the Manhattan heuristic calculates, smaller values are realized and the paths calculated are more consistent. This is another reason why the Euclidean heuristic was included in the experiment.

Although using the Euclidean heuristic decreases the variation between the paths calculated considerably, when only considering movement in four directions, independent of the heuristic used, consecutive paths still differ quite drastically. *Figure 5.8* shows an example of this, where initially a path to the right of the obstacle is calculated, and a few milliseconds later, while the robot, goal and all the obstacles are kept stationary, another path to the left of the obstacle is calculated.

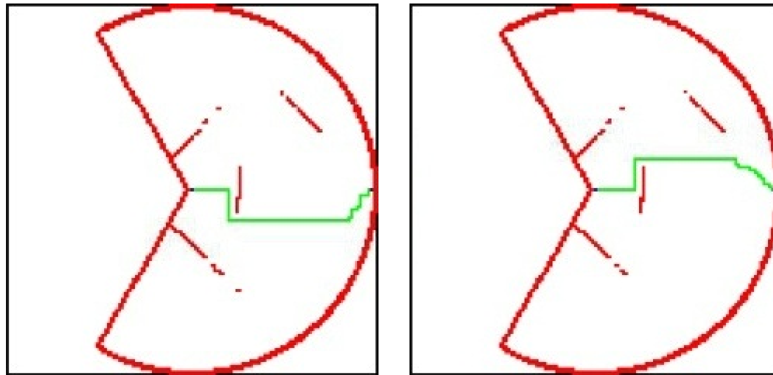


Figure 5.8: Paths deviate when movement is restricted to four directions

This sort of behaviour affects the usability of the system, as in practice, in this scenario, the robot turns to the right after the first path is calculated and then to the right after the calculation of the second path. This behaviour repeats, and no progress is made.

This is rarely an issue when considering diagonal movement and using the Chebyshev heuristic. In this configuration, paths hardly ever deviate from previously calculated paths provided that the map data and the goal are kept stationary.

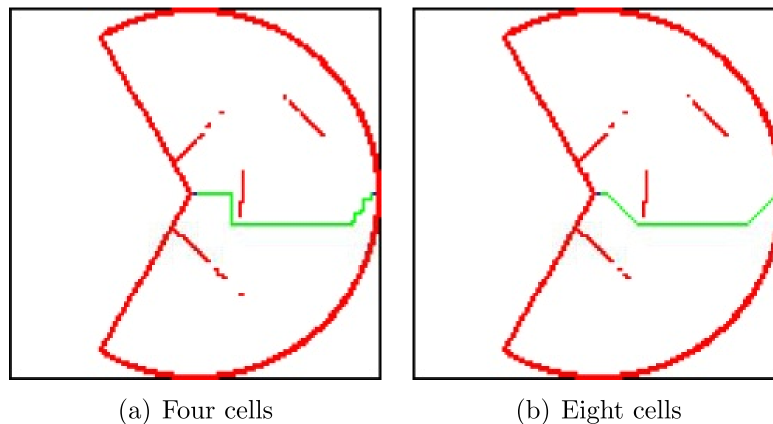
The results from the performance tests performed are presented in *Table 5.5*.

Cells Considered	**Heuristic**	**Test 1 Ave (ms)**
8	Chebyshev Distance	145.1
4	Manhattan Distance	19.2
4	Euclidean Distance	5.7

Table 5.5: Effect on performance based on the number of cells considered

As expected, by limiting movement to four directions, performance increases dramatically. However, the 236.8% performance increase noted when changing from the Manhattan to the Euclidian heuristic was unexpected. This increase seems to be due to the smaller H value calculated when using the Euclidean heuristic, as well as the Chebyshev heuristic for that matter. The Manhattan heuristic results in a larger H value for each considered cell, and therefore, also a larger F value for each cell. Owing to the many mathematical calculations and comparisons that occur in the A* implementation, these larger values hinder the performance of the algorithm. Since the performance is decreased and, as mentioned earlier, the paths calculated deviate excessively when using the Manhattan heuristic, this heuristic was not used in further experiments. Owing to its increased performance and more consistent path calculations, the Euclidean heuristic was used instead when movement was restricted to four directions.

Concerning the optimality of the paths calculated, shorter, and less jagged routes are determined when diagonal movement is allowed, compared to when it is prohibited. The scenario illustrated in *Figure 5.9* shows an example of this.



(a) Four cells (b) Eight cells

Figure 5.9: Different paths calculated when four and eight cells are considered

Not only is extra mileage added to the routes calculated, but in certain scenarios, openings detected cannot be passed though. This occurs in scenarios where the openings are fairly small, and at angles to the robot, and so a diagonal path needs to be calculated.

5.5 Final Decisions and Limitations

Final Decisions

After experimentation, decisions regarding the various variables discussed were made. Note that these variables should be set according to the environment that the system is used in and therefore the configuration decided on may not be suitable in different settings. The final configuration for the robot is as follows:

- A marker size of 160 mm by 160 mm was used
- The camera resolution was set to 320x240 and all automatic lighting was disabled
- The occupancy grid was set to represent a limited range of 3 m from the LIDAR
- Each cell in the occupancy grid was set to represent 148 mm by 148 mm of the environment
- Diagonal movement was allowed and therefore, the Chebyshev heuristic was used

Remarkable performance was noted when using the above configuration, while still allowing optimal, or close to optimal paths to be determined.

As previously mentioned, the detection distance when using a marker with a side length of 160 mm along with a camera resolution of 320x240, was sufficient for the small to medium sized environments in which the system was used. Since this resolution was sufficient and automatic lighting was disabled, huge performance increases were noted. With this configuration, tracking and following the marker, in mode 1, were successful and robust, provided a well-lit environment was used.

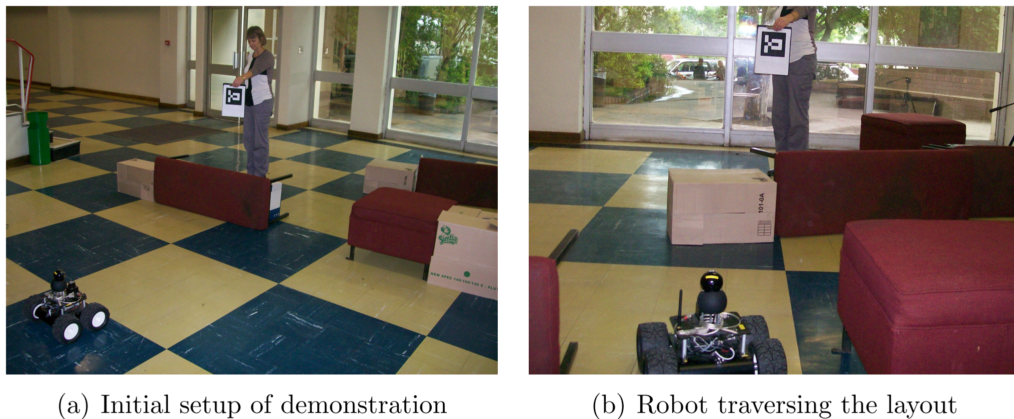
Reducing the scan range from 5.6 m to 3 m rarely affected the optimality of the path calculated, as mentioned earlier, owing to the LIDAR being unable to detect obstacles that are behind other obstacles, and therefore most often, the extra information obtained when using a scan range of 5.6 m is wasted. However, by reducing this scan range, performance was hugely increased. This choice was also impacted by the marker size and camera resolution selected, as this allows robust detection of the marker at distances up to roughly 3 m. Since it was rare that the paths calculated when using a cell size of 88 mm, as opposed to when using a cell size of 148 mm, differed, and since the performance gained when using a cell size of 148 mm was large, a cell size of 148 mm was selected.

Owing to the extra mileage added to the routes calculated, and especially owing to the deviation between consecutive routes, movement restricted to only the horizontal and vertical directions was decided against. Since movement could occur in all eight directions,

the Chebyshev heuristic was used.

System in Action

A video of the robot, set with this final configuration can be found on the accompanying CD. This video shows the system at work and demonstrates the robot in each of the three different modes. To demonstrate the robot avoiding obstacles in mode 2, the robot was set up in the layout shown in *Figure 5.10(a)*.



(a) Initial setup of demonstration (b) Robot traversing the layout

Figure 5.10: Example layout for demonstration

As seen in this figure, the marker was intially held in front of the robot. As the robot moved along its calculated route towards the marker, the marker was moved away from the robot, keeping the distance between the robot and the marker between 2.5 and 3 m. The route that the robot traversed was through the opening between the two obstacles in the first set, as shown in *Figure 5.10(b)* and then around the left of the final set of obstacles. *Figure 5.11* shows a screenshot of the system captured while the robot was traversing this layout.

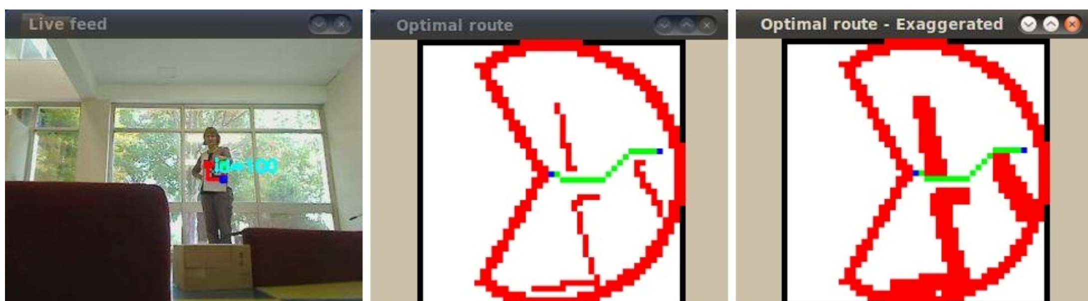
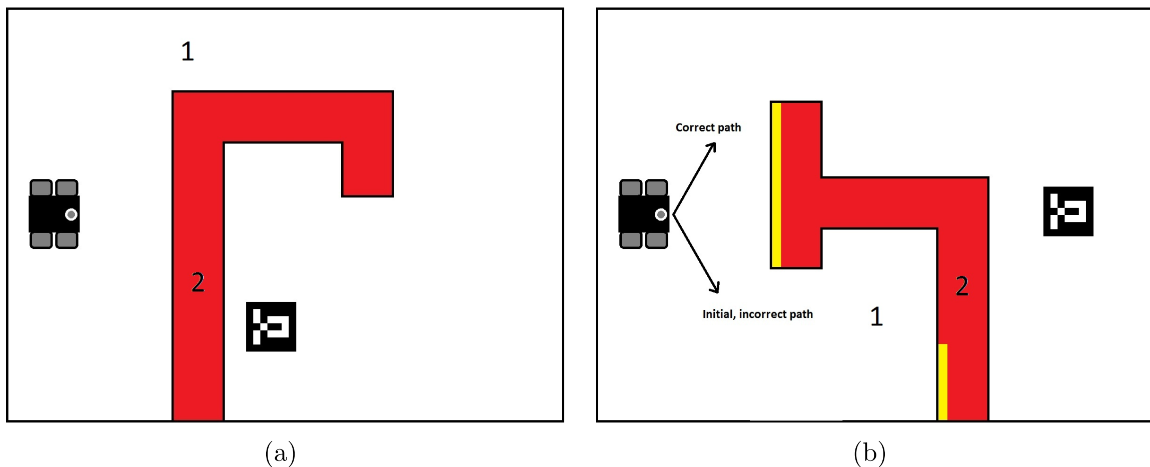


Figure 5.11: System in action

Project Limitations

Various setbacks were encountered throughout this project. The largest of these was the inaccuracy of the robot's odometer readings and this was the reason for many of the changes made to the project. As a result of this inaccuracy, it was not possible to create an occupancy grid of the area, but rather the map was rebuilt from scratch each time the environment was scanned. This meant that the robot could not traverse through certain obstacle layouts. *Figure 5.12* illustrates some of these unsuccessful layouts.



(a) (b)

Figure 5.12: Unsuccessful layouts

The layout in *Figure 5.12(a)*, illustrates an example where the robot initially calculates a path to the marker, but then when it turns to the left to follow this calculated path, it cannot keep track of the marker any longer. This is due to the web camera being unable to pan to such a wide angle. Even if it could, as soon as the robot traversed near to the area marked 1 on the figure, the LIDAR would not be able to detect the section of the obstacle, labelled 2, that was originally obstructing its route to the marker. Therefore, the robot would turn back again. To solve this issue, previous map information is needed and therefore a map needs to be created and continuously updated as more information is found. Owing to the inaccurate odometer readings, this was not possible.

The layout in *Figure 5.12(b)*, shows an example where the initial path calculated is incorrect. This path is chosen as the LIDAR only detects the parts of the obstacles highlighted in yellow and therefore, with only this information, the path seems valid. This is a limitation of the equipment. However, when the robot traverses this incorrect path to position 1, marked on the figure, the previously undetected obstacles are detected by the LIDAR. Since this is a dead-end, the robot needs to turn back. If this were to happen, the obstacle

information, marked 2 on the figure, would be lost and the robot would therefore turn around again to face its original direction. Turning around is therefore prohibited by the placement of the ring and the two lines of obstacles on the occupancy grid, and therefore in this situation it would not be possible for a path to the goal to be calculated.

5.6 Summary

The results presented in this chapter indicate that by reducing the accuracy of the system marginally, huge performance increases can be achieved. For this reason, even when using a powerful robot such as Wifibot, it proved useful to lower these settings slightly, as the marker detection distance proved sufficient and the paths chosen, in most scenarios, remained optimal. By doing so, the performance, and therefore the usability, of the system was improved significantly. If the system was implemented on a robot with lower specifications, these variables could be set with lower values. Although accuracy regarding marker detection and/or the paths calculated would be reduced, a well chosen configuration of these settings should allow for sufficient accuracy.

Chapter 6

Conclusions

6.1 Project Summary

The objective of this project was the creation of a system that allows for the navigation of a mobile robot in open or busy environments using a fiducial marker. This system needs to perform efficiently since the system is targeted at mobile robots, many of which have low processing power. An investigation into the tools available for marker detection and the search algorithms appropriate for this project's goals was conducted. This investigation also included previous research done in these areas. After decisions regarding the options discussed in the literature survey had been made, the proposed system was designed and implemented. The system's accuracy and performance were then evaluated, after which its capabilities were discussed.

6.2 Revisiting the Objectives

The first research objective concerning the system created was the accurate detection of markers using a standard web camera. This objective has been met as shown by the results presented in *Chapter 5*. These results also indicate the success of the second objective, to use this marker to allow a mobile robot to navigate autonomously independent of its surroundings. Although not all layouts were traversable owing to the setbacks encountered, in most practical scenarios, navigation was possible. The last objective was ensuring that the system performs efficiently so that it is usable on less powerful mobile

robots. Although the nature in which the A* search algorithm is used is not efficient, the benchmarks performed in *Chapter 5* indicate that the system can be configured to perform adequately, without significantly affecting accuracy. This final objective was therefore also met.

In *Chapter 2*, a further objective was mentioned: that the search algorithm chosen should be capable of functioning whether the environment was known, partially known or completely unknown. Unfortunately, owing to the inaccuracy of the robot's odometer and therefore the nature in which the search algorithm was used, the benefits of known or partially known environments could not be utilized. Thus, all environments were effectively treated as unknown environments. Although the system can be given a map of the area, once the robot moves, it cannot localize itself on the map.

6.3 Future Work

There are several extensions that can be made to the existing system and owing to time constraints, have been left as future work. These are as follows:

Solving the inaccurate odometry: This could be achieved by using the system on a robot with an accurate odometer, but this may prove costly. Another solution is to add either two trailing wheel encoders, or a single trailing wheel encoder and a digital compass, to the Wifibot robot. Either of these options could provide the robot with accurate odometry as well as information regarding direction. Another, more interesting solution, would be to attach at least two optical mouse sensors, at least one on each side of the robot. Provided that the surface used is smooth, this would allow for the distance travelled as well as the direction of travel to be determined. After this issue has been solved, the search algorithm, perhaps now changed from A* to Moving Target D* Lite, could be implemented in the manner that it was intended, thus resulting in a large increase in performance. A wandering algorithm could also then be implemented in the manner mentioned in *Chapter 3*.

360 Degree marker detection: The marker could be detected at any angle from the robot if a camera with a 360 degree panning capability is used. This may prove costly, and so a cheaper solution would be to use multiple standard web cameras in the manner mentioned in *Chapter 3*.

Bibliography

[1] ARToolKit. ONLINE. Available from: http://www.hitl.washington.edu/artoolkit/.

[2] Artoolkit - based tracking: A new device in VRPN. Tech. rep., Universidad de los Andes, 2007.

[3] Brunner, J. ArUco: Augmented Reality library from the University of Cordoba. ONLINE. Available from: http://www.ros.org/doc/api/aruco_pose/html/index.html.

[4] Coombs, J., and Prabhu, R. OpenCV on TI's DSP+ARM platforms: Mitigating the challenges of porting OpenCV to embedded platforms. White Paper - Texas Instruments, 2011.

[5] Demetriou, G., Vystavkin, A., Anastasiou, S., Theofilou, A., Giannopoulos, C., and Yerolemou, D. Indoor mobile robot localization using a wireless network: Wifibot case study. In *IC-AI* (2008), H. R. Arabnia and Y. Mun, Eds., CSREA Press, pp. 668–674.

[6] Fiala, M. ARToolKit applied to panoramic vision for robotic navigation. In *Proceedings of the Vision Interface, Halifax, Nova Scotia, Canada, June 11-13, 2003* (2003).

[7] Fiala, M. ARTag, an improved marker system based on ARToolKit. Tech. rep., National Research Council of Canada, July 2004.

[8] Hirzer, M. Marker detection for augmented reality applications. *Image, Rochester, NY* (2008).

[9] Hornecker, E., and Psik, T. Using ARToolKit markers to build tangible prototypes and simulate other technologies. In *Proceedings of the 2005 IFIP TC13 in-*

ternational conference on Human-Computer Interaction (Berlin, Heidelberg, 2005), INTERACT'05, Springer-Verlag, pp. 30–42.

[10] Kawakami, A., Tsukada, K., Kambara, K., and Siio, I. Potpet: pet-like flowerpot robot. In *Proceedings of the fifth international conference on Tangible, embedded, and embodied interaction* (New York, NY, USA, 2011), TEI '11, ACM, pp. 263–264.

[11] Koenig, S., and Likhachev, M. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics 21*, 3 (2005), 354–363.

[12] Kortenkamp, D., and Weymouth, T. Topological mapping for mobile robots using a combination of sonar and vision sensing. In *Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)* (Menlo Park, CA, USA, 1994), AAAI'94, American Association for Artificial Intelligence, pp. 979–984.

[13] Kraetzschmar, G. K., Pagès Gassull, G., and Uhl, K. Probabilistic quadtrees for variable-resolution mapping of large environments. In *Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles* (Lisbon, Portugal, July 2004), M. I. Ribeiro and J. Santos Victor, Eds., Elsevier Science.

[14] Kuffner, J. Efficient optimal search of Euclidean-cost grids and lattices. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'04)* (September 2004), IEEE.

[15] Lester, P. A* Pathfinding for Beginners. ONLINE, July 2005. Available from: `http://www.policyalmanac.org/games/aStarTutorial.htm`.

[16] Li, Y., Wang, Y., and Liu, Y. Fiducial marker based on projective invariant for augmented reality. *J. Comput. Sci. Technol. 22*, 6 (2007), 890–897.

[17] Marshall, D. Heuristic Search. ONLINE. Available from: `http://www.cs.cf.ac.uk/Dave/AI2/node23.html`.

[18] Mutka, A., Miklic, D., Draganjac, I., and Bogdan, S. A low cost vision based localization system using fiducial markers. *World Congress 17* (2008), 9528–9533.

[19] Nosrati, M., Karimi, R., and Hasanvand, H. A. Investigation of the * (star) search algorithms: Characteristics, methods and approaches. *World Applied Programming 2*, 4 (April 2012), 251–256.

[20] Owen, C. B., Xiao, F., and Middlin, P. What is the best fiducial? In *The First IEEE International Augmented Reality Toolkit Workshop* (Darmstadt, Germany, Sept. 2002), pp. 98–105.

[21] Patel, A. Heuristics. ONLINE, 2012. Available from: `http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html`.

[22] Portugal, D., and Rocha, R. P. Extracting topological information from grid maps for robot navigation. In *Proc. of 4th Int. Conf. on Agents and Artificial Intelligence (ICAART'2012)* (Vilamoura, Algarve, Feb. 2012), pp. 137–143.

[23] Sementille, A. C., Lourenço, L. E., Brega, J. R. F., and Rodello, I. A motion capture system using passive markers. In *Proceedings of the 2004 ACM SIGGRAPH international conference on Virtual Reality continuum and its applications in industry* (New York, NY, USA, 2004), VRCAI '04, ACM, pp. 440–447.

[24] Speers, A., Topol, A., Zacher, J., Codd-Downey, R., Verzijlenberg, B., and Jenkin, M. Monitoring underwater sensors with an amphibious robot. In *Proceedings of the 2011 Canadian Conference on Computer and Robot Vision* (Washington, DC, USA, 2011), CRV '11, IEEE Computer Society, pp. 153–159.

[25] Stanco, F., Tanasi, D., Gallo, G., Buffa, M., and Basile, B. Augmented perception of the past. The case of Hellenistic Syracuse. *Journal of Multimedia 7*, 2 (2012), 211–216.

[26] Stentz, A. The focussed D* algorithm for real-time replanning. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2* (San Francisco, CA, USA, 1995), IJCAI'95, Morgan Kaufmann Publishers Inc., pp. 1652–1659.

[27] Stentz, A. T. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)* (May 1994), vol. 4, pp. 3310 – 3317.

[28] Stentz, A. T. Map-based strategies for robot navigation in unknown environments. In *Proceedings of the AAAI Spring Symposium on Planning with Incomplete Information for Robot Problems* (March 1996).

[29] Sumbal, M. S. U. K. Environment detection and path planning using the e-puck robot. Master's thesis, University of Girona, 2010.

[30] SUN, X., KOENIG, S., AND YEOH, W. Generalized adaptive A*. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 1* (Richland, SC, 2008), AAMAS '08, International Foundation for Autonomous Agents and Multiagent Systems, pp. 469–476.

[31] SUN, X., YEOH, W., AND KOENIG, S. Generalized fringe-retrieving A*: faster moving target search on state lattices. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1* (Richland, SC, 2010), AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, pp. 1081–1088.

[32] SUN, X., YEOH, W., AND KOENIG, S. Moving target D* Lite. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1 - Volume 1* (Richland, SC, 2010), AAMAS '10, International Foundation for Autonomous Agents and Multiagent Systems, pp. 67–74.

[33] THRUN, S. Learning occupancy grid maps with forward sensor models. *Auton. Robots 15*, 2 (Sept. 2003), 111–127.

[34] THRUN, S., AND BCKEN, A. Learning maps for indoor mobile robot navigation. *Artificial Intelligence 99* (1998), 21–71.

[35] THRUN, S., AND BUCKEN, A. Integrating grid-based and topological maps for mobile robot navigation. In *Proceedings of the thirteenth national conference on Artificial intelligence - Volume 2* (1996), AAAI'96, AAAI Press, pp. 944–950.

[36] WOODEN, D. T. *Graph-based path planning for mobile robots*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2006. AAI3248795.

[37] XU, A., AND DUDEK, G. Fourier tag: A smoothly degradable fiducial marker system with configurable payload capacity. In *Proceedings of the 2011 Canadian Conference on Computer and Robot Vision* (Washington, DC, USA, 2011), CRV '11, IEEE Computer Society, pp. 40–47.

Appendix A

Code Listings

A.1 Methods Used in Issuing Move Commands

```
1  int setRS232Motor33f(int hUSB, short speed1, short speed2, unsigned char
       speedFlag) {
2    unsigned int n;
3    unsigned char sbuf[20];
4    int ress = 0;
5
6    sbuf[0] = 255;
7    sbuf[1] = 0x07;
8    sbuf[2] = (unsigned char) speed1;
9    sbuf[3] = (unsigned char) (speed1 >> 8);
10   sbuf[4] = (unsigned char) speed2;
11   sbuf[5] = (unsigned char) (speed2 >> 8);
12   sbuf[6] = speedFlag;
13
14   short mycrcsend = crc16(sbuf + 1, 6);
15
16   sbuf[7] = (unsigned char) mycrcsend;
17   sbuf[8] = (unsigned char) (mycrcsend >> 8);
18
19   ress = writers232(hUSB, sbuf, 9, &n);
20   return ress;
21 }
22
23 short crc16(unsigned char *buffer, unsigned char max) {
24   unsigned int crc = 0xFFFF;
25   unsigned int polynome = 0xA001;
```

```
26     unsigned int cptoctet, cptbit, parity;
27
28     for (cptoctet = 0; cptoctet < max; cptoctet++) {
29         crc ^= *(buffer + cptoctet);
30
31         for (cptbit = 0; cptbit <= 7; cptbit++) {
32             parity = crc;
33             crc >>= 1;
34             if (parity % 2 == 1)
35                 crc ^= polynome;
36         }
37     }
38
39     return(crc);
40 }
41
42 //write to the rs232 port
43 int writers232(int hUSB, unsigned char *buffer, unsigned int
       nNumberOfBytesToWrite, unsigned int *lpNumberOfBytesWritten) {
44     if (!hUSB)
45         return 0;
46
47     *lpNumberOfBytesWritten = 0;
48     for (int kk = 0; kk < nNumberOfBytesToWrite; kk++)
49         *lpNumberOfBytesWritten += write(hUSB, buffer + kk, 1);
50
51     return (*lpNumberOfBytesWritten >= 0);
52 }
```

A.2 A* Implementation

```
1 static int occupancy_map[n][m];
2 static int closed_nodes_map[n][m]; //map of closed (tried-out) nodes
3 static int open_nodes_map[n][m]; //map of open (not-yet-tried) nodes
4 static int dir_map[n][m]; //map of directions
5 const int dir = 8; //number of possible directions to go at any position
6 static int dx[dir] = {1, 1, 0, -1, -1, -1, 0, 1}; //{1, 0, -1, 0} if dir=4
7 static int dy[dir] = {0, 1, 1, 1, 0, -1, -1, -1}; //{0, 1, 0, -1} if dir=4
8
9 class node {
10     int xPos, yPos; //current position
11     int level; //total distance already travelled to reach node
```

```
12      int priority; //priority=level+remaining distance estimate; smaller
            priority is higher
13
14      public:
15          node(int xp, int yp, int d, int p) {
16          xPos = xp;
17          yPos = yp;
18          level = d;
19          priority = p;
20      }
21
22      int getxPos() const {
23          return xPos;
24      }
25
26      int getyPos() const {
27          return yPos;
28      }
29
30      int getLevel() const {
31          return level;
32      }
33
34      int getPriority() const {
35          return priority;
36      }
37
38      void updatePriority(const int & xDest, const int & yDest) {
39          priority = level + estimate(xDest, yDest) * 10; //a*
40      }
41
42      void nextLevel(const int & i) { //i is the direction
43          level += (dir == 8 ? (i % 2 == 0 ? 10 : 14) : 10); //better priority
                to going straight than diagonally
44      }
45
46      //estimation function for remaining distance to goal
47      const int & estimate(const int & xDest, const int & yDest) const {
48         //return (abs(xDest - xPos) + abs(yDest - yPos)); //manhattan
               distance heuristic
49         return (max(abs(xDest - xPos), abs(yDest - yPos))); //chebyshev
               distance heuristic
50      }
51  };
```

```
52
53 string pathFind(const int & xStart, const int & yStart, const int & xFinish
   , const int & yFinish) {
54
55   //two priority queues so that one can be used as a temp
56   //these priority queues contain the list of open (not-yet-tried) nodes
57    static priority_queue<node> pq[2];
58    static int pqi = 0; //priority queue index
59    static node* n0;
60    static node* m0;
61    static int i, j, x, y, xdx, ydy;
62    static char c;
63
64    //reset the node maps...
65
66    //create the start node and push into the list of open nodes
67    n0 = new node(xStart, yStart, 0, 0);
68    n0 -> updatePriority(xFinish, yFinish);
69    pq[pqi].push(*n0);
70    open_nodes_map[x][y] = n0 -> getPriority(); //mark on the open map
71
72    //a* search
73    while(!pq[pqi].empty()) {
74
75        //get the node with highest priority from open list
76        n0 = new node(pq[pqi].top().getxPos(), pq[pqi].top().getyPos(), pq[
            pqi].top().getLevel(), pq[pqi].top().getPriority());
77        x = n0 -> getxPos();
78        y = n0 -> getyPos();
79
80        pq[pqi].pop(); //remove the node from the open list
81        open_nodes_map[x][y] = 0;
82        closed_nodes_map[x][y] = 1; //mark it on the closed nodes map
83
84        //stop searching when goal state is reached
85        //generate path from finish to start by following directions
86        if((x == xFinish) && (y == yFinish)) {
87            string path = "";
88            while(!((x == xStart) && (y == yStart))) {
89                j = dir_map[x][y]; //pointer to parent
90                c = '0' + (j + dir/2) % dir;
91                path = c + path;
92                x += dx[j];
93                y += dy[j];
```

```
94              }
95
96              //perform some garbage collection...
97
98              return path;
99          }
100
101         //generate moves (child nodes) in all possible directions
102         for (i = 0; i <dir; i++) {
103             xdx = x + dx[i];
104             ydy = y + dy[i];
105
106             //check that nodes coordinate is valid, no obstacle exists
                    there and node is not in closed list
107             if(!(xdx < 0 || xdx > n - 1 || ydy < 0 || ydy > m - 1 ||
                    occupancy_map[xdx][ydy] == 1 || closed_nodes_map[xdx][ydy]
                    == 1)) {
108                 m0 = new node(xdx, ydy, n0 -> getLevel(), n0 -> getPriority
                        ()); //generate a child node
109                 m0 -> nextLevel(i);
110                 m0 -> updatePriority(xFinish, yFinish);
111
112                 //if not in the open list then add it
113                 if (open_nodes_map[xdx][ydy] == 0) {
114                     open_nodes_map[xdx][ydy] = m0 -> getPriority();
115                     pq[pqi].push(*m0);
116                     dir_map[xdx][ydy] = (i + dir/2) % dir; //mark its
                            parent node direction
117                 }
118
119                 //if already in open list but has higher priority
120                 //update priority and parent info
121                 else if (open_nodes_map[xdx][ydy] > m0 -> getPriority()) {
122                     open_nodes_map[xdx][ydy] = m0 -> getPriority();
123                     dir_map[xdx][ydy] = (i + dir/2) % dir;
124
125                     //replace node by emptying one queue into the other,
126                     //ignoring node to be replaced...
127
128                     pq[pqi].push(*m0); //add better node instead
129                 }
130                 else
131                delete m0; //garbage collection
132             }
```

```
133             }
134             delete n0; //garbage collection
135         }
136         return ""; //no route found
137 }
```