Fourth Year Project

Autonomous Robot Operation for Simulated Disaster Environments

Authors: Jonathan Abbey (0616961)
Adish Bharadwaj (0530157)
Jason Cook (0609664)
James Griffin (0519038)
Jan Vosecky (0534192)
Supervisors: Sarabjot Anand
Nasir Rajpoot

Department of Computer Science
University of Warwick
2010

Abstract

Recent natural disasters have highlighted the need for more efficient search-and-rescue techniques, in particular the use of robotics for both tele-operated and automated survivor detection. The RoboCup Rescue league is designed to test robots in simulated disaster environments and encourage research and development in this field. This year, Warwick Mobile Robotics have built a robot for autonomous use, and in collaboration with this group, we have developed software to enable the robot to compete in the competition. This makes use of Simultaneous Localisation and Mapping with autonomous navigation techniques, as well as image-based Victim Identification using a combination of different methods such as image processing with both standard and Infrared cameras. We will present our solutions, and discuss the results from the competition. Finally, we will demonstrate the potential for future iterations of the software and present a potential specification for future developers.

Keywords: Autonomous Robotics, Simultaneous Localisation and Mapping, Image Processing, Face Detection, Ellipse Fitting

Acknowledgements

We would like to thank our supervisors Professors Nasir Rajpoot and Sarabjot Anand for their support and guidance throughout the project. We would also like to thank the Warwick Mobile Robotics group, in particular Professor Ken Young, for the opportunity to undertake this project and for their efforts with regards to collaboration. We must also thank the staff in the Department of Computer Science, in particular Professor Artur Czumaj, for offering funding which gave us the opportunity to attend the RoboCup competition.

Contents

Chapter 1

Introduction

1.1 Overview

Autonomous robots are devices that perform tasks without human guidance. Advances in computer speeds and reduction of computer sizes have increased the range of tasks that can be performed by such robots in the past few years.

The increase in autonomous robot's capabilities has led to their increased use in military and industrial organisations to perform tasks such as spotting enemies and survivors, carrying out reconnaissance missions and cleaning factory floors.The increase in demand for these robots has led to extensive research in this field in the past few years.

In this project, our group has implemented SLAM (Simultaneous Localisation and Mapping), and Victim Identification algorithms on an autonomous robot built by the Warwick Mobile Robotics Group (part of the Warwick Manufacturing Group), in order to enable the robot to take part in the RoboCup German Open Competition, which was held in Magdeburg, Germany from 15 - 18th April 2010.

1.2 Objectives

The project was supervised by Professor Nasir M Rajpoot and Professor Sarabjot S Anand, specialists in the field of Media Processing and Artificial Intelligence, with the Warwick Mobile Robotics Group (WMR) acting as the customer. In order to satisfy the customer's requirements, the following objectives were proposed:

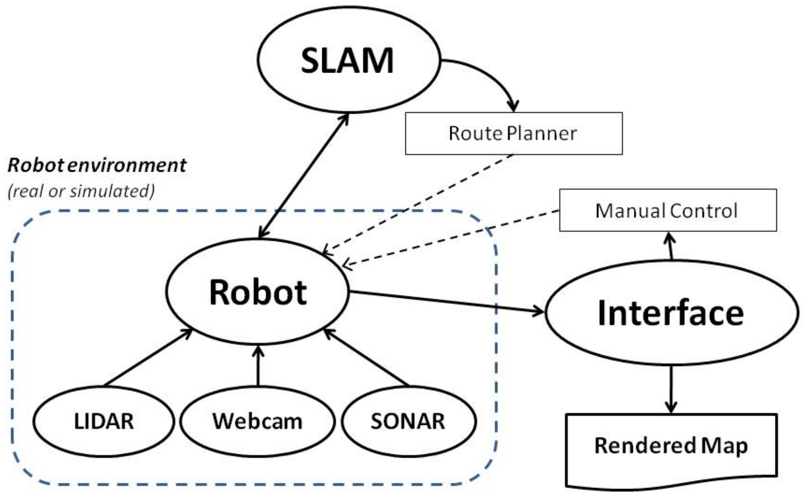


Figure 1.1: The project components.

- Implementation of a SLAM solution for controlling the robot and mapping a simulated disaster environment in the RoboCup German Open.
- Production of an accurate map of the simulated environment in GeoTIFF format for submission in the competition.
- Further development of the existing simulation tool.
- Implementation of a Victim Identification function which is able to: detect CO_2 using CO_2 sensors, detect heat signatures using and infrared camera, detect signboards using shape and colour detection techniques and detect victim faces using face detection algorithms.
- Further development of the existing client interface for controlling the robot.

The above mentioned solution was intended to run on a powerful on board computer on the robot running LINUX OS.

1.3 Schedule

The project was divided into 4 main parts - SLAM, GeoTIFF/Interface, Simulator and Victim Identification.

The project specification and objectives (excluding Victim ID related objectives, which were obtained in the beginning of the second term) were finalised after extensive discussions with the customer on 30/10/2009. As the requirements were fluid, regular meetings were held with the customer. The GeoTIFF Module was completed by 05/12/2009, and work on the Simulator was completed by was 10/01/2010. Once these tasks were completed, the group members working on these tasks went on to work on the client interface and Victim Identification parts of the project, which were fully complete by 15/03/2010. SLAM was completed by 15/03/2010.

A poster was presented to the project supervisors on 10/12/2009. In addition, regular meetings were held with group members, supervisors and customers, to obtain information about progress, requirements and solution techniques. The project timeline can be seen in Figure 1.2.

The RoboCup German Open was held between 15/04/2010 and 18/04/2010, the final presentation was given on 03/05/2010 and the final report was completed by 26/04/2010.

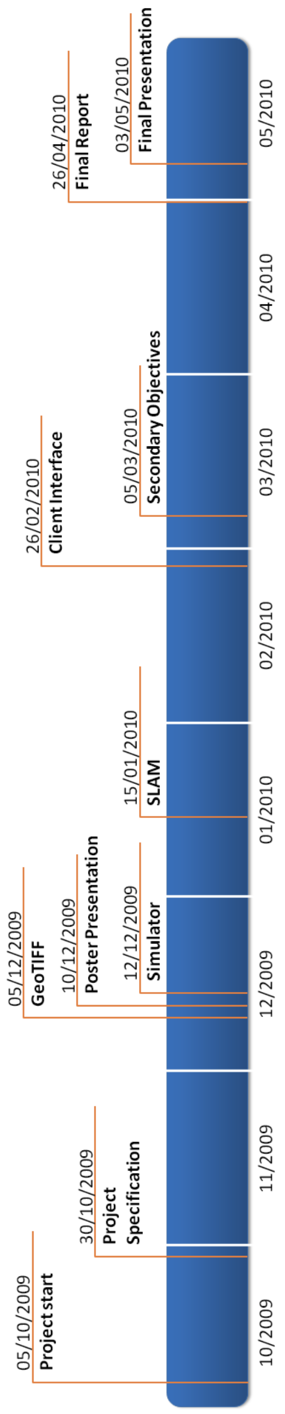


Figure 1.2: The project timeline.

Chapter 2

Project Management

2.1 Overview

The group was formed in June 2009 (Jan Vosecky joined in October 2009), and meetings with the WMR group and the Engineering team were held in June 2009 to obtain a brief overview of the project. Supervisor's approval was obtained once these meetings were held, and the project commenced in October 2009, with the aim of completion by May 2010.

2.2 Methodology

The widely used and recognised iterative and incremental development methodology [20] was used to structure the project's progress. This methodology was chosen over other project management techniques such as the waterfall model because the customer requirements were fluid. The methodology allows for a strong research and design phase which considers all conceivable aspects of the project's implementation and testing. With a strong emphasis on the planning and design stage of the process it was expected that project implementation would go as smoothly as possible, but should requirements change or problems arise the development methodology allows for a redesign to take place.

2.2.1 Requirements Analysis

The first stage was the requirements analysis stage. This stage involved obtaining information about the robot, the competition, the existing software and what was required from our team in terms of making the robot autonomous, and having internal meetings to decide which requirements could be successfully satisfied considering issues such as available resources and the amount of time available for the project.

2.2.2 Design

Once the objectives were laid out, extensive research was done on the various methods that could be used to satisfy the customer's requirements. Following this, extensive research was done on the SLAM process, victim identification algorithms, the GeoTIFF module and Simulator design. The necessary research was broken down into several sub-topics, which was delegated among the different members of the group, based on each members strengths and interests.

After sufficient research work was completed, several group meetings were held, in which the solution was designed. This part of the design stage involved drawing a UML diagram for the code framework and a component diagram which included details about how each component of the project would interact with each other.

It was decided that the solution would be designed in such a way that each component would be abstracted to the maximum level possible, thus keeping in line with good programming practice. It was also decided that each member of the group would work on the component that he researched, so as to prevent redundancy and enable faster completion of the solution.

2.2.3 Implementation/Verification

The implementation stage of the project was divided into five main parts - SLAM implementation, Client Interface Development, Victim Identification, Simulator Development and GeoTIFF conversion.

SLAM

In accordance with WMR's requirements, SLAM was the most crucial part of the solution, as the SLAM algorithm allowed the robot to navigate in an unknown maze, and successfully generate a map. Hence SLAM was considered the core objective, and it was decided that two group members (Jonathan Abbey and James Griffin) would work on the SLAM algorithm for the entire project.

The SLAM solution implemented by our group consists of four parts: Landmark Extraction, Data Association, State Estimation and Route Planning. The tasks of implementing the Landmark Extraction and State Estimation algorithms were assigned to Jonathan Abbey and the State Estimation and Route Planning tasks were assigned to James Griffin.

An agile development methodology was followed for the SLAM implementation.The agile software development method is a method of iterative software development, where requirements and solutions evolve through collaborations between cross functional teams. This strategy was adopted because of the extensive collaboration between the different parts of the SLAM process (i.e, while SLAM can be split into four different parts, these parts were not necessarily modular, with extensive interaction required between the programmers working on SLAM).

Simulator

The Simulator tool essentially mimics the sensor operation, supplying the data to the robot to test the its software, and produces images displaying what the robot saw in the test arena. The main purpose of the Simulator Tool is to act as a testing tool to test the SLAM algorithm, as there were two main limitations in testing the SLAM algorithm on the robot.

The first limitation was that robot runs on lithium polymer batteries, which last for approximately 45 minutes before must be recharged. This essentially means that one can only test algorithms for a short while before the batteries need to be recharged (which could lead extensive delays in the testing process). The other issue was that only one member could test their work on the robot at any given point in time, a limitation that does not present itself when using the simulator.

It was decided that only one member (Jason Cook) work on its development. An iterative strategy was adopted for simulator development, as this would enable functional versions of the tool to be available as soon as possible for use by the SLAM programmers.

GeoTIFF / Interface Development

The GeoTIFF module converts the environment map generated by SLAM to GeoTIFF (Geographic Tagged Image File Format), which is an internationally recognised format for geospatial data representation.

The client interface is a visual tool that displays the final map, along with victim data, and is run on the client PC as opposed to the robot. The interface utilises the GeoTIFF module to convert the map generated by the robot to GeoTIFF format. A command line interface (CLI) can also be used to control the robot manually.

Like the simulator, functional versions of the interface were required as quickly as possible, to make it easier for the SLAM programmers to their SLAM algorithms. Hence, an iterative development strategy was adopted for this part of the project as well.

Victim Identification

Victim Identification consists of five parts:

- Heat signature detection using an infrared camera.
- CO_2 detection using a CO_2 sensor.
- Face detection using a Logitech 3000 USB-Enabled Web Camera.
- Colour Detection to detect signboards using a Logitech Web Camera.
- Shape detection for circles, squares and ellipses, to detect eye charts and holes with victims.

Since these five parts were all modular, it was decided that one member (Jan Vosecky) of the group would implement face detection and ellipse fitting, and another member (Jason Cook) would be in charge of implementing colour detection, heat detection, CO_2 detection.

All these modules would be linked in using a 'Victim Identification' class, which connects these modules with the rest of the solution. It was decided that this class would be implemented by both the members working on the Victim Identification part of the project.

2.3 Division of Responsibilities

Once the majority of the project's research had been completed it was possible to create five main divisions - SLAM, the GeoTIFF module, Victim Identification, Simulator Development and Project Management.

It was decided that each member would be a 'specialist', i.e each member would work on the tasks associated with his division from start to finish. Since SLAM was the primary objective, it was decided that two members would work on this aspect of the project.

The tasks of development of the Simulator and GeoTIFF module were assigned to one member each respectively, and since these were smaller tasks, these members would proceed to work on the Victim Identification aspect of the project once these tasks were completed.

It was mutually agreed that all members working on programming would be involved in linking the different components of the project together, with each member responsible for his component's side of the linking process.

A project manager was appointed, and it was decided that this person would be in charge of organising meetings, monitoring performance and acting as a secretary and spokesperson for the group.

The following lists each member along with their roles in the project:

Project Management	
Adish Ramabadran	Project Manager, Spokesperson for the group, Documentation, Website Management.
SLAM	
James Griffin	SLAM Implementation (Route Planning, Data Association)
Jonathan Abbey	SLAM Implementation (Landmark Extraction, State Estimation)
Victim Identification, Client Interface and Tools	
Jan Vosecky	GeoTIFF Module, Client Interface, Victim Identification (Face Detection, Eye-chart Detection, Hole Detection)
Jason Cook	Simulator Development, Victim Identification (Heat Detection, Colour Detection, Sign Detection)

2.4 Communication

It was understood that a high level of communication was necessary in order to keep the project on track for success, and as such meetings were scheduled to be held twice every week. In addition to this, meetings with the project supervisor were scheduled once a week (depending on availability of both the group members as well as the supervisors) in order to inform them of the project's progress, and obtain suggestions with respect to the various aspects of the solution. Meetings with the Engineering team were held once every week so as to discuss various aspects of the construction of the robot, including the types of sensors required for the best performance at the competition and minimum PC requrements.

As well as these meetings an online forum was created which was used to discuss any ongoing ideas or problems. This forum provided a setting for decisions to be made as a group and gave an opportunity for problems to be raised during the week. An IRC channel was created where members could communicate with each other through the Internet if required. Finally, all members of the group exchanged contact details such as phone number and e-mail in case timely assistance was required.

2.5 Website

In order to advertise our project to potential sponsors for funding purposes, and also to act as a source of information(for external parties) about our solution, the competition and progress about the project, it was decided that a project website was necessary. Wordpress.com was used to create this website, and details about the project were posted on this website.

The website link is as follows: http://warwickcsrobot.wordpress.com/

It was decided that the project manager would be in charge of the content of the website.

2.6 Risk Management

As with almost all projects of this size, several problems were encountered during the course of the project. Some of the major ones are outlined below.

Problem: Delay in Robot commissioning

Since a new robot was being commissioned for the purpose of competing in the autonomous regions of the competition, our team did not have a robot to work with for the majority of the project. This meant that less time was available to port and test the solution on the robot.

Problem: Lack of Well-Defined Competition Specification

The competition rules and environment specifications were not clearly defined upon starting this project, as all such information was obtained from publications from the 2008 competition, and through discussion with the Engineering team. It was judged though that this would be an acceptable risk, because the degree of the changes experienced by the Engineering team the previous year was not too excessive. Some significant change in the 2010 competition was always a possibility however, and in the end this was realised. Our assumption of uniform-sloped floors as in previous years was incorrect, and thus the robot was not suited mechanically for the competition. Furthermore the motion models defined in the EKF were invalid on this new floor, rendering state estimation non-operational.

Problem: Lack of Testing

According to the Engineering team's original plan, the robot was to be completed and available to our group for testing by the beginning of the Christmas break. However, due to unfortunate delays, the robot was only completed by the beginning of the Easter break, nearly 3 months behind schedule.

This meant that very little time was available for testing the solution on the robot, which contributed to the robot's poor performance in the competition.

Problem: Change in Victim Identification Techniques

The competition specification was released by the organisers in small parts over a period of time. Hence, our group was forced to plan victim identification techniques based on what was encountered in the competition last year which did not include IR, but included movement recognition. However, midway through the project, our team had to alter the victim identification to include heat signature detection using the IR camera, which caused a disruption in our project schedule.

This problem was managed easily because of following good design practice. Since a separate function was implemented for each victim identification technique, it was easy to implement and add the heat detection function, and de-link the movement recognition function from the victim identification class.

Problem: Simulator Development Problems

The existing simulator had several bugs and issues that needed fixing, which meant that the task of simulator development was a lot harder than previously thought. A GUI was successfully created for initialising the simulator, and the simulator was refactored to run the SLAM code. Collision detection was successfully implemented, and the overall design of the simulator tool was improved, but due to time constraints, simulator development was stopped in favour of victim identification.

Problem: Slippage when turning the Robot

The autonomous robot uses tracks to move. This caused problems while turning the robot, as depending on the surface, a varying amount of slippage was encountered. This disrupted the SLAM process, and led to the generation of an erroneous map.

This problem could have been easily solved by using wheels instead of tracks, but due to time constraints, the engineering team were unable to install these tracks. Therefore, the problem was solved by using three techniques simultaneously.

The first technique was to use a compass to obtain the bearing of the robot. However, this was error prone too, due to interference in the electromagnetic flux due to the various electronic devices and metal parts inside the robot. The second technique was to turn the robot very slowly to minimise slippage, and add a ‘slippage factor’ depending on the surface, which was used to account for remaining errors in the slippage. The only drawback resulting from the use of the ‘slippage factor’ technique is that it needs to be altered depending on the surface the robot is going to be run on, which might be tedious, and also cause problems when the competition arena involves multiple floor surfaces. Scan matching was the third technique employed.

2.7 Version Control

The existing WMR Subversion repository, hosted by CompSoc, was used during the project to manage the code base. This was particularly important since multiple people were working on the software simultaneously, and Subversion allows for proper handling of conflicts and merges between other people’s work. Furthermore, Trac was used as a front-end for

Subversion which provided basic information about the repository and access to the latest code.

Chapter 3

Robot Specification

3.1 Overview

A fully autonomous robot, by definition, must be able to:

- Gain information about the environment.
- Perform its tasks without continuous human intervention

The robot used for this project was built by the Engineering team in consultation with the CS group. Images of the robot can be seen in Figure 3.1.

3.2 Sensors

The sensors are the tools used by the robot to gain information about the environment. The following sensors were mounted on the robot:

- 1 Hokuyo URG-04LX Laser RangeFinder (LIDAR)
- 1 IR Camera
- 1 USB Enabled Web Camera
- 4 SONAR Sensors
- 1 Oceanserver 5000 Compass

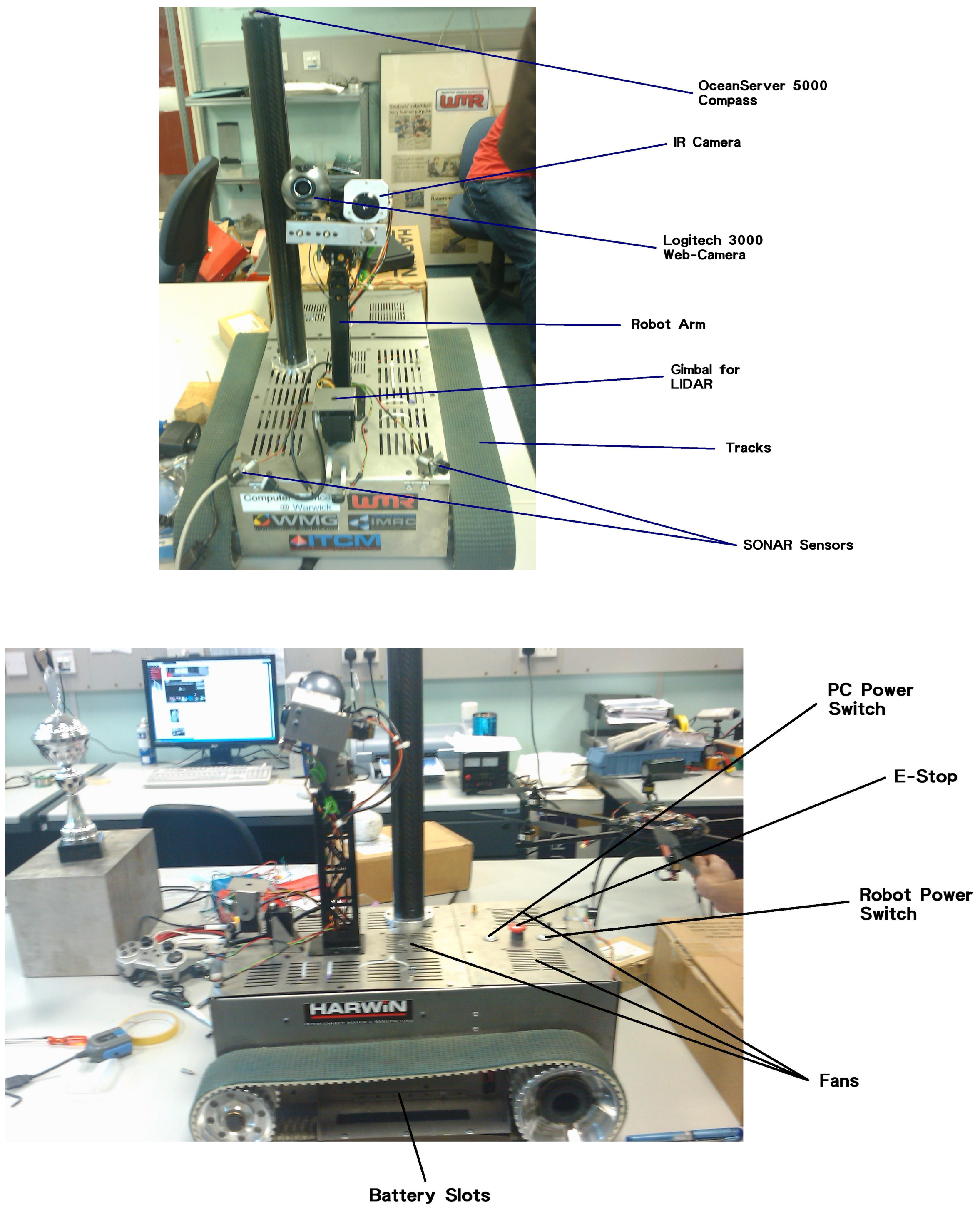


Figure 3.1: The components of the robot

LIDAR

A LIDAR (Light Detection and Ranging) scanner emits light within its scan field, receives the scattered reflection of the light from an obstacle in the environment, and then measures properties of the returned light to determine information about the obstacle. Time of Flight (TOF) systems (as used by the device on our robot) emit a short pulse, measure the return time of the pulse and perform a velocity-time calculation to determine the distance to the obstacle.

The Hokuyo URG-04LX uses light of wavelength 785nm and uses a power source of 5V. It has a scanning angle of 240 degrees and a resolution of 1mm. It is connected to the on-board computer using a USB port, and is located on a gimbal on the top of the robot.

The LIDAR can be considered the most important sensor for the purpose of automation. It is used to gather observations of the environment for use in SLAM.

Advantages:

- Effectively instantaneous scans so no need to compensate for robot motion
- Good range accuracy (< 10 mm)
- Excellent angular resolution (0.25° to 0.5°)
- Can directly interpret data as range and bearing
- Data is dense

Disadvantages:

- Range information limited to a plane
- Expensive
- Some materials are transparent to laser (e.g. glass)
- Affected by brightly lit environments

SONAR

A SONAR (Sound Amplification and Ranging) sensor operates in essentially the same way as a TOF LIDAR scanner, except using a sound pulse instead of a light pulse. However, unlike LIDAR a single SONAR sensor cannot return point data as reflection of the pulse may occur at any bearing within the field of the emitted sound pulse. A single reading will therefore represent an arc in 2D cartesian space. Instead, a common technique is to combine data from multiple SONAR sensors at different positions and compute the intersection of each sensor's range arc (Triangulation Based Fusion [29]).

SONAR sensors were used for collision avoidance. Four SONAR sensors were installed on the robot so as to detect possible collisions on all sides of the robot. The SONAR sensors were connected to the on-board PC using a Phidget Board.

Advantages:

- Low cost
- High range precision ($< 1\%$ error)
- Operation in any brightness condition
- Can directly interpret data as range and bearing
- Disregarding specular reflection & crosstalk, data can be interpreted directly

Disadvantages:

- Weak echoes cause exaggerated range estimates
  - Delay in return signal power reaching detection threshold
- Multiple reflections cause false readings
- Crosstalk between multiple transmitters
- Poor angular resolution ($\approx 25°$)
- Limited range sensing (0.2m to 10m, 0.2m - 5m for small objects)
- Speed of sound limits pulse firing rate

Vision

Cameras make available a much greater variety of information about the environment than sensors such as LIDAR and SONAR, as range sensors cannot gather data about surface properties. Image processing can be used to identify edges, colours, reflections and textures, and stereopsis can be used for depth perception. However vision information content is much less well defined than for range sensing.

The USB Enabled web camera used on the robot was a Logitech Quickcam 3000 USB Enabled Web Camera. This camera has a 640x480 resolution, and can capture images in JPEG, BMP, TIFF, PCX, PSD, PNG, TGA formats.

The web camera was used for detecting holes where victims could be hidden in, face detection for detecting doll faces, symbol detection to detect the 'Tumbling Es' and signboard detection to detect Danger Symbols and the like. The web camera is located on the robot arm, on the same Gimbal as the IR camera.

Advantages:

- Large amount of available information
- Can derive 3D information
- Passive sensing

Disadvantages:

- Very high computational complexity
- Highly affected by lighting of environment
- Expensive

3.2.1 IR Camera

An IR camera forms images using infrared radiation. It works in a similar manner to a standard camera, except that it uses radiation wavelengths belonging to the IR spectrum rather than the visible light spectrum.

The IR camera is primarily used form Victim ID. It can be used to detect differences in temperature between an object and its surroundings due to colour differences in the image formed between hot and cold regions.

The IR camera used on the robot has a 160x120 pixel resolution, and is connected to the on-board PC via a USB TV Capture Card. The IR Camera is located on the robot arm, on a gimbal.

3.2.2 OceanServer 5000 Compass

The Oceanserver 5000 compass was used to obtain an initial bearing, so as to ensure that the map was aligned correctly.

The compass was located on a carbon tower located on top of the chassis so as to minimise interference from the metal and electronic components of the robot.

3.3 On-Board PC

The on-board PC acts as the brain of the robot. It serves a variety of functions, including:

- **Movement**
  - Responsible for sending values to the motor control board, which in turn controls the movement of the robot
  - Controlling the gimbal
- **Mapping**
  - Processes LIDAR data and feeds it into the SLAM algorithm.
  - Runs the SLAM and Route Planning algorithms
  - Checks for collisions using data obtained from the SONAR sensors
- **Victim ID**
  - Processes data obtained from the various sensors, and runs the relevant Victim ID algorithms on the obtained data. The PC then uses the results of the Victim ID algorithms and feeds it to the SLAM/Route Planning algorithms.

- **Communication**
  - The on-board PC is also responsible for processing all communication (through the router) between the robot and the client.

The PC has the following Specifications:

- AMD 3GHz Dual Core Processor with 6 MB Cache.
- 4 GB RAM
- 60 GB SSD
- Linux OS with Java and C installed.

A powerful processor and RAM were required for SLAM and Victim ID algorithms, both of which are performance intensive. A SSD drive was used as SSDs tend to be more robust than conventional HDDs due to the lack of moving parts. In addition, SSDs are faster. The LINUX OS was used as LINUX tends to require less processing power than other OSs like Windows, and this extra processing power can be used by the SLAM/Victim ID algorithms.

In addition, a BUFFALO Single Channel 5GHz 54 MBPS router was used by the robot. This is used to send and receive data from the client application. The router is connected to the PC using a standard ethernet connection, and has a 20m range.

3.4 Movement

The robot uses tracks, as they provide more traction than wheels, which means that the robot is more stable when moving at high speeds. The tracks are moved using aluminium wheel pulleys. Track movement is powered by a 165W Megamotor, which is controlled by an AX3500 Motor Control Board. This board is connected through a serial port to a serial-USB converter which is connected by USB to the on-board PC.

3.5 Power

The robot is powered by a 24V Lithium Polymer 5000 MA battery. DC/DC convertors are used to convert the power provided by the battery to standard component voltage. The battery used is rechargeable.

3.6 Chassis

The chassis is made of 0.9mm stainless steel as it is light, and easy to cut into the necessary shapes.

3.7 Cooling

Three fans are used in the robot to cool the on board PC. Two fans are located on the back lid and one fan on the front lid of the chassis. In addition, several vents are cut into the chassis to allow heat to dissipate out of the robot.

Chapter 4

About RoboCup Rescue

4.1 Introduction

RoboCup Rescue Robot league is a competition for Urban Search and Rescue robots designed to locate survivors in an earthquake damaged building. The test arena design and competition rules are organised by the American National Institute of Science and Technology (NIST). NIST defines standards and test criteria for urban search and rescue robots, to provide guidance to rescue organisations in the US [19]. RoboCup Rescue both allows competitors to test their robots, and allows organisers to test their test standards. As a result of this, though the core requirements of the competition are consistent from year to year, new challenges are regularly added [16].

4.2 Competition Rules

Robots score points by locating victims in certain zones, such as yellow autonomous operation zone, red high mobility zone and so on. In some zones, robots may only obtain points by operating autonomously, while in other zones teleoperation is permitted. Points are also awarded for generating a map of the arena.

Competitions start with all entrants participating in several preliminary rounds. In each round, each team performs a 20 minute 'mission', which involves navigating the maze with their robots and attempting to identify as many victims as possible, for which points are awarded. Teams may run as many robots as they like simultaneously, but only one human operator is allowed to control all these robots.

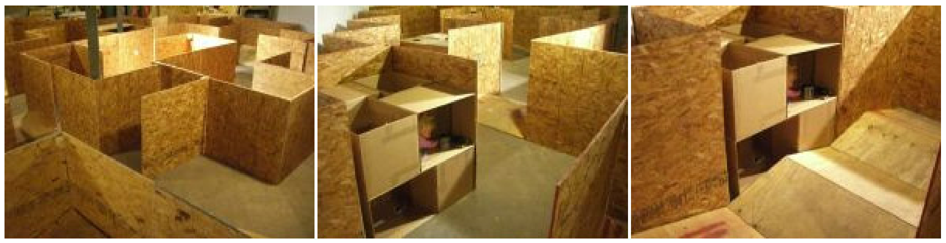


Figure 4.1: The competition arena

Operators may look at the course beforehand, including the positions of victims, but operators cannot look at the course or their robot while the mission is in progress.

After the preliminary rounds, the teams scoring the most number of points are selected to compete in the final round. The final round uses a more detailed scoring system, taking into account the number of signs of life detected (form, motion, heat, sound, CO_2), the detail level provided by the sensors, and the quality of the map produced.

Two additional awards are available; the best in class autonomy award, awarded to the robot producing the best map, and the best in class mobility award, awarded to the most mobile robot.

4.3 Arena Design

Competition test arenas are constructed from 1.2m x 1.2m tiles, with an obstacle on each tile. This obstacle can range from a simple 15° slope to a 45° staircase. Arena sizes vary too, but are usually under 10m x 10m.

Course edges and divisions are created with vertical wooden walls 1.2m high. In some competitions these are smooth and vertical; in others they may slope or have non-smooth features. An image of the arena from a previous year can be seen in Figure 4.1.

4.3.1 Yellow Zone

The yellow area is the simplest to traverse. The floor has 10° slopes throughout, and the area may feature dead ends. The slopes are arranged so that there are no steps up or down, except at the edges of the yellow zone. Victims in the yellow arena may only be identified by autonomous robots and are usually present within boxes. These victims can only be seen through a 15-20 cm diameter hole.

4.3.2 Orange Zone

The orange area of the course presents intermediate difficulty mobility challenges, and can be scored by all robots, be they autonomous or tele-operated. Obstacles in the orange area include 15° slopes with steps, stairs; a 45° carpeted ramp; and half-cubic step fields. Orange area victims are located through 150mm diameter holes, and so directed perception and illumination are required to identify these.

4.3.3 Red Zone

The red area of the course is the most complicated to traverse. It contains full cubic step fields, and victims located through 15-20cm diameter holes.

4.3.4 Blue Zone

The blue area of the course tests robot's manipulators, by providing a number of payloads which can be lifted and moved for additional points. These include wooden cubes with loops on the top, small radios, and plastic 500ml water bottles. The blue area is a recent addition to the test arena and the manipulation tasks carry a score equal to finding two victims.

4.4 Victim Identification

Victims are emulated using the following techniques :

- Dolls to emulate human bodies.
- CO_2 sources to emulate the CO_2 emitted during human respiration
- Heat sources to emulate the heat emitted by the human body.
- Eye charts consisting of 'Tumbling Es'.
- Signboards.

All victims are located inside a box, with a hole 15-20 cms in diameter through which these victims can be identified. A few example victims can be seen in Figure 4.2.

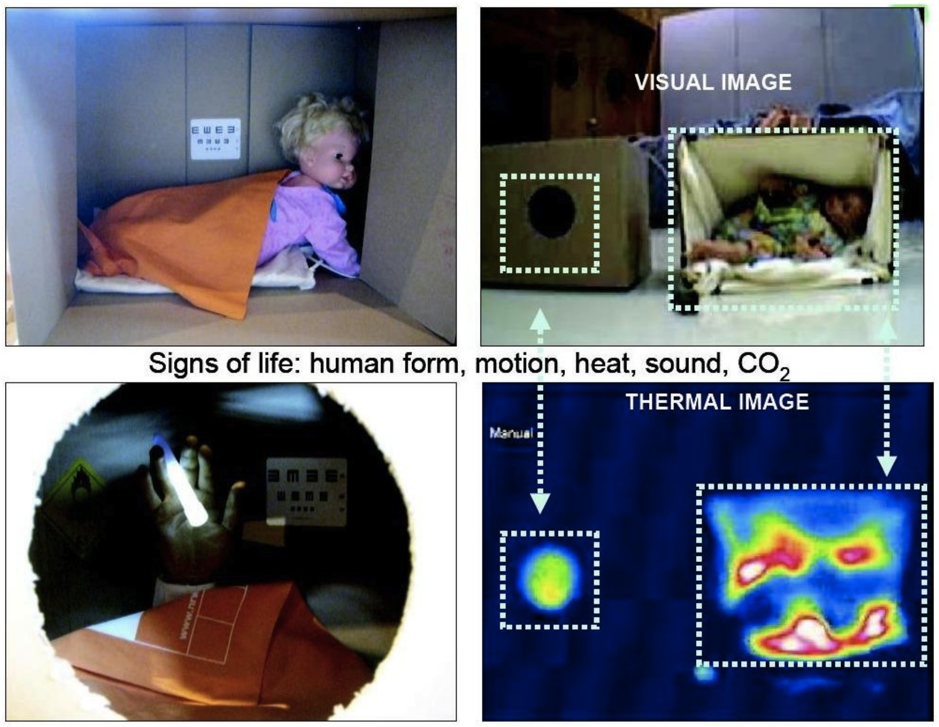


Figure 4.2: Some example victims

Chapter 5

Research

5.1 SLAM

5.1.1 Overview

Simultaneous Localisation and Mapping refers to the combination of the sub-problems of Localisation and Mapping. Separately, these problems are eminently solvable. Together as SLAM however they are yet to be completely solved; it is still a popular topic of research in the field of autonomous mobile robotics. In all cases, the issue is in interpreting data from some agent's sensors (odometric and environmental) appropriately.

Localisation refers to the task of determining accurately an agent's position within a known environment, given perfect environmental sensors. Even with significant levels of noise in odometry data, the absolute confidence in the environmental knowledge means that localisation algorithms can be highly robust. Of course, an algorithm with poor noise handling could still report incorrect and even divergent agent positions. Similarly, sensors that (though perfect) are unsuited to capture sufficient environment information would have the same effect. [11]

Mapping is (in many respects) the counter-problem, whereby the agent attempts to build an accurate map of an unknown environment given perfect odometry sensors. In this case, despite having no prior knowledge of its surroundings the agent can know exactly its pose (relative to its starting point) at all times. A calculated map can therefore be robustly accurate and convergent to the true environment, since it is based on consistently accurate agent pose data. [11]

Of course, solutions to either of these problems require input that would not usually be available in scenarios in which they would be of practical use (a perfect map and perfect odometry respectively). The further difficulty with solutions to SLAM, as may be apparent, is that the required input for either problem is given by a solution to the other. It is therefore a coupled problem: both solutions must be gleaned from the same information. The concept, first introduced by Cheeseman and Smith in 1986 [26], is that of an agent (from now on we will assume a robot) creating a map of an unknown environment whilst simultaneously localising itself within the environment, given some degree of noise in all sensors.

This is achieved through determining a matching between current and past observations. The noisy odometry data (or just dead reckoning) provides an estimate of the robot's current pose (position, orientation, configuration, etc). Expected observations may then be computed, and compared to actual observations to determine a best estimate of the system state. This state which the SLAM algorithm must track consists of the robot pose and a representation of the environment. [11]

Due to the noise associated with both the odometry and sensor data, the system state can easily diverge from the true real-world state by the propagation of these inherent errors. Within the common feature-based approach to the SLAM problem, insights such as a precise formalism of landmark location uncertainty [8] and a correlation between landmark location errors due to motion [27] were key in the development of state estimation methods to handle this uncertainty, such as the widely used Extended Kalman Filter and Particle Filter.

5.1.2 Approaches

A number of approaches exist for developing an overall SLAM solution, differentiated chiefly by the method of map representation employed. The three main options considered for this project were grid-based, feature-based and topological mapping.

Grid-Based Mapping

Grid-based mapping is perhaps the most basic of the three approaches - the environment is represented as a grid of cells, termed a 'certainty grid' or 'occupancy grid'. Each cell is of a fixed size, and is assigned a probability that it is occupied by some object. These probabilities are tracked and updated by the SLAM algorithm in order to localise and map. Such approaches operate directly on the raw scan data rather than processing the data to identify features. [13]

This in turn means that the matching between current and past observations is achieved by a search of some sort through the robot's 'configuration space'. This space's dimensions are the various aspects of the robot's pose (position, orientation, etc). The search algorithm typically uses the odometric pose estimate as a starting point, attempting to find the pose that gives the best overlap between current and past observations (a technique known as 'scan matching'). In this case the matching step also determines directly the difference between the robot poses expected from odometry and observation, for use in estimation of the system state. [22]

Advantages:

- Intuitive approach
- Simple implementation
- Natural extension to higher dimensions

Disadvantages:

- High computational cost of localisation
- High storage requirement
- Weak theoretical foundation as data is smeared:
  - critical information removed
  - inconsistent information added
- Quality and robustness critically dependant on:
  - sensor system
  - grid update mechanism

Feature-Based Mapping

Feature-based mapping adds another stage of processing - rather than working with the raw scan data directly, a feature extraction operation is performed first to identify recognisable landmarks within the environment. Such features might be straight walls, curves, colours, heat blobs, areas of motion, etc, depending on the sensor data available. The system state is represented as a relational map containing many such features. [21]

Of course to operate successfully such features must be sufficiently abundant, readily distinguishable from each other and easily re-observable. If these conditions are not satisfied SLAM will fail simply through lack of environmental input. Association of observations then reduces to a search through the correspondence space of current and past features, in order to find the best pair-wise matching between the two. In contrast with grid-based mapping, the 'correspondence space' association required for feature-based approaches does not provide an odometric vs. environmental pose difference directly. [21]

Advantages:

- Provides metrically accurate navigation robustly
- Capable handling a degree of error in sensor data
- Natural extension to higher dimensions

Disadvantages:

- Increased robustness at some cost to precision
- Environment must contain suitable features

Topological Mapping

Topological mapping offers significantly different results to the other two approaches - the tracked state represents a graph-like description of the environment, rather than a precise metric map. Nodes in the graph are 'significant places' in the observed environment (e.g. rooms) which must be easy to distinguish from each other and the environment as a whole. Edges in the graph correspond to action sequences required to traverse between places. [5]

Advantages:

- Computationally simpler
- Appropriate for simple environments

Disadvantages:

- Does not produce a metrically accurate map
- Unsuccessful in large/complex environments

5.1.3 Chosen Approach

It was decided that a feature-based SLAM algorithm would be developed. Grid-based mapping would likely be too computationally intensive if also performing visual processing for victim identification, and would also be less robust than feature-based mapping. Topological mapping would not produce a metrically accurate map as required, and a hierarchical hybrid system would be very difficult to implement. Feature-based mapping can provide robust production of an accurate map as required, and the regular maze layout of the environment is highly suited to extraction of straight wall or corner features.

Range data would be gathered using the LIDAR scanner, as its high accuracy and point density makes it ideal for wall extraction. Also the wooden construction of the environment presents no problems for the scanner. It was judged that the SONAR sensors would likely be too inaccurate and possibly too limited in effective range to be used for accurate mapping, but would be very useful for collision detection as a backup. It was decided that the potential for visual SLAM would be investigated if time permitted, but that it was an unlikely candidate. The camera was used however for victim identification through appropriate image processing techniques. Range data can be more easily gathered by LIDAR rather than through stereopsis, and the high computational complexity could easily become an issue if performing visual victim identification as well.

Within the chosen feature-based approach to SLAM, four fundamental sub-problems were identified:

- Feature extraction
- Data association
- State estimation
- Route planning

5.2 Feature Extraction

5.2.1 Overview

Feature extraction is the process of identifying landmarks in the environment, for use as a frame of reference for the robot if it re-observes the same landmarks later. As such they must satisfy certain suitability criteria [21]:

1. Features should be easily re-observable by the robot at future time-steps, or more formally they should exhibit a reasonable level of invariance. Consider some geometric feature of the environment such as a straight wall or a corner; we would require it to be detectable from a large area of positions and wide range of orientations, so it would be re-observed after the robot moved. Similarly, features must also be stationary (i.e. time invariant - a crucial assumption in most SLAM approaches). Alternatively, if we were to use some visual feature such as the colour of a floor or wall, then this feature would need to be constant.

2. Features should be unique enough so as to be easily distinguishable from each other, a property that is especially important in environments densely populated with landmarks. This requirement might be readily satisfied in complex (e.g. outdoor) environments, but in more uniform settings where similar features, such as straight walls, are numerous, the task may be more difficult. It is important to note that a combined arrangement of features can effectively be considered a single feature as a group, and depending on the environment these combinations may be enough to satisfy the uniqueness criteria. Further, if the odometry sensors are accurate enough it be argued that only features in the same locality should be easily distinguishable.

3. Features should be abundant in the environment, so as not to leave the robot in an area with too few landmarks to make a meaningful pose estimate. Environments such as large open spaces pose a particular problem because of this, in which case it may be necessary to use floor landmarks or rely solely on odometry.

Clearly then the choice of designated landmarks depends greatly on the environment in which the robot is operating; in the RoboCup Rescue arena for example, straight walls and corners might fulfil the criteria outlined, and they can be identified effectively through various methods including Hough Transformation and RANSAC. In other environments, protrusions from a uniform surface might be suitable features, which could be identified by the Spikes method. In any case it is usually advisable to subject potential landmarks to a validation period, whereby they are not accepted as viable for SLAM until they have been successfully re-observed over multiple time-steps. This should increase confidence in their satisfying the above criteria for landmark suitability. Non-stationary landmarks such as humans for instance should be filtered out by this approach, preventing them from skewing the state estimate. [21]

5.2.2 Approaches

There exist a wide variety of algorithms for extracting straight line features from 2D range data, many of which are general techniques that may be extended to handle other feature models. The following algorithms were considered [18]:

Split-and-Merge

This algorithm initially puts all data points into a single set and fits a line to all the points (using least-squares). It then proceeds recursively. The point in the set furthest away from the line is found, and if the distance is larger than some threshold the point set is split either side of the distant point (along the line), and new lines calculated for the two smaller sets. Once the recursion has finished splitting the points, any computed lines that are close enough to being the same line within some threshold are merged (point sets merged and new line computed).

Incremental

A rather simple algorithm, this loops through all data points in order around the scan field. Initially the first two points in the sweep are taken and a line computed. For each new point a new line is computed from the current set of points and the new point. If the new line meets some condition (i.e. properties match the old line closely enough) then the new point is accepted as part of the current line. Otherwise the old line is stored and a new line is started.

Hough Transform

The key insight with this approach is to consider the line as a point in a space with dimensions representing the line parameters, called a Hough space. The convention is to use the length of the line's normal to the origin and the bearing of the normal, as taking for instance the gradient and intercept fails for vertical (or even just steep) lines.

First, an 'accumulator array' is initialised to represent a bounded discrete approximation of the true Hough space. The algorithm then loops through each scan point and transforms them to curves in Hough space; there are infinitely many possible line orientations through each point and each orientation has only one valid normal length, resulting in a sine curve in Hough space for each scan point. Any accumulator cell lying on this Hough curve is incremented.

Finally, the algorithm loops through the accumulator cells from highest to lowest voted. If the number of votes does not meet some threshold the algorithm terminates. Otherwise the algorithm determines all scan points lying on the cartesian line represented by the Hough point, fits a line to these points (least-squares), stores the line and removes the points. [7]

Line Regression

Making use of the same principle of a search through a transformed parameter space, the line regression algorithm first fits a line (least-squares) to every group of consecutive scan points, the number of points per group being some appropriately set sliding window size. A 'line fidelity array' is then computed, comprising sums of Mahalonobis distances between lines in all groups of 3 adjacent windows - this represents the closeness of the lines. The algorithm then repeatedly searches for adjacent elements in the line fidelity array with values below some threshold, indicating overlapping line segments. If such a pair is found the lines are merged, otherwise the algorithm terminates.

RAndom SAmple Consensus (RANSAC)

The procedure here is designed to ensure robustness despite unwanted outliers in the data, for instance through scanner errors or gaps & protrusions in the environment. A small sample of points is taken within some radius of a central point, chosen at random, and a candidate line is fitted to the sample (least-squares). The set of all points that lie on this candidate line is then computed, and if there are enough such points to meet a defined consensus threshold the line is accepted and the points removed from the pool. This process is repeated until there are too few points to meet the consensus or some iterations limit is reached [21].

Expectation Maximisation (EM)

This is a statistical technique intended to maximise the likelihood of computed line parameters given missing data, the missing data in this case being the assignments of scan points to a potential line. The algorithm first generates new initial line parameters at random. It then repeatedly executes an E-step then an M-step. The E-step computes the likelihoods of each point lying on the current line. The M-step computes new line model parameters based on the updated assignments likelihoods. At each iteration likelihood is guaranteed to increase until convergence. If convergence is achieved or an iterations limit is reached the loop is terminated. If a good enough line has been found it is stored, the assigned points removed and the process repeated, until a trials limit is reached at which point the algorithm terminates [6].

5.2.3 Chosen Approach

Due to the relative simplicity of the algorithm and the high likelihood of offshoots in the LIDAR scan data, it was decided that development of a RANSAC algorithm would be attempted first and depending on the results this choice would then be re-evaluated. The final RANSAC algorithm, which also includes a post-processing stage making use of geometric knowledge of the maze environment, proved to be extremely effective and so the other options were not investigated further.

5.3 Data Association

Data Association is the problem of matching observed landmarks from different scans with each other [21]. Using this re-observation of landmarks, the robot can improve its location estimate, and as such it plays a very important role in the SLAM process. Correctly associating landmarks is paramount, since wrongly associated landmarks can lead to divergence in state estimation, which leads to the localisation estimate becoming increasingly inaccurate.

A number of techniques for Data Association exist, and these can be divided into two main categories: configuration space and correspondence space association. We shall be considering only correspondence space association techniques, since those are most appropriate for our feature extraction based approach to SLAM. Each correspondence space data association technique is designed to produce a `Hypothesis`, which contains a number of pairings. Each feature from the latest set of observations is either paired with an existing feature or it is denoted as being new. Almost all Data Association techniques use statistical validation gating [4], as can be seen in algorithms such as the *Individual Compatibility Nearest Neighbour* (ICNN) algorithm. This algorithm simply pairs each feature to the most compatible feature based upon some measurement method. [17] claims that this method is likely to introduce spurious pairings (incorrect associations) into the hypothesis as a result of a lack of consistency in the hypotheses that ICNN produces. They therefore suggest a batch gating method, where multiple associations are considered statistically simultaneously, known as *Joint Compatibility Branch and Bound* (JCBB).

JCBB is able to produce consistent hypotheses which are robust against spurious pairings by exploiting the fact that the probability of an incorrect association being jointly compatible with all the pairings in a given hypothesis decreases as the number of pairings in the hypothesis increases [10]. To test joint compatibility, the algorithm performs statistical gating on the hypothesis as a whole, as well as testing the individual pairings. A downside to JCBB is that its running time can be exponential, since it performs tree traversal to construct and check hypotheses. In fact, in [17] it was computed to be $O(1.53^m)$, where m is the number of observations. In our case, however, since the number of walls the robot can see at any one time will be relatively small, certainly < 10, the algorithm's complexity should not pose much of a computation problem. The JCBB algorithm can be seen in Algorithm 1.

Traditionally, the measurement method uses the Mahalanobis distance, which is able to identify and analyse statistical patterns between variables. Unlike Euclidean distance, it is scale invariant, and so is a useful method for determining the similarity of one data set to another. However, [21] suggests that for RANSAC features, as we will be using in this project, the Euclidean distance is a more appropriate measure due to the separated and relatively sparse nature of features in the environment. This also saves on computation, and thus makes the JCBB algorithm more efficient.

Algorithm 1 JCBB Algorithm

```
Require: ExistingFeatures $F_e$, ObservedFeatures $F_o$
  Best = []
  JCBB( [], 1 )
  return Best

  {Find pairings for feature $F_o^i$}
  Function JCBB( H, i )
  if i > ObservedFeatures.size() then
    if pairings(H) > pairings(Best) then
      Best = H
    end if
  else
    for j = 1 to ExistingFeatures.size() do
      if individual_compatibility(i,j) and joint_compatibility(H,i,j) then
        {Pairing $(F_o^i, F_e^j)$ accepted}
        JCBB([H j], i + 1)
      end if
    end for
    if pairings(H) + ObservedFeatures.size() - i > pairings(Best) then
      {Perhaps we can do better. Do not pair $F_o^i$}
      JCBB([H 0], i+1)
    end if
  end if
```

5.4 State Estimation

5.4.1 Overview

State estimation is perhaps the most crucial capability of any effective SLAM system, because it is here that all the accumulated uncertainty from sensor noise is taken into account and in some way managed. Without a robust state estimation method in place computed states (robot pose and landmark positions) quickly diverge from the true state. The key feature of whichever method is chosen is that it should give the optimal estimates (or close to it depending on efficiency considerations) for the Localisation and Mapping sub-problems simultaneously, based on the most recent estimates, sensor data and confidence in each of these. For typical applications with a range sensor and odometry sensor, state estimation is tasked essentially with shifting the odometry estimates by some proportion of the calculated 'innovation', which is the difference between the odometry and range sensor estimates. [11]

There are a number of different methods for computing an optimal state estimate, and the following selection were considered for this project.

Extended Kalman Filter (EKF)

The Extended Kalman Filter is the non-linear version of the Kalman Filter, a linear estimator designed to compute optimal estimates for the variables of a linear dynamic system, given measurements from multiple noisy (zero-mean Gaussian) sources. The term optimal here refers to minimising the overall least squares error. [28]

The KF achieves this by maintaining the covariance matrix of the system, which contains the variance (error) of individual elements in the system state, and the covariance (error correlation) between all pairs of elements. One of the key understandings that furthered the SLAM field was that there exists correlation between error of individual feature estimates due to the motion of the robot that observed them. [21]

The EKF allows for the same techniques to be applied to non-linear systems (provided they are not too non-linear), by taking Jacobians (matrix of partial derivatives) of the various models required for a Kalman Filter implementation, instead of standard differentiation. Essentially, the EKF linearises the non-linear system around the current system state estimate. The capability of the EKF formulation to exploit known error correlation between features and to handle non-linear systems led to its widespread adoption as the standard state estimation technique in SLAM for many years. [28]

One common issue with the EKF is poor computational complexity when scaling to large numbers of features due to the ballooning number of required feature correlations. Another issue (common to all SLAM approaches) is the difficulty of loop closing: if the robot travels in a long loop around an obstacle and returns to a previously visited area, prolonged occlusion of familiar features may lead to enough error propagation to cause the robot to not recognise the area as having been previously visited. [11]

Unscented Kalman Filter (UKF)

The Kalman Filter has in fact seen a wide array of variations and extensions, many of which are used in the SLAM field. Another of these is the Unscented Kalman Filter. This filter again considers the system state elements as Gaussian random variables, but represents them instead with a few chosen sample points (the 'Unscented Transformation') that completely capture their true mean and covariance. These may then be propagated through the non-linear system whilst still maintaining the mean and covariance accurately to the 3rd order, rather than the 1st order achieved by the EKF (linearisation by Jacobians). [12], [11]

Particle filter

The previous techniques considered each assume that all random variables and noise sources are Gaussian. In many non-linear real world scenarios this assumption does not hold, with the existence of multi-model distributions, non-Gaussian noise, etc. Particle filters are sequential Monte Carlo methods that do not make this assumption and are therefore suitable for a wider range of real world problems. Monte Carlo methods repeatedly take random samples from a system to compute approximations of statistics about the system. Particle filters track multiple possible robot paths/poses sampled probabilistically, in order to give an approximation of the optimal least squares solution. [12]

5.4.2 Chosen Approach

All effective non-linear state estimation methods employ complex mathematical and statistical techniques to achieve their goal, and a good understanding of their operation is essential to developing an implementation tailored to this project. Therefore it was concluded that the relative complexity of the various techniques would play a significant factor in our choice of estimator, purely to ensure we had the ability to learn and implement the system in the available time. Availability of documentation detailing the operation of the techniques was also an important factor.

It was decided therefore that an Extended Kalman Filter would be developed. The EKF has been tried and tested over many years and much explanatory documentation is available. Furthermore, it appeared to be the least complex of the methods considered. A judgement was made that the EKF's limitations with large numbers of features would not become an issue due to the nature of the maze environment, and also that the fairly uniform indoor maze environment likely shouldn't be too non-linear for the EKF to handle. Simply from the point of view of accurately modelling the system, the environment did not appear to be too complex for this to be achievable.

5.5 Route Planning

An essential part of the autonomous robot is its ability to deduce and traverse a path from its current location to some other location, so that it may explore more of its environment. In order to construct a path between two points, the robot requires a representation of its environment that can be treated as a graph; an occupancy grid being the standard method used in robotics. This grid allows the robot to query the state of points in the environment; for example a point may contain an obstacle or it may be empty. Using this graph-esque representation, the robot can use pathfinding techniques to construct a path from its current location to some goal point. The most commonly used algorithm for this purpose is the A* Graph Search algorithm. This algorithm is an extension of the classic Dijkstra's algorithm that obtains performance improvements through the use of heuristics. The A* algorithm can be seen in Algorithm 2.

The difficulty then becomes deciding which point in the map to navigate to next. With a lack of literature in this area, it was left to experimentation to find a suitable technique. This problem shall be discussed in more detail in a later section.

Algorithm 2 A* Pathfinding Algorithm

```
Require: OccupancyGrid $grid$, Point $startingPoint$, Point $goalPoint$
  PriorityQueue $open \leftarrow [startingPoint]$
  PriorityQueue $closed \leftarrow []$
  while $open$.size() $> 0$ AND $!foundGoal$ do
    $current \leftarrow open$.poll()
    for all Points $point$ neighbouring $current$ do
      if $point = goalPoint$ then
        $foundGoal \leftarrow$ TRUE
      end if
      if !$closed$.contains($current$) then
        if $open$.contains($current$) then
          Update heuristic cost if now cheaper
        else
          $open$.add($current$, heuristicCost($current$))
        end if
      end if
    end for
  end while
```

5.6 Victim Identification

5.6.1 Viola Jones Object Detection

The Viola Jones technique uses 'Haar Features' (as they bear resemblance to Haar Basis functions) to detect objects. A feature is a rectangle within an image made up of black and white areas.

The Viola Jones technique originally used 3 kinds of features:

1. Two rectangles of equal size either horizontally or vertically aligned.
2. Three rectangles of equal size with the two outside rectangles coloured.
3. Four equally sized rectangles arranged in a two by two formation with the diagonally aligned squares having the same colour.

Since the features can be of any size within the image window, the number of possible features is large. For example, in a 24x14 pixel window, the number of possible features is 180,000. Therefore, it is inefficient to evaluate all the possible features, and hence, the AdaBoost algorithm is used to select the best features.

A feature 'value' is then calculated for each rectangle, which is essentially the difference between the sum of all the white pixels and the sum of all the black pixels within the feature. This feature value is used for detecting objects.

Using Features to Detect Objects

Since every image has several thousand features, the idea is to find the features that best 'describe' the object that we need to detect. In order to do so, we need to 'train' the classifiers using large sets of training data (usually images). Training data is of two types - positive and negative. Positive training data contains the objects that need to be found, and negative training data does not contain the objects that need to be found. The training sets are used by the Viola Jones algorithm to select features that best classify the required objects. Increasing the number of features will increase algorithm accuracy at the cost of speed.

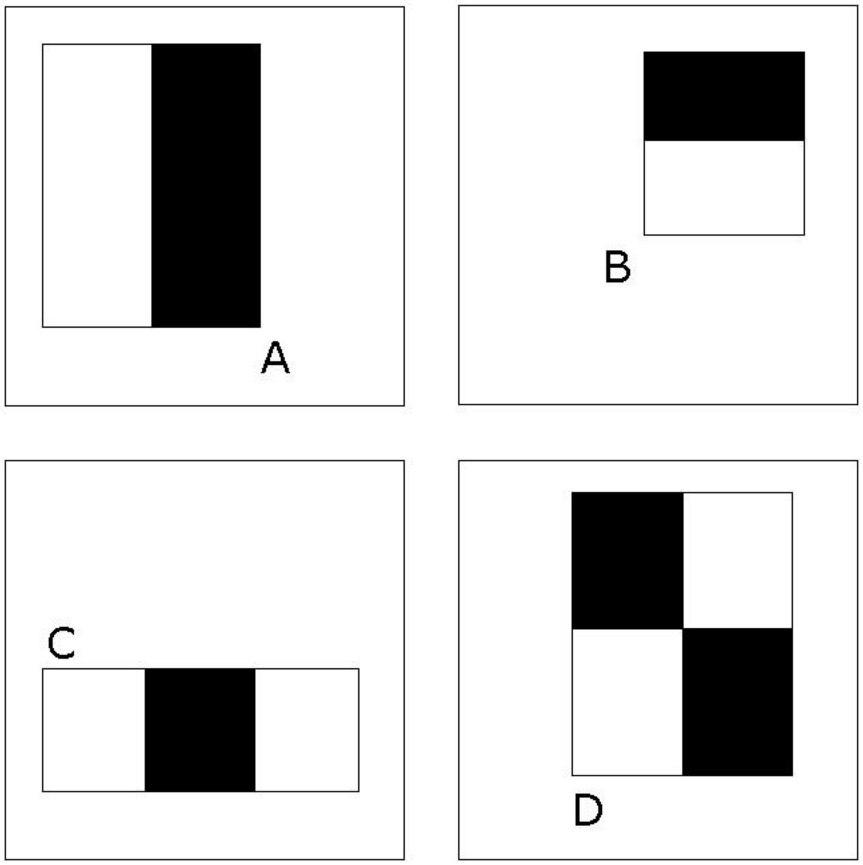


Figure 5.1: Haar Features.

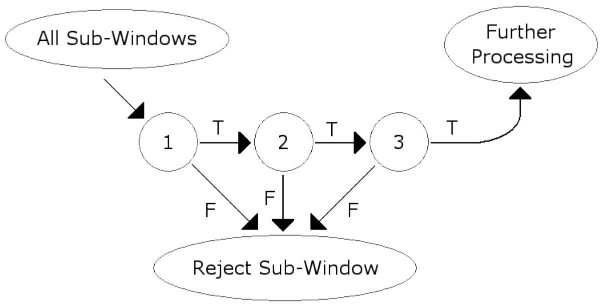


Figure 5.2: Attenuation Cascade

AdaBoost Learning Algorithm

Viola Jones used a variant of the AdaBoost learning algorithm to select the best features for object detection and to train the cascade of these features to improve detection.

The AdaBoost algorithm is a stepwise classification algorithm which starts off by selecting a set of weak classifiers, which are combined to create a stronger classifier by assigning larger weightings to good classifiers and smaller weightings to bad classifiers.

For each feature an optimum threshold classification function is calculated that best classifies the images. This involves calculating a value that best separates the negative and positive images using that feature's value alone. If the feature's value is above this value in an image then the image it is classified in one class, if it is below this value then it is classified in the other.

Attenuation Cascade

In order to improve the performance of the algorithm, a cascade of classifiers known as the attenuation cascade is used which ensures that only those examples that pass through the weak classifiers are tested on strong classifiers.

Conclusion

The Viola Jones technique is faster than other techniques used for object detection. This makes it the best choice for real time object detection. The algorithm is implemented in the OpenCV library as `cvHaarDetectObjects()`, and this was used in the face detection, ellipse detection and symbol detection algorithms.

5.6.2 Detection of Eye-Charts and Holes in Boxes

In order to increase the chances of successful victim identification, it was considered that victims may be placed inside closed cardboard boxes, with only a round hole available, through which the robot may view the victim. In such a case, the robot would initially only observe a hole in the box, after which it should move nearer and explore the inside of the box.

As well as detecting holes, we have investigated methods for detecting eye-charts. Such eye-charts with tumbling E's were to be placed on the wall behind the victim, according to the arena specification. Successful eye-chart detection would provide more evidence for the presence of a victim.

Both of these problems were tackled using the camera and image processing functions. Since these problems can be seen as structural analysis problems in image processing, they are described together in this section.

It has been found that ellipse fitting in image processing could provide means to accomplish hole and eye-chart detection. Ellipse fitting is a method of fitting an ellipse around a set of points. Commonly, ellipses are fitted to a set of at least five points and the best-fitting ellipse is chosen using the least squares method [9]. In the least squares method, the sum of squares of distances between the points and an ellipse should be minimal. Other ways of choosing the best-fitting ellipse exist as well, as described in [23].

In the case of hole detection, it is assumed that the hole will have a circular shape and that the inside of the hole will be darker than the outside, due to the difference in light. Ellipse fitting could then be applied in combination with contour detection. The reason for using ellipse fitting rather than circle fitting lies in the fact that the camera may be in any angle relative to the hole, which could make the hole appear as an ellipse from the camera's point of view. With regards to eye-chart detection, ellipse fitting could help in identifying individual 'E' symbols on the eye-chart. It is assumed that each symbol is printed in dark colour and placed on a white background, providing enough contrast for contour detection. Individual symbols identified in the image could then provide a basis for further processing, such as geometric checks, in order to confirm a specific structure corresponding to an eye-chart.

In this project, we have implemented ellipse fitting using the Open Source Computer Vision (OpenCV) library. OpenCV provides implementations of common computer vision and image processing functions and algorithms. Main functions utilised for hole and eye-chart detection were functions for creating a threshold image, finding contours and applying ellipse fitting. The high-level flow of the basic ellipse fitting algorithm implemented with OpenCV is a follows:

1. Given an input image, convert it to greyscale.
2. Create a binary image based on a threshold on the amount of black colour.
3. Apply a function to find contours. Contours are returned as sets of points, corresponding to each contour.
4. For each contour, if the number of contour points is greater than the minimum, apply ellipse fitting and record any ellipse found.

As mentioned earlier, ellipse fitting provides a basis for further custom processing used in our detection functions to enable eye-chart and hole detection.

5.6.3 Other Image Processing Methods Researched

Colour histograms: In connection with eye-chart and hole detection, a need arise to analyse the amount of a particular colour, in this case black or white, in specific regions of interest. This analysis would provide useful evidence to support or reject a particular detection result. A colour histogram can be used to show the absolute amounts of colours in an image. OpenCV provides standard functions for calculating histograms, which were utilised in our functions.

Hand detection: During the research phase, several other image processing techniques were investigated, such as hand detection. Work on hand detection using OpenCV exists, which uses colour abstracts [14]. However, it was found that further development and refinement of this method would be necessary, as well as overcoming the problem of differences in lighting and colour of the hand. Other methods of hand detection exist, which use more advanced mathematics, whose implementation lay outside the scope of this project. Nevertheless, hand detection would be an interesting addition to victim identification in the future.

5.7 GeoTIFF

The Geographic Tagged Image File Format (GeoTIFF) [1] is a universal format for representing geographical information, such as maps, satellite imagery and elevation models. It is based on the public domain Tagged Image File Format 6.0 (TIFF) [3] for its wide use and support across platforms. GeoTIFF files may be displayed on any system capable of reading the TIFF raster data format. The first GeoTIFF specification has been developed at the NASA Jet Propulsion Laboratory and improved by a collaboration of more than 160 organisations worldwide [24].

The basic building blocks of a GeoTIFF file consist of metadata tags, which can be used to represent coordinates, geometrical shapes and transformations, as well as additional metadata about the content. To comply with the TIFF 6.0 format, standard TIFF file headers and tags are used. The GeoTIFF specification provides tags for describing most cartographic information. A standard TIFF file has the following structure: (i) File header, (ii) Image File Directories and (iii) Data. The Image File Directories (IFD) contain tags codes and pointers to the data. In GeoTIFF, the set of supported tags includes tags for geospatial representation. The so-called GeoTag approach is used to encode all necessary information into a small set of tags, while staying compliant with all standard TIFF tags. These include projection types, coordinate systems, ellipsoids and other geo-referencing information, defined by the European Petroleum Survey Group (ESPG).

GeoTIFF is currently used by a number of Geographical Information Systems (GIS) and image processing software packages. The GeoTools open source toolkit is a Java toolkit for geospatial data representation and manipulation and provides reusable libraries for developers. GeoTools can be used to export maps into the GeoTIFF format. In the process of creating a GeoTIFF file, GeoTools makes use of the open source GeoAPI library, which follows the ISO 19100 standard for referencing geospatial data.

In order to export a map into the GeoTIFF format, internal image representation of the map needs to be created. This step required research into the internal representation of images in Java, such as the components an image object consists of and creating a custom sample model and colour model of the image. This need was mainly due to the task of creating an RGB colour image from a single numeric value for each pixel. With a custom colour model, the colours of the image could be assigned based on a single pixel value, as opposed to 3 channels as in typical RGB images. This way of image creation and manipulation can be achieved with Java Advanced Imaging (JAI).

In summary, the GeoTools, GeoAPI and JAI libraries can be employed to create maps which are then exported into the GeoTIFF format.

5.8 Signboard Detection

5.8.1 Introduction

The problem of square detection in an image needed to be solved and implemented directly because of the competition specification. The specification and the engineering teams past experience explains that with each victim will be a collection of various coloured square signs, for example warning signs and hazard signs. So to detect a victim be it would be beneficial to have a way to detect these signs to give a clearer picture of whether a victim has been identified or not.

There are various ways to go about detecting these signs; a detection system could be implemented which recognises when we look at a region of interest which has a different colour from its surroundings, however this would be too general and give too many false positives as it is impossible to be 100% sure on the arena layout for the competition. This method also seemed too tailored for the competition which is something that needs to be avoided. Another possible approach would be to implement an image classification system for these signs. However there were problems with this method as well because there was not nearly enough specific training images from the competition to build a reasonable classifier and it would be a huge risk if the competition organisers changed their signs only slightly. Although there are ways to mitigate both of these factors, again the system would have been too tailored for the competition and would only work in a quite specific instance.

5.8.2 Signboard Detection Process

The method that was opted for in the end should give us a good general and specific way of detecting these signs. The method implements a square detection algorithm using edge detection and top of this there needed to be a way to narrow down detected squares to the interesting ones, so using colour histograms each detected squares can be compared to a sequence of images represent signs that are known to be situated near a victim.

This method gives a general detection method of square detection using edge detection (though of course it will be tailored as much as possible to 'interesting' squares) and then using colour histogram comparisons to narrow further down. Following is the description of the rough layout of this process:

Signboard Detection Algorithm

Algorithm 3 Find Squares

```
Require: image
  Apply down-sampling and then up-sampling steps of Gaussian pyramid decomposition
  for each colour plane of the image do
    Convert to greyscale For each threshold level
    Apply canny algorithm for edge detection and use dilation to filter out holes between
    segments
    Threshold into a binary image
    Retrieve contours
    For each contour Approximate each contour to a polygon
    if polygon is a square then
      Store it
    end if
  end for
  for each square do
    Get colour histogram of square Compare histogram to array of pre-calculate histograms
    Return true or false based on threshold
  end for
```

This is a very rough outline of the general algorithm that research has suggested and following is a brief description of the separate processes in the algorithm.

The FindSquares function will take an image as its input and return whether or not it has detected any squares of interest. The first thing that should be done is to filter out noise in the image to make it easier for the edge detection algorithms to work; this can be performed using Gaussian pyramid decomposition. The Gaussian filter applied to the image in the down-sampling and up-sampling steps looks for each pixel at its neighbours in a fixed window (usually 5x5) and averages their values. This applies a blur to the image which is useful in getting rid of noise. The down-sampling step then removes all even rows and columns to create a image that is half its size. The image is then sampled up to its original size by adding in even rows and columns full of zeros and the Gaussian filter then reapplied.

The next step would be to separate the colour planes so each plane can be converted to a greyscale image and have edge detection applied to it. There are two main methods of edge detection are best in this situation, these are Canny and thresholding. Thresholding is a very simple method which takes a greyscale image and turns it into a binary image (i.e. black and white), then each pixel goes through a test and if the pixels value is greater than the threshold it is set to black, otherwise it is set to white. Setting the threshold level is the trickiest part of this method but a way to overcome this is to use varying thresholds and perform the rest of the square detection algorithm on each differently thresholded image.

Another way to produce the binary image needed for the rest of the algorithm is to use Canny.The Canny algorithm was developed by John F. Canny in 1986 and will be particularly useful in detecting squares with gradient shading.The Canny edge detection algorithm consists of 3 general steps; pre-processing to reduce noise, edge detection using Sobel masks, and edge refinement using non-maxima suppression and hysteresis thresholding.

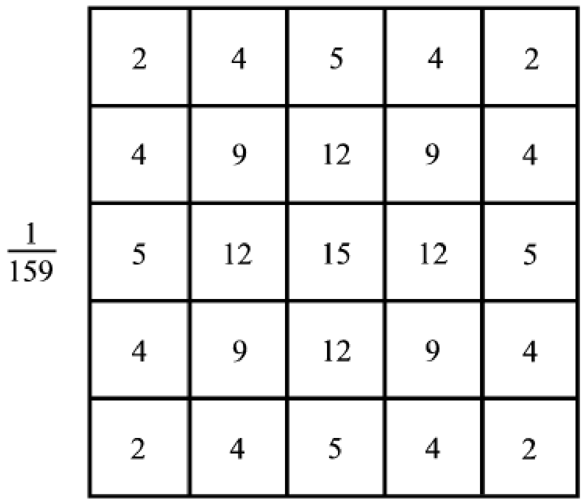
5.8.3 Preprocessing

The image is blurred with a Gaussian filter. This is done by applying the mask (5.3) to each pixel in turn. For each pixel a new value is given by the weighted average of the intensity values of all its neighbours in a 5x5 window (using the weights indicated by the mask). Since the Canny edge detector is based on gradients, the effect of Gaussian blurring is to remove noise that would result in the detection of false edges.

5.8.4 Edge Detection

The image is transformed into a binary image that consists only of detected edges (white in intensity). This is done by applying further convolution masks on a per-pixel basis; this time, the Sobel operators are applied to estimate the gradient of a pixel in the x and y-directions.

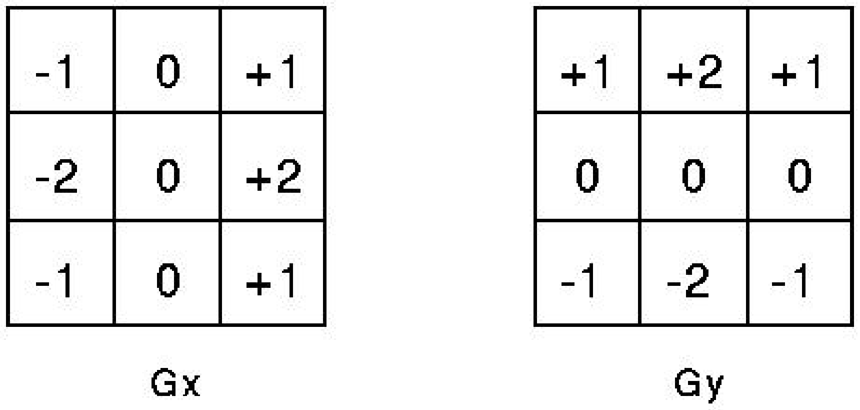
Applying the Sobel masks to a pixel provides Gx and Gy values; from these, the magnitude of edge strength at that pixel is measured by $\sqrt{Gx^2 + Gy^2}$, and the edge direction is given by $\theta = atan(Gy/Gx)$. To approximate the directions an edge within an image can take, each edge direction value is clipped to one of the four possible categories (i.e. hori-



$\frac{1}{159}$

2	4	5	4	2
4	9	12	9	4
5	12	15	12	5
4	9	12	9	4
2	4	5	4	2

Figure 5.3: 5x5 Gaussian convolution mask



-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 5.4: 3x3 Sobel masks for x-direction gradient estimation (left) and y-direction gradient estimation (right).

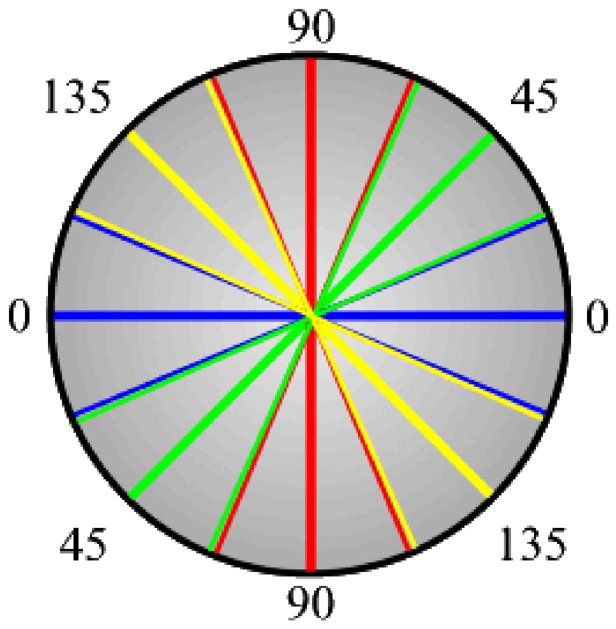


Figure 5.5: An edge direction falling within the yellow range is set to 0 degrees; within the green range, to 45 degrees; within the blue range, to 90 degrees, and within the red range, to 135 degrees.

zontal, vertical, or one of the two diagonals). The clipping ranges are illustrated in Figure 5.5.

At this point, having applied the Sobel operators and not used edge strength and direction information, potential edges may already be identified as those pixels with strong gradients. However, these edges are likely to vary widely in thickness and continuity. Simply thresholding the gradient magnitude does not solve this problem; weak but genuine edges would be discarded and strong pixels that make edges unattractively thick would remain.

The Canny algorithm intelligently refines edges using non-maxima suppression and hysteresis thresholding. Non-maxima suppression consists of convolving a 3x3 mask over the image, which results in retaining only those pixels that represent local maxima in gradient magnitude. This removes many irrelevant edge-pixels.

Secondly, edges are thinned using hysteresis thresholding. This involves tracing individual edges: first, 'starting' pixels of strongest edge magnitude are identified (using a high threshold), then their edge direction is followed to find adjacent, similarly strong pixels. Whilst tracing an edge, weak neighbours that represent parallel edges are discarded. This leaves edges uniformly 1-pixel wide in width. So as not to disregard the tracing of edges that are lesser in strength but equally important, a low threshold is used to find low starting pixels. After a binary image has been created by Canny there needs to be applied further dilation to remove edges that might still exist between edge segments. Dilation is just a method of growing areas of foreground (white) pixels which would close any small gaps in edges. Once the greyscale image has been converted to binary using either Canny or standard thresholding there must be found the contours within the image, each contour being a chain of vertical or horizontal or diagonal segments, represented by their end points. Using these contours we can then approximate them into polygons. A well known and popular algorithm for this is the Douglas-Peucker algorithm which converts curves composed of line segments to similar curves that are made up of fewer points. The algorithm can be talked of simply as a recursive 'divide and merge' algorithm and its pseudo code is as follows.

The algorithm should then check whether the polygon approximated by the above algorithm is a square or not. This can be done by checking that each contour should only have 4 vertices, the contour is convex and the internal angles are all around 90 degrees.

There will now be a list of squares the algorithm can continue checking whether or not each square is 'interesting' or not. To do this there needs to be a computed array of colour histograms from pre-selected images. A colour histogram is a representation of the number of pixels in the image that have colours in a list of colour ranges. The ranges must span the entire colour space of the image. Therefore the histogram represents the distribution of colour in the image and gives us a good way of sorting between the squares that might be the signs being looked for, or just random squares.

So for each square detected, the region defined by this square in the image is used to calculate its colour histogram, compare it to the list of stored histograms and generate a value which depending on whether its greater than a threshold signifies true or false.

5.8.5 Infrared Blob Detection

One of the main parts of victim identification has been foreseen to be heat blob detection which will enable the robot to detect victim locations from range. A blob is simply a collection of white pixels where each white pixel represents an area of heat. To calculate blobs from an image, there are two stages. The first stage is to convert the data from the infrared camera into a binary image of black and white pixels where white pixels represent heat over a certain threshold. The second stage is to group together these white pixels using a suitable algorithm.

The image into the blob detection algorithm will be a RGB image which will be converted into a binary image. To do this each pixel in turn can be to checked to see whether or not it passes a threshold. If it does then set the corresponding pixel in the output binary to white, if not set it to black.

The blob detection algorithm was presented by last years Engineering team and is a simple scan-line type algorithm which needs to be modified. The algorithm can be said to be based on a recursive flood fill algorithm frequently used in paint programs to bucket fill an area. The algorithm is described in 4 and is applied to each pixel in the image.

Algorithm 4 Classify Blobs Scanline

```
Require: pixel
  if already part of blob or not white then
    Exit
  end if
  Add this pixel to new blob
  Add all connected white pixels to the right and left of current pixel to blob unless they
  are already in a blob
  for each pixel in current blob do
    ClassifyBlobsScanline(pixelAbove)
    ClassifyBlobsScanline(pixelBelow)
  end for
```

This algorithm will detect all blobs in the image which can be thresholded so only return them if they are above a certain size, we can them order them by size so that the biggest blob is given the most priority by the robot in calculating direction.

5.8.6 Simulator Tool

The simulator tool is a Java program created by the previous year's Engineering group for their robot. It is designed so that the robot software can be tested independently of the hardware so that some work can be carried out with a certain amount of testing.

The simulator tool can be broken down into 4 parts:

- The first part is the group of classes describing the state of the robot within the simulator; these classes hold information such as robot position and orientation and update this information using a timer.

- The second part, called the datastore, gives access to the simulator's sensors for the robot's automated software.
- The third part is the simulated sensors, these classes try to mimic what the actual physical sensors would return.
- The fourth and final part of the simulator contains the output and interface classes that the end user interacts with. These include classes to show maps outputting the robots movements and a interface to set up the simulator.

The simulator works by starting up several different threads, the majority of these threads are simulated sensors which in the background update the datastore with their information. The sensors use an internal map to create their sensory information; this map is supplied to the simulator on start-up. The map works by using colour as a height signifier. This means that white is equal to ground level height whilst black denotes the maximum height. Therefore the most common map uses black lines to denote walls in a maze.The simulator also starts a orientation thread which polls the datastore for the robots position and updates it based on the current velocity supplied to it by the simulated motor encoder. The last thread started is the actual AI thread which is supplied by the simulator.

The simulator as is works fine with last year's robot but needs to be updated and refined to be integrated with our SLAM system. This means updating a lot of the access and infrastructure of the simulator.

Chapter 6

SLAM

6.1 Implementation

6.1.1 Framework

Before any work began on developing implementations for the various parts of the system, it was decided that a code framework should be established. This was in order to provide a clearer picture of the required areas of development, and ensure interoperability between system components from the ground up.

As the framework was implemented it also came to encompass a range of supporting classes used throughout the system. These include the `Point2D` class, used to store a point in 2D space in either cartesian or polar form, with methods for easy conversion as required across the system. Another important support class is the `Line2D` class, which stores a line in 2D space represented by its equation in general form, and also provides methods for line fitting including a linear-least-squares implementation.

The code framework consists of a range of parameterised interfaces and abstract classes organised in four main sections:

- Sensory
- Feature Extraction
- Data Association
- State Estimation

Sensory

The sensory framework starts with an abstract `Sensor` class, from which all sensor implementations are extended. A sensor contains a `Device instance` which handles the interaction with the underlying physical hardware, and `Sensor` provides access to readings from this `Device`.

Every `Sensor` extension has an associated `Reading` extension. The abstract `Reading` class serves simply as a container for output from the sensor.

The next layer in the sensory framework is the abstract `SensorSet` class. This class is a container for multiple sensors of arbitrary types, designed to enable aggregation of related sensor readings or sensory fusion. It is also intended as the layer at which raw sensor data is processed into a form usable throughout the rest of the system.

Analogous to `Sensors` and `Readings`, every `SensorSet` extension has an associated `SensorData` extension. The `SensorData` class is a container for data processed from combined sensor readings.

At each of these levels a further specification is made for internal and external variants of each of the sensory framework classes. This was done to enable differentiation between odometric and environmental sensors in the other sections of the code framework.

Feature Extraction

The `Feature` interface is just a placeholder for representations of features in the environment. The abstract `FeatureMap` class provides a generic container for storing a map of such features. The `FeatureExtractor` interface is defines the implementation of an object class with a method to extract feature from an instance of `ExternalSensorData`.

Data Association

The `DataAssociator` interface defines the implementation of an object class with a method to produce some form of matching between new observations and the current map. This is further refined into the `ConfigSpaceAssoc` and `CorrespSpaceAssoc` interfaces, differentiating between data association as a search through the robot pose configuration space, and as a search through the new to old feature correspondence space.

For feature based data association the abstract `FeaturePair` class provides a container for a single pair of a new and old feature, and the `Hypothesis` class implements storage for a set of such pairings.

If the Joint Compatibility Branch and Bound algorithm is to be used, then features must implement the `CompatibilityTestable` interface, which defines method signatures for testing individual compatibility between features.

State Estimation

This section provides only a single placeholder interface `StateEstimator`, due to the wide variation in possible implementations.

6.1.2 World State

The `WorldState` class tracks all aspects of the system state tracked by the system, in various forms as required by the different components in the system. It contains a `WallMap` instance which stores the global map of walls observed in the environment, an `Ekf` instance for state estimation and an `OccupancyGrid` instance used for route planning. The EKF also tracks the errors associated with the system state estimates, along with the correlation between corners caused by motion of the robot.

Updating of the tracked state is provided by wrapper methods for the move, turn & transition operations in the EKF, and a hypothesis application method wrapping the integrate and observe operations in the EKF. Each wrapper operation propagates any changes in one part of the world state to all others, to maintain internal consistency. Another function of `WorldState` is to enforce an approval period for observed corners, which must be observed a set number of times before being included in the EKF.

Wall Mapping

The `WallMap` class is a container for a connected map of observed walls, used as part of `WorldState` and also for post-processing of initial RANSAC output during feature extraction. The instances of `Wall` contained within the map consist of three alternate representations of the stored wall, for use in different parts of the system: the positions and types of the wall's endpoints, the central point, orientation and length of the wall and the line equation of the wall.

The endpoint type refers to the added complication of observing walls partially. An observed wall may extend outside the LIDAR scanner's field of vision, and so a detection procedure during feature extraction determines whether the computed endpoints are the real ends of the wall, or just some arbitrary point along it. These two endpoint types are termed ‘partial’ and ‘complete’.

Whenever a mapped wall is moved or a new wall added to the wall map, any new connections between walls are detected automatically and the appropriate action taken. Pairs of walls that meet or overlap and are co-linear are merged into a single larger wall, and any endpoints that meet but do not match in orientation are joined together to produce a paired corner. Merges may only take place between two partial endpoints, and joins may not take place between two complete endpoints. Joined endpoints constitute a ‘paired corner’, and un-joined complete endpoints are ‘unpaired corners’. The full procedure for this geometric processing is described in the feature extraction section of this report.

This internal merging and joining has implications for the corner approval period enforce by WorldState. Any partial endpoints merged together are of course erased, and so if approval counts exist they are erased. Any partial endpoint joined with another endpoint will circumvent the normal approval period - an intentional behaviour, as endpoints detected as forming a corner do not suffer the partial-vs-complete uncertainty. Any approval counts for joined endpoints are also erased.

6.1.3 Main SLAM Algorithm

See algorithm 5 for pseudo-code.

Initialisation

The algorithm begins by taking readings from the tilt sensors and compass, and initialising the world state. It then proceeds to make a number of initial observations before moving, to provide an accepted starting map tracked by the EKF. Without these initial observations the corner approval period maintained by world state could leave the entire starting map untracked by the EKF, and thus render state estimation inactive.

Map Exploration Loop

The main section of the algorithm then executes, with the outermost loop running until there are no unexplored areas left in the map. This is determined by checking the boundary set maintained by the occupancy grid within world state. If this loop runs, a new destination is selected and a route planned to reach it.

Route Navigation Loop
A new loop then starts which navigates the computed route plan, running until the plan has been completed or abandoned. At each iteration, the next section of the route is computed and the movement or turn required to execute this section on the current slope is computed (using models provided by the EKF).

Motion Execution
A separate motion execution thread is then started. The motion execution thread handles instructing the robot to move or turn, recording the progress of the motion, and interrupting the motion on detection of a slope transition.

Observation & Estimate Updating
Observations of the environment are made during the execution of the computed motion, within a loop that proceeds while the motion is still executing or has stopped since the last observation. A set delay is first performed, then the current progress of the motion (distance/angle) is retrieved and a new LIDAR scan is taken. These polling operations are performed at the start of each loop iteration so as to minimise timing error. The current progress of the motion is then applied to the EKF using the move or turn operation. If the motion was interrupted by the motion execution thread, then new tilt & compass readings are taken and the transition operation applied to the EKF, and a replacement LIDAR scan is taken. These steps should ensure that the world state is consistent ready for the observation step: walls are extracted from the scan and associated with the current wall map, and the hypothesis applied to the world state.

Loop Terminations
Once the plan section execution loop terminates, the rest of the route is checked against the updated map for blockages and recorded as abandoned if any are found. If the route plan is still valid and not yet complete then the plan execution loop will continue, else it will terminate. Once the map is completely explored the main loop terminates and the final wall map output.

Algorithm 5 Main SLAM Algorithm

```
initialise world state
for required number of initial observations do
    Observe environment
    Associate and apply to World State
end for
while unexplored area(s) remain do
    select new destination
    plan new route to destination
    while current plan not complete do
        compute next chunk of route
        begin motion execution thread
        while Motion executing, or stopped since last do
            Observe environment
            Get motion progress and update estimates
            if Motion was interrupted then
                Observe environment
                take tilt & compass readings
                apply transition to EKF
            end if
            Associate and apply to World State
        end while
    end while
end while
```

Chapter 7

Feature Extraction

7.1 Overview

A RAndom SAmple Consensus (RANSAC) based algorithm for extraction of straight wall features from a LIDAR range scan was developed, chosen because of the regular grid nature of the maze environment and the strong possibility of offshoots in the scan due to gaps between walls. A number of extensions were made to the standard algorithm, including two level candidate generation, minimum segment length and determination of partial and complete wall endpoints. *See algorithm 6 for pseudo-code.*

However, even after concerted parameter optimisation efforts the random nature of the algorithm still caused potentially significant variation in walls extracted from the same scan in repeated trials. The algorithm would sometimes output two slightly overlapping walls with very similar orientations, instead of what to a human observer would be a single straight wall. To overcome this variation a post-processing step was added to merge such walls together.

This post-processing algorithm also scans for pairs of close endpoints at different orientations, joining any pairs found by extending the two walls to intersect each other. This was necessary to ensure correct operation of the EKF. This functionality is actually provided through constructing a new `WallMap` object, of the same type used to record the global wall map. *See algorithm 7 for pseudo-code.*

7.2 Part 1: RANSAC

7.2.1 Algorithm Description

Extraneous Scan Filter

Initially any points beyond the maximum range of the scanner are filtered out. Individual extraneous scans are returned by the LIDAR at its maximum range, so including such points in RANSAC could cause non-existent walls to be extracted.

Sampling & Level 1 Candidate Line

The algorithm then begins to loop, terminating after a maximum number of iterations or if there are too few points remaining to construct a new line. First a random sample of a set size is taken from the pool of remaining points, within a certain radius of a randomly chosen centre point, and a first-level candidate line is fitted to the sample by linear-least-squares.

Association & Level 2 Candidate Line

Next, points from the pool are associated to the candidate line if they are within a set normal distance. A second-level candidate line is then fitted to all the associated points, and association is performed again. The distance tolerance for this first level association is more generous than for the second level, to allow for more points to be included in the second level generation.

Consensus Check & Point Clustering

If the number of points associated to the candidate line meets a set consensus level, the algorithm proceeds to cluster the associated points into continuous segments along the line. First the points are projected onto the line and sorted. If the gap between any two adjacent projected points exceeds a set distance tolerance, the points are segmented at that position. This is a necessary step because, especially with the regular grid nature of the maze environment, the same line equation could associate points either side of a junction in the maze. Without splitting these junctions would appear to be blocked up.

Consensus Check & Segment Bounding

For each cluster of points the consensus check is performed once again, and if it passes the

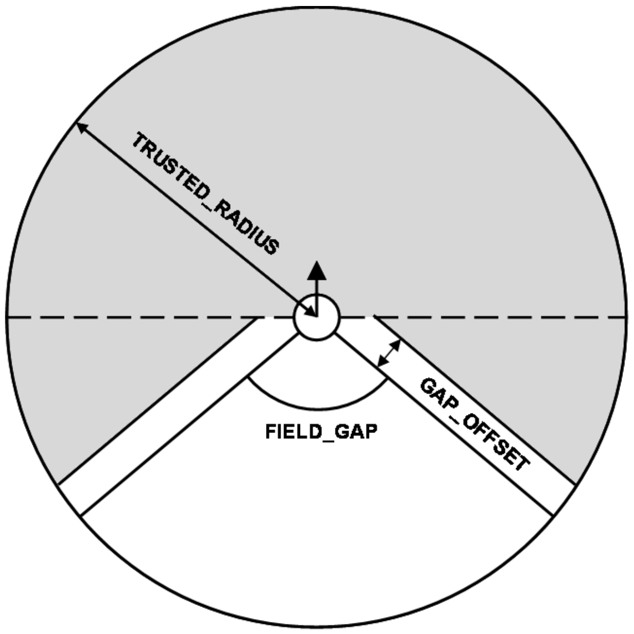


Figure 7.1: Endpoint type determination: points in shaded area classed as complete.

line is bounded to a segment covering only the points in the cluster, simply by taking the outermost points in the cluster (already sorted) as the endpoints of the wall feature.

Endpoint Type Determination

During this bounding step the type of both endpoints is determined. The LIDAR scanner is limited to a certain range and field angle, so a scan might only return points from part of a wall, rather than the complete wall. To ensure correct global mapping operation and data association a distinction must be made between complete endpoints (the actual true end of the wall) and partial endpoints (some position partway along the wall). The algorithm records any points outside of a set maximum trusted range (less than the maximum scanner range) as partial, along with any points in the blind spot behind the robot in its field of vision (or within a set buffer distance of this blind spot). *See figure 7.1.*

Minimum Length Check

Finally a check is made to ensure the wall is at least a set minimum length. If the wall is long enough it is accepted and the associated points removed from the pool, otherwise it is rejected.

7.2.2 Algorithm Parameters

ITERATIONS	Maximum number of iterations to perform
SAMPLES	Number of points to take in each sample
SAMPLE_RADIUS	Radius around centre within which to sample
LINE_TOLERANCE_1	Distance limit for 1st-level candidate association
LINE_TOLERANCE_2	Distance limit for 2nd-level candidate association
GAP_TOLERANCE	Maximum allowed point gap along line segments
CONSENSUS	Number of points required for wall acceptance
MIN_LENGTH	Minimum allowed wall length
MAX_RANGE	Maximum effective range of scanner
TRUSTED_RADIUS	Maximum trusted radius of scanner
FIELD_GAP	Angle of blind sector behind scanner
GAP_OFFSET	Buffer distance for blind sector

7.3 Part 2: Geometric Post-Processing

7.3.1 Algorithm Description

All pairs of endpoints (from different walls) from the initial RANSAC output are looped through and checked for various closeness properties, to determine whether a join or merge operation is appropriate.

Two endpoints may be joined by moving them to the intersection point of their respective walls, and they may be merged by replacing the two walls with a single wall connecting the two outer unmatched endpoints. When joining an extra check is performed to ensure the computed intersection is not too far away from the original points (just in case), and if the check fails the midpoint of the original points is used instead.

If two endpoints are within a set threshold distance of each other and the bearings of their walls are not similar, then they are joined. If the bearings are within a set angular tolerance then the endpoints are merged.

In some situations these basic checks would not be sufficient to detect endpoint pairs that should be merged, most commonly when the two walls overlap, so extra checks must be performed as follows if the endpoint positions do not pass the closeness test.

If the wall bearings are similar and within a set tolerance the walls follow along the same

Algorithm 6 RANSAC Wall Extraction

```
discard any scan points beyond $MAX\_RANGE$
for $iter \in 1$ to $ITERATIONS$ do
    if pool size $< CONSENSUS$ then
        terminate loop
    end if
    sample $SAMPLES$ points within $SAMPLE\_RADIUS$ of a chosen centre
    fit 1st-level candidate line to sampled points (by LLS)
    associate any points within $LINE\_TOLERANCE\_1$ of line
    fit 2nd-level candidate line to 1st-level associated points (by LLS)
    associate any points within $LINE\_TOLERANCE\_2$ of line
    if num. associated points $> CONSENSUS$ then
        project points onto line and sort
        split into clusters at gaps $> GAP\_TOLERANCE$
        for all point clusters do
            if num. points $> CONSENSUS$ then
                take first and last points as wall endpoints
                classify endpoints beyond $TRUSTED\_RADIUS$ as partial
                classify endpoints in blind sector $(FIELD\_GAP, GAP\_OFFSET)$ as partial
                if wall length $\geq MIN\_LENGTH$ then
                    remove points and accept wall
                end if
            end if
        end for
    end if
end for
```

line, then the algorithm first projects both endpoints onto their opposite wall. Then if the projected endpoints pass the closeness check or the endpoints overlap then they are merged.

7.3.2 Algorithm Parameters

MAX_GAP	Maximum gap between points for them to match
AVG_THRESH	Distance check for wall intersections
MAX_ANGLE	Maximum angle between wall bearings for them to match
MAX_LINE_GAP	Maximum gap between lines for them to match

Algorithm 7 Geometric Wall Processing

```
if point gap $\leq MAX\_GAP$ then
    if wall bearing difference $\leq MAX\_ANGLE$ then
        merge endpoints
    else
        join endpoints
    end if
else
    if wall bearing difference $\leq MAX\_ANGLE$ then
        if both endpoints within $MAX\_LINE\_GAP$ of opposite wall then
            project both endpoints onto opposite wall
            if projection point gap $\leq MAX\_GAP$ then
                merge endpoints
            else if walls overlap at endpoints then
                merge endpoints
            end if
        end if
    end if
end if
```

Chapter 8

Data Association

8.1 Testing Compatibility

In order to be able to associate walls, the software must have a method for testing whether two walls, observed at different times, could possibly be the same wall. To this end, the following factors were integrated to produce a method for testing compatibility:

Relative bearing: the difference in the bearings of the two walls. If this is above a certain threshold, the walls are not compatible.

Overlap: if the walls overlap by more than a certain amount, and they have passed the relative bearing test, then the walls can be said to be compatible.

Endpoints: if either of the wall's endpoints is within a certain range of an endpoint of the other wall, and they have passed the relative bearing test, then the walls can be said to be compatible.

The algorithm for testing compatibility between walls can be seen in Algorithm 8.

For testing Joint Compatibility, the JCBB algorithm performs a statistical test on the sum of all the individual compatibility scores from the pairings in the hypothesis. To speed this up, when two features are tested for individual compatibility, they store the value calculated in a `HashMap`, with the feature's HashCode as the index. This allows the individual compatibility scores to be accessed by JCBB in $O(1)$ time once they have been calculated. We can be sure that they will have been calculated when JCBB comes to ask for them since a feature pair will only be added to a hypothesis if it is individually compatible.

Algorithm 8 Wall Individual Compatibility Algorithm

```
Require: Walls $wall1, wall2$ Maximum Bearing Difference $b$, Maximum Endpoint Distance $e$
  if $wall1$.bearing - $wall2$.bearing $< b$ then
    {If the distance between the wall centres is less than half the sum of the walls lengths, they must overlap}
    if $wall1$.centre.distanceTo($wall2$.centre) $<$ ()$wall1$.length + $wall2$.length) / 2 then
      return Walls are individually compatible
    else
      {Check the distance between each pair of endpoints for whether any of them are within the maximum distance}
      if Distance between closest endpoints of $wall1, wall2$ ¡ $e$ then
        return Walls are individually compatible
      end if
    end if
  end if
  return Walls are not individually compatible
```

8.2 Implementation

A brief outline of our implementation of the JCBB algorithm can be seen in Algorithm 1 in the *Research* section. In order to make the algorithm as re-usable as possible, the code uses Generics to ensure it is passed features which are *CompatibilityTestable*, an interface which defines the `individuallyCompatible` and `individualCompScore` methods. Using these, different feature classes can define their own methods for determining compatibility; for example, walls may use their centre-point, orientation and length, whereas individual points may use their Euclidean distance. This means that the SLAM code can be converted from using one type of feature to another with relative ease.

The JCBB algorithm produces a *CompatHypothesis*, which is a Hypothesis that can be tested for Joint Compatibility. The `jointlyCompatible` function uses the results of Individual Compatibility between features to determine whether the hypothesis is consistent, and so can maintain portability by using the `indvCompScore` function. In this way, JCBB can be viewed as a library function which can be used in any situation with any *CompatibilityTestable* features.

Chapter 9

State Estimation

9.1 Overview

An Extended Kalman Filter implementation for optimal estimation of the robot pose and feature positions was developed. The various models required for this filter were developed to handle the kinematic movement of the robot over uniform sections of sloped floor. During the design stage the possibility of developing an EKF that could handle wall features directly was investigated, but little was found in published literature to aid in this effort, and even from initial design work the level of complexity seemed simply too great to be feasible.

It was decided therefore to develop a wrapper for the EKF which would pre-process wall observations and convert them to observations of corner features. EKF implementations handling such point features were readily available in published literature and their complexity is manageable. The wrapper functionality is provided by the `WorldState` object class used by the main SLAM algorithm.

9.2 Stored Matrices

The EKF maintains an internal representation of the system state vector (matrix X), and also its covariance matrix (matrix P). The system state consists of current best estimates of the robot pose and all observed corners. Essentially the state of the system is treated as comprising a set of Gaussian random variables, with their means and covariances tracked in these two matrices.

The robot pose is recorded by its horizontal x,y position, and three angles used to determine its orientation in 3D space. These are the incline of the slope the robot is currently on, the direction of the current slope in the horizontal plane and the current bearing of the robot on the slope relative to the slope direction. Observed corners are recorded simply by their horizontal x,y positions.

EKF framework definitions in the Stored Matrices & Operations sub-sections were gleaned from: [21], [11], [28], [29]

System State Vector:

$$X = \begin{bmatrix} x_r \\ y_r \\ \theta_{sv} \\ \theta_{sh} \\ \theta_{rp} \\ x_0 \\ y_0 \\ \vdots \\ x_{n-1} \\ y_{n-1} \end{bmatrix}$$

x_r	Horizontal x-axis position of robot's turning centre
y_r	Horizontal y-axis position of robot's turning centre
θ_{sv}	Incline of current slope
θ_{sh}	Horizontal direction of current slope
θ_{rp}	Robot bearing on current slope
x_c	Horizontal position in x-axis of corner with ID c
y_c	Horizontal position in y-axis of corner with ID c

System Covariance Matrix:

$P =$

r, r	$0, r$	$\dots$	i, r	$\dots$	j, r	$\dots$	$n-1, r$
$r, 0$	$0, 0$	$\dots$	$i, 0$	$\dots$	$j, 0$	$\dots$	$n-1, 0$
$\vdots$	$\vdots$	$\ddots$	$\vdots$		$\vdots$		$\vdots$
r, i	$0, i$	$\dots$	i, i	$\dots$	j, i	$\dots$	$n-1, i$
$\vdots$	$\vdots$		$\vdots$	$\ddots$	$\vdots$		$\vdots$
r, j	$0, j$	$\dots$	i, j	$\dots$	j, j	$\dots$	$n-1, j$
$\vdots$	$\vdots$		$\vdots$		$\vdots$	$\ddots$	$\vdots$
$r, n-1$	$0, n-1$	$\dots$	$i, n-1$	$\dots$	$j, n-1$	$\dots$	$n-1, n-1$

Each section in the matrix records the covariance between one element and another, for instance section r, i records the covariance between the robot pose and corner i. Opposite sections are the transpose of each other, so j, i is the transpose of i, j. The covariance between two different elements is a measure of the correlation between them, which is how the EKF tracks error correlations. A special case is the covariance between an element and itself, which is in fact the variance of the element. The sections along the diagonal of the system error covariance matrix record the individual variances of the robot pose and corners. The variance of an element is a measure of the uncertainty of the estimate for that element, which is how the EKF tracks errors.

9.3 Operations

The EKF framework provides two classes of operation that may be applied to the maintained system state: prediction and observation. A prediction operation applies some action (typically movement of the robot) to the system state directly, and as the name suggests the new system state is predicted. An observation operation inputs a new measurement of a part of the system, in this case a new sensor reading for a re-observed corner. This new reading is used to compute a new optimal state estimate, effectively by using the corner as a known frame of reference. In SLAM applications a third type of operation is defined: integration. An integration operation inputs a sensor reading for a previously unobserved feature, which is then added to the state representation. Although the operation observes a feature, it falls mathematically under the class of prediction operations.

Operations may be applied by the EKF sequentially and in any order, and so to reduce the level of model complexity our particular implementation defines separate prediction operations for movement, turning and slope transitions.

9.3.1 Predict Operation

$$X' = \mathrm{E}[f(X, u) + w]$$

Where:
X' is the new system state estimate
f is the process model function
u is the input vector
w is a zero-mean multivariate Gaussian process noise

The process model is the defined mathematical model describing the action taken by the operation (i.e. move, turn), and updating the system state estimate merely involves applying this model to the current estimate. The EKF framework assumes all error in the system is zero-mean and Gaussian, so the noise term disappears when taking the expected value.

$$P' = FPF^T + WCW^T$$

Where:
P' is the new system covariance
F is the Jacobian matrix of f with respect to X
C is the sample covariance of w
W is the Jacobian matrix of f with respect to w

The Jacobian matrix of some non-linear vector function is the matrix of partial derivatives of each element in the vector, with respect to all specified variables. The result is an approximation of the true derivative of the function, linearised around the specified variables. Here then the process model is linearised around the current system state estimate, and around the process noise. This is how the EKF can handle non-linear systems, although as the Jacobian is an approximation it will fail if there is too much error or the system is too non-linear.

Updating the system covariance matrix involves propagating existing error and correlation (feature correlation arises as a result of robot movement) and adding new error and correlation introduced by the operation. The matrix C is a representation of the uncertainty of the values in the input vector u, constructed as a diagonal matrix with an uncertainty value for each input value.

9.3.2 Observe Operation

$$y = z - \mathrm{E}[h(X, i) + v]$$

Where:
y is the measurement innovation
z is the new measurement vector
h is the observation model function

i is the ID of the re-observed feature
v is a zero-mean multivariate Gaussian observation noise

The first step simply calculates the difference between the actual reading and the expected reading given the current system state, termed the 'innovation'. The observation model computes the reading expected for the feature with the given ID. As with prediction operations, the noise term disappears when taking the expectation.

$$S = HPH^T + VDV^T$$

Where:
S is the innovation covariance
H is the Jacobian matrix of h with respect to X
D is the sample covariance of v
V is the Jacobian matrix of h with respect to v

$$K = PH^TS^{-1}$$

The next step computes the all important optimal Kalman gain K. This is a weighting matrix that is applied to the innovation, such that when the system state is adjusted by this weighted innovation the covariance of the system is minimised. Intuitively, the optimal Kalman gain best determines the varying levels of trust taken in the new observation. Lower uncertainty in the observation will cause it to be more trusted, lower uncertainty in the system state will cause the observation to be less trusted.

$$X' = X + Ky$$
$$P' = (I - KH)P$$

The final step updates the system state estimate and covariance based on the measurement innovation and using the computed Kalman gain as a weighting matrix.

9.4 Initialisation Model

This model is used only when the EKF is initialised, to setup the initial state estimate and covariance. Tilt sensor and compass noise is assumed constant relative to the produced bearings. Turning and movement noise is assumed proportional to the size of the action. Observation noise is assumed proportional in range but constant in bearing.

Inputs:

x_r, y_r	Initial robot position
θ_r	Initial robot bearing
$\theta_{pitch}, \theta_{roll}$	Initial tilt readings
c_{pitch}, c_{roll}	Tilt sensor inaccuracies
$c_{compass}$	Compass inaccuracy
c_{turn}	Turning inaccuracy
c_{move}	Movement inaccuracy
$d_{\rho c}, d_{\theta c}$	Observation inaccuracies
o_f, o_v	Forward & vertical scanner offsets

Model:

$$C_{turn} = \begin{bmatrix} c_{turn} \end{bmatrix} \qquad C_{move} = \begin{bmatrix} c_{move} \end{bmatrix}$$

$$C_{trans} = \begin{bmatrix} c_{pitch} & 0 & 0 \\ 0 & c_{roll} & 0 \\ 0 & 0 & c_{compass} \end{bmatrix} \qquad D = \begin{bmatrix} d_{\rho c} & 0 \\ 0 & c_{\theta c} \end{bmatrix}$$

These just set up the various sample covariance matrices used in the other models.

$$X^T = \begin{bmatrix} x_r & y_r & \theta_{sv} & \theta_{sh} & \theta_{rp} \end{bmatrix}$$

This sets up the initial system state estimate. The three bearings are calculated using the transition model.

9.5 Turning Model

This model handles the turning of the robot on the spot. Because the EKF stores the robot's centre of turning, the model needs only to adjust the bearing of the robot on the current slope.

Inputs:

$\Delta\theta_{rp}$	Turn angle

Model:

$$\theta'_{rp} = \theta_{rp} + \Delta\theta_{rp} \pmod{2\pi}$$

The turn angle is simply added to the robot's current bearing on the slope.

$$(X')^T = \begin{bmatrix} x_r & y_r & \theta_{sv} & \theta_{sh} & \theta'_{rp} & x_0 & y_0 & \vdots & x_{n-1} & y_{n-1} \end{bmatrix}$$

9.6 Movement Model

This model handles the straight-line forward or backward movement of the robot, and must take account of the current slope.

Inputs:

d_p	Move distance

Model:

$$\theta_{rph} = \begin{cases} \theta_{rp} & \text{if } \theta_{rp} = 0, \frac{\pi}{2}, \pi \text{ or } \frac{3\pi}{2} \\ \operatorname{atan}\left(\frac{\tan\theta_{rp}}{\cos\theta_{sv}}\right) \pmod{\pi} & \text{if } \theta_{rp} < \pi \\ \left(\operatorname{atan}\left(\frac{\tan\theta_{rp}}{\cos\theta_{sv}}\right) \pmod{\pi}\right) + \pi & \text{otherwise} \end{cases}$$

$$\theta_{rh} = \theta_{rph} + \theta_{sh} \pmod{2\pi}$$

This calculates the robot's bearing in the horizontal plane, by projecting its plane bearing onto the horizontal and adding the slope direction bearing. The extra calculation in for the projection is required to handle singularities at right angles, and loss of orientation information through using the arctangent function. Loss of precision during calculations in the Java language was also an issue.

$$\theta_{pitch} = \theta_{sv} \cos \theta_{rp}$$
$$d_h = d_p \cos \theta_{pitch}$$

This calculates the distance travelled in the horizontal plane, by projecting the distance travelled on the slope using the slope incline.

$$\Delta x_r = d_h \sin \theta_{rh}$$
$$\Delta y_r = d_h \cos \theta_{rh}$$
$$x'_r = x_r + \Delta x_r$$
$$y'_r = y_r + \Delta y_r$$

This calculates the horizontal translation in the x & y axes, and applies this translation to the current robot position.

$$(X')^T = \begin{bmatrix} x'_r & y'_r & \theta_{sv} & \theta_{sh} & \theta_{rp} & x_0 & y_0 & \dots & x_{n-1} & y_{n-1} \end{bmatrix}$$

9.7 Transition Model

This model handles the robot transitioning from one slope to another. Deducing the change in direction caused by a transition proved to be far too impractical, so instead a compass bearing is used.

Inputs:

$\theta_{pitch}, \theta_{roll}$	Pitch & roll tilt bearings
θ_r	Robot bearing from compass

Model:

$$\theta'_{rp} = \begin{cases} \theta_r & \text{if } \theta_{pitch} = 0 \text{ and } \theta_{roll} = 0 \\ \frac{\pi}{2} & \text{if } \theta_{pitch} = 0 \text{ and } \theta_{roll} > 0 \\ \frac{3\pi}{2} & \text{if } \theta_{pitch} = 0 \text{ and } \theta_{roll} < 0 \\ 0 & \text{if } \theta_{roll} = 0 \text{ and } \theta_{pitch} > 0 \\ \pi & \text{if } \theta_{roll} = 0 \text{ and } \theta_{pitch} < 0 \\ \text{atan}\left(\frac{\theta_{roll}}{\theta_{pitch}}\right) \pmod{\pi} & \text{if } \theta_{roll} > \pi \\ \left(\text{atan}\left(\frac{\theta_{roll}}{\theta_{pitch}}\right) \pmod{\pi}\right) + \pi & \text{otherwise} \end{cases}$$

This calculates the robot's bearing on the new slope, using the ratio of roll to pitch. The extra cases are again due to right angle singularities and loss of orientation.

$$\theta'_{sv} = \begin{cases} |\theta_{pitch}| & \text{if } \theta_{roll} = 0 \\ \frac{\theta_{roll}}{\sin\theta_{rp}} & \text{otherwise} \end{cases}$$

This calculates the incline of the new slope.

$$\theta_{rph} = \begin{cases} \theta_{rp} & \text{if } \theta_{pitch} = 0 \text{ or } \theta_{roll} = 0 \\ \text{atan}\left(\frac{\tan\theta_{rp}}{\cos\theta_{sv}}\right) \pmod{\pi} & \text{if } \theta_{rp} < \pi \\ \left(\text{atan}\left(\frac{\tan\theta_{rp}}{\cos\theta_{sv}}\right) \pmod{\pi}\right) + \pi & \text{otherwise} \end{cases}$$

$$\theta'_{sh} = \theta_r - \theta_{rph} \pmod{2\pi}$$

This calculates the direction of the new slope, by projecting the robot's slope bearing onto the horizontal and then subtracting this from the compass bearing.

$$(X')^T = \begin{bmatrix} x_r & y_r & \theta'_{sv} & \theta'_{sh} & \theta'_{rp} & x_0 & y_0 & \dots & x_{n-1} & y_{n-1} \end{bmatrix}$$

9.8 Integration Model

This model adds a newly observed corner into the state representation, using its sensor reading.

Inputs:

ρ_c Scan range

θ_c Scan bearing

Model:

$$\theta_{rph} = \begin{cases} \theta_{rp} & \text{if } \theta_{rp} = 0, \frac{\pi}{2}, \pi \text{ or } \frac{3\pi}{2} \\ \operatorname{atan}\left(\frac{\tan\theta_{rp}}{\cos\theta_{sv}}\right) \pmod{\pi} & \text{if } \theta_{rp} < \pi \\ \left(\operatorname{atan}\left(\frac{\tan\theta_{rp}}{\cos\theta_{sv}}\right) \pmod{\pi}\right) + \pi & \text{otherwise} \end{cases}$$

$$\theta_r = \theta_{sh} + \theta_{rph} \pmod{2\pi}$$

This calculates the current horizontal robot bearing, by projecting the robot's bearing on the current slope onto the horiztonal and adding the slope direction bearing.

$$o_{fah} = o_f \sin\theta_{rp}$$
$$o_{fuh} = o_f \cos\theta_{rp} \cos\theta_{sv}$$
$$o_{fh} = \sqrt{o_{fah}^2 + o_{fuh}^2}$$
$$\Delta x_{of} = o_{fh} \sin\theta_r$$
$$\Delta y_{of} = o_{fh} \cos\theta_r$$

This calculates the horizontal translation of the observation point caused by the forward scanner offset, relative to the robot's centre of turning.

$$o_{vh} = -o_v \sin\theta_{sv}$$
$$\Delta x_{ov} = o_{vh} \sin\theta_{sh}$$
$$\Delta y_{ov} = o_{vh} \cos\theta_{sh}$$

This calculates the horizontal translation of the observation point caused by the vertical scanner offset, relative to the robot's centre of turning.

$$\theta_{cr} = \theta_r + \theta_c \pmod{2\pi}$$
$$\Delta x_c = \rho_c \sin\theta_{cr}$$

$\Delta y_c = \rho_c \cos \theta_{cr}$

This calculates the horziontal translation from the observation point to the observed corner, by rotating the observation by the robot's current horizontal bearing. The LIDAR sensor is attached to a gimbal which automatically levels the sensor, so at this stage no consideration of slope is required.

$x_n = x_r + \Delta x_{of} + \Delta x_{ov} + \Delta x_c$
$y_n = y_r + \Delta y_{of} + \Delta y_{ov} + \Delta y_c$

This calculates the horizontal position of the new corner, by totalling the translations calculated previously and adding them to the robot's current position.

$(X')^T = \begin{bmatrix} x_r & y_r & \theta_{sv} & \theta_{sh} & \theta_{rp} & x_0 & y_0 & \ldots & x_{n-1} & y_{n-1} & x_n & y_n \end{bmatrix}$

9.9 Observation Model

This model computes the sensor reading for a corner that would be expected given its current position estimate and the current robot pose estimate.

Inputs:

i ID of observed corner

Model:

The values of $\theta_{rph}, \theta_r, \Delta x_{of}, \Delta y_{of}, \Delta x_{ov}, \Delta y_{ov}$ are calculated as in the integration model.

$\Delta x_c = x_i - x_r - \Delta x_{of} - \Delta x_{ov}$
$\Delta y_c = y_i - y_r - \Delta y_{of} - \Delta y_{ov}$

This calculates the horziontal translation from the observation point to the observed corner.

$\rho_c = \sqrt{\Delta x_c^2 + \Delta y_c^2}$

$$\theta_{cr} = \begin{cases} \frac{\pi}{2} & \text{if } \Delta y_c = 0 \text{ and } \Delta x_c > 0 \\ \frac{3\pi}{2} & \text{if } \Delta y_c = 0 \text{ and } \Delta x_c < 0 \\ 0 & \text{if } \Delta x_c = 0 \text{ and } \Delta y_c > 0 \\ \pi & \text{if } \Delta x_c = 0 \text{ and } \Delta y_c < 0 \\ \operatorname{atan}\left(\frac{\Delta x_c}{\Delta y_c}\right) \pmod{\pi} & \text{if } \Delta x_c > 0 \\ \left(\operatorname{atan}\left(\frac{\Delta x_c}{\Delta y_c}\right) \pmod{\pi}\right) + \pi & \text{otherwise} \end{cases}$$

$\theta_c = \theta_{cr} - \theta_r \pmod{2\pi}$

These calculate the scan range and bearing that consistute the expected sensor reading. The scan bearing is calculated by computing the bearing from the observation point in the global horizontal plane, and then rotating by the robot's current bearing.

$$h = \begin{bmatrix} \rho_c \\ \theta_c \end{bmatrix}$$

9.10 Hypothesis Wrapper

The wrapper functionality for converting from wall observations to corner observations is provided by the `WorldState` class, in a function which applies a computed hypothesis to the EKF and propagates the changes to the other parts of the `WorldState` object.

First the hypothesis is separated into re-observations of walls already observed, and new observations of so far unobserved walls. The cartesian positions of the observed endpoints are converted to sensor readings before being used, and for re-observations the correct orientation of the new observations over the stored wall must be determined.

Re-observations are handled first, beginning with separating them out into various observed endpoint groups. Processed observations of complete endpoints are applied to the EKF in these groups in sequence so the better observations provide information for the others. Processed observations of partial endpoints are stored until the final stage when the wall map and occupancy grid are updated. The groups are as follows:

1. Paired corners with both walls observed

2. Paired corners with one complete endpoint observed
3. Unpaired corners with a complete endpoint observed
4. Paired corners with one partial endpoint observed
5. Unpaired corners with a partial endpoint observed
6. Partial endpoints

Group 1 Observations:

If the two observed endpoints are close enough (or even are joined), then the intersection of the two walls is used as the observation passed to the EKF. Cartesian representations of the observed walls are computed from the sensor readings, the intersection computed, and a new sensor reading computed. Otherwise, if one of the observed endpoints is close enough to the stored position then it is moved to either group 2 or group 4 depending on the type of the observed endpoint.

Group 2 Observations:

If the observed endpoint is close enough to the stored position then it is used as the EKF observation, otherwise it is moved to group 4.

Group 3 Observations:

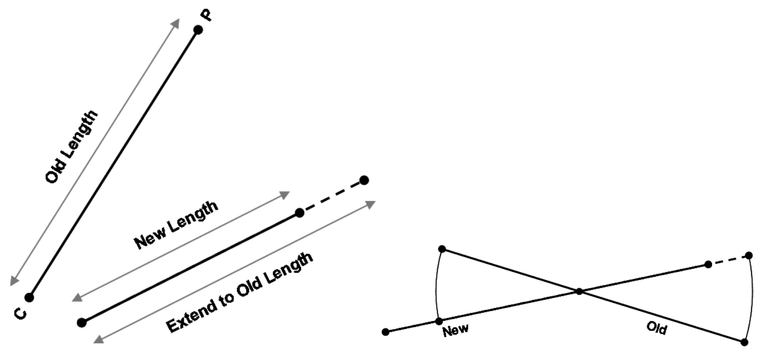
If the observed endpoint is close enough to the stored position then it is used as the EKF observation, otherwise it is discarded.

Group 4 Observations:

If the observed endpoint is very close to the stored position then it is used as the EKF observation. Otherwise, if it is close enough to the stored position and the other end of the other (stored) wall is a complete endpoint, then the intersection between the observed wall and the stored other wall is used. Otherwise the observation is moved to group 5.

Group 5 Observations:

If the observed endpoint is very close to the stored position then it is used as the EKF observation. Otherwise, if it is close enough to the stored position and the other end of the



(a) Other end complete (b) Other end partial

Figure 9.1: Partial endpoint observation handling

other (stored) wall is a complete endpoint, then the intersection between the observed wall and the stored other wall is used. Otherwise the observation is discarded.

At this point all re-observations of complete endpoints have been applied to the EKF, so the updated corner positions may be retrieved. Only positions that differ from the previous positions (still in the wall map) are recorded for updating the wall map.

Group 6 Observations:

If the other end of the stored wall is a complete endpoint: calculate the length of the new wall observation. If the new length is longer than the length of the stored wall then the observed point is used as the new position. Else if the observed point is close enough to the stored position, then extend the line from the other end (position from EKF) to the observed end up to the length of the stored wall. *See figure 9.1a.*

If the other end of the stored wall is a partial endpoint: compute the intersection of the stored and observed walls. Next calculate the distances from this intersection to the stored point and to the observed point. If the distance to the new point is longer then use it, otherwise extend the observed wall to the longer stored point distance. *See figure 9.1b.*

The next step is to adjust all updated endpoint positions within the wall map, paired corners first, then unpaired corners, then partial endpoints. Observations of new walls are then handled, simply adding them as new walls into the wall map. Another responsibility of the `WorldState` class is the enforcement of a validation period for any observed endpoints marked as complete. The output from feature extraction may well be imperfect, and it was judged that integrating complete endpoints into the EKF immediately would then require difficult detection of mistakes, and the removal of corners from the EKF which could introduce errors into the system covariance matrix.

Therefore when a new wall is observed with complete endpoint(s), a count of the number of times they have been observed as complete is started. The endpoints are then re-marked as being partial and added into the wall map, but not the EKF. Complete endpoints in re-observations are handled by incrementing this count. If a set acceptance threshold is eventually met then then the endpoint is elevated in the wall map to be complete, and integrated into the EKF.

Of course any movement of existing walls or addition of new ones to the wall map could result in automatic internal joins or merges, all of which are recorded. Corners created by internal endpoint joins are integrated into the EKF. A complete record of all changes made to the wall map is maintained which is then passed finally to the `OccupancyGrid` instance within `WorldState`, thus providing the route planning algorithm with the latest map.

Chapter 10

Route Planning

For the robot to be fully autonomous, it must be able to not only map its environment, but also navigate through the arena at the same time. Thus, the SLAM algorithms are supplemented with Planning algorithms that determine where the robot should travel to next, and how it can get there. These algorithms shall be discussed in this section.

10.1 Occupancy Grid Mapping

The robot maintains two maps of its environment: a feature-based representation that is used during the SLAM operations, and an Occupancy Grid which is used in the Route Planning sections. The Occupancy Grid is a much more traditional form of map, since it shows clearly which areas of the environment the robot has seen and where obstacles lie, as opposed to the more abstract map created by SLAM.

The map is an instance of the `OccupancyGrid` class, and is implemented with a 'Vector of Vectors' - an `ArrayList` object, each element of which is an `ArrayList`. This is only modified through a specific set of operations, which ensure that the size of the map is maintained such that no Null Pointer Exceptions are encountered. Through these functions, the map can increase in size in any direction, through a process of copying and re-sizing. These methods also maintain the coordinate transforms required to map real-world coordinates to map coordinates, which differ by a preset constant that defines the resolution of the map. These coordinate transforms also include a y-component transform, which places the origin of the map in the bottom left corner, rather than in the top right as is usual for matrix and array operations. Each element of the map is a *MapState* value; an enumerated set that defines common situations, such as a grid point being Unmapped, Unoccupied, a Wall, etc. This allows new states to be added quickly and easily.

Data can be drawn onto the map in one of two ways: by setting an individual grid point to a certain value, or by drawing a line between two points. The first is useful for situations such as plotting the robot's location. Drawing lines is much more common however, since all walls, scan lines and paths can be represented with straight lines. For this, Bresenham's line algorithm was implemented, which is an efficient and common line drawing algorithm.

For debugging, the Occupancy Grid class defines method to allow the map to be output to a PNG image. A small Graphical User Interface was also designed, which was used extensively in the SLAM simulations performed during development and testing. As well as this, the map is sent intermittently to the client (if one is connected), so that the robot's operator can see what the robot has mapped and how it is planning to navigate through the map. This client view also provides the ability to save the map in the GeoTIFF format, which shall be discussed in the GeoTIFF section.

10.1.1 Maintaining the Map

The Occupancy Grid is maintained by the *WorldState* object, which also maintains the SLAM map of the environment. Thus, whenever the SLAM map is updated, the Occupancy Grid map is also updated. Many of these updates will be the addition of new walls, however movement of existing walls will also be required since their locations may change based upon the information retrieved through Data Association. Since the Occupancy Grid doesn't store the positions of walls, merely their representation on the map, the World State has to keep a copy of any wall that it moves or erases, so that Occupancy Grid can write-over the grid locations that represent that wall, and thus display more accurately the map that is maintained by SLAM.

As well as this, the LIDAR data is added to the grid by the *SLAM* object, since the World State doesn't deal with any of the raw sensor data.

10.2 Choosing the Next Location

For the robot to navigate autonomously, it must be capable of choosing areas of the map that are worth investigating. For this, the robot can use one of three main methods. In all methods, a *Possible Point List* is used to store the locations the robot intends on investigating. This is a linked list of points, such that the robot has the ability to store ‘options’, where it may have the option of moving to one of a number of points from a given point. In this way, the robot should be able to navigate the entire arena by adding all possible exploration points to this list at each time step, and then navigating to the point at the top of the list at each time step. The list provides a basic filtering technique to prevent it from including any points that are close to points which have previously been explored or are already in the list.

10.2.1 Directional Movement

One way the robot can move around the course is through a simple search in four directions: straight ahead, behind, left and right. If the robot finds it can move in any of these directions, they are added to the list of possible exploration points. The robot will prioritise moving forwards over turning, since straight line motion is better for the production of an accurate map. If the robot has the ability to move either left or right, then the algorithm will explore these points for their potential benefit. This involves searching around those points to discover how far they are from unexplored areas on the map in a simple increasing radius search. Using this, the robot can assign priorities to the different points based upon the amount of information the robot stands to learn from exploring at that location. An example of this exploration technique can be seen in Figure 10.1.

10.2.2 Boundary Points

Another way the robot can choose to explore the environment is by always navigating to points on the boundary between what it has seen, and what is unexplored. To ensure the robot does not run into any obstacles, these points lie on an inner boundary a robot’s length in from the actual boundary. Using this technique, the robot can explore more of the environment quicker than with the Directional Movement technique, but it has the downside that the movements can be quite complex. Given sufficiently accurate localisation methods, however, it is the preferred navigation method. An example of this technique can be seen in Figure 10.2.

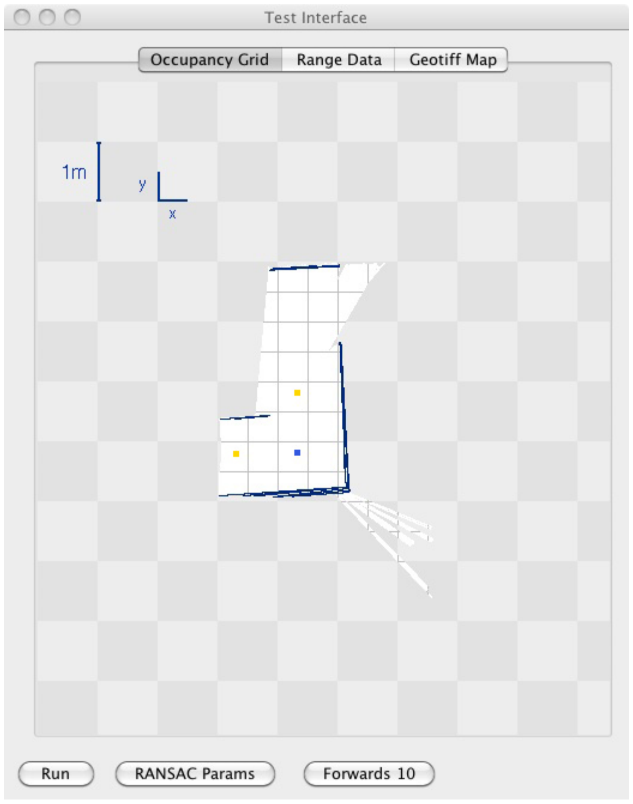


Figure 10.1: Possible next exploration points using the Directional Movement technique. The robot is at the blue point, and the yellow points are points it could move to.

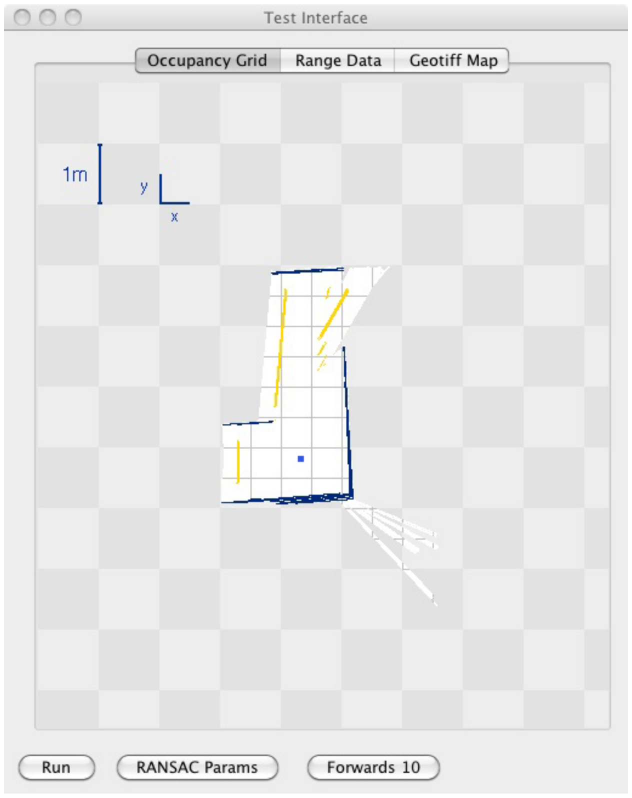


Figure 10.2: Possible next exploration points using the boundary Points technique. The robot is at the blue point, and the yellow points are points it could move to.

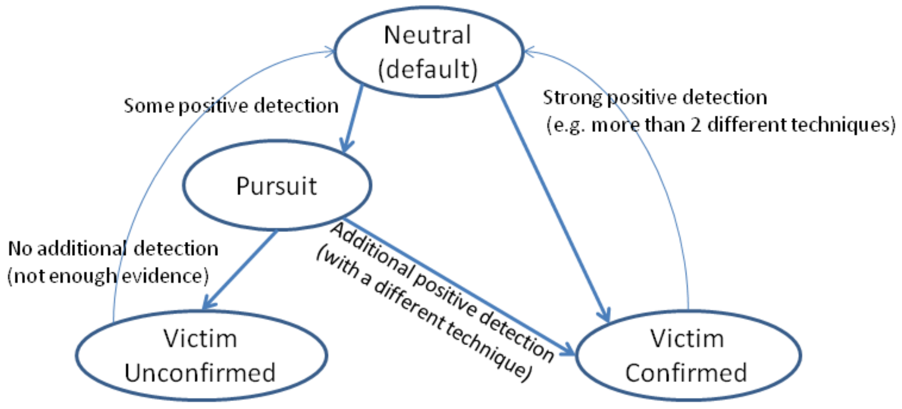

Neutral (default)
Some positive detection
Strong positive detection (e.g. more than 2 different techniques)
Pursuit
No additional detection (not enough evidence)
Additional positive detection (with a different technique)
Victim Unconfirmed
Victim Confirmed

Figure 10.3: A state diagram describing how the Victim ID software interacts with the Route Planning software.

10.2.3 Searching for Victims

The Victim Identification software has the ability to tell the planning algorithms that it thinks there might be a victim in a certain direction. The planning algorithm can then look in that direction until it finds either an obstacle or a boundary between a mapped and unmapped area, and adds that point to the top of the Possible Point list. In this way, the Victim Identification component can force the robot to follow certain paths along which it thinks victims might be found. While this may not help the exploration of the environment it will improve the chances of victims being positively identified which ultimately is the where the most competition points are won. A state diagram explaining how this interacts with Victim Identification can be seen in Figure 10.3.

10.3 Choosing a Route

Once the next location has been found, the route to this point can be calculated using a simple A* algorithm. Various heuristics were tested, including ones which tried to prioritise paths in which the robot moved using (as much as possible) only forward motions and turns through 90°. These, however, tended not to perform as reliably as simpler heuristics, such as the combined distance from the start and goal points, so for the sake of performance this very simple heuristic was used. A problem this method exhibited was that it would sometimes produce routes which involved incredibly fine motions, such as turns of 5° and under. Since such motions can make the robot's motion jerky, and since large numbers of small motions can be damaging to the motors and gearboxes, a filter was added which removed all motions under certain thresholds, such as turns of less than 5°. Furthermore, the route is smoothed, the algorithm for which will be discussed in a later section.

10.4 Avoiding Obstacles

An addition to the A* algorithm was the ability for it to check that each point it considers actually has enough space around it to accommodate the robot. This would allow the robot to rely less upon its sensors for collision avoidance, since the route should not take it too close to any obstacles. Checking that a particular grid point has enough space for the robot is no easy task, since the bearing of the robot cannot be determined absolutely by the planning algorithm. Another issue is that the algorithm doesn't need to check behind the robot, since it can be assumed that wherever the robot has come from is obstacle free. This is important since the initial scan will not see behind the robot due to the field of view of the LIDAR, and this unseen area behind the robot must not affect the generation of a path.

After many attempts, a simple algorithm that conservatively determines suitability for a grid point was produced. It works by checking four quadrants around the point, each quadrant having sides of length half of the robot's length. Using the previous point in the route so far, the algorithm can estimate the general bearing of the robot, and choose which quadrants to check dynamically. See Figure 10.4 for a diagrammatic representation.

10.5 Route Smoothing

The output of the A* algorithm is a series of adjacent grid references from the occupancy grid, which on their own would produce a very jerky and slow robot motion. To produce a nicer motion, the Planning algorithm performs smoothing, which attempts to find a small number of lines which represent the motions suggested by the list of grid references. It does this using Algorithm 9, which reduces the grid references to a set of turning points. By connecting these with straight-line motion, the robot will move in a path much smoother than the one produced by the A* algorithm.

The algorithm takes as its input a list of n points that make up the path, and then attempts to turn this into a series of lines by drawing lines between two points in the list,

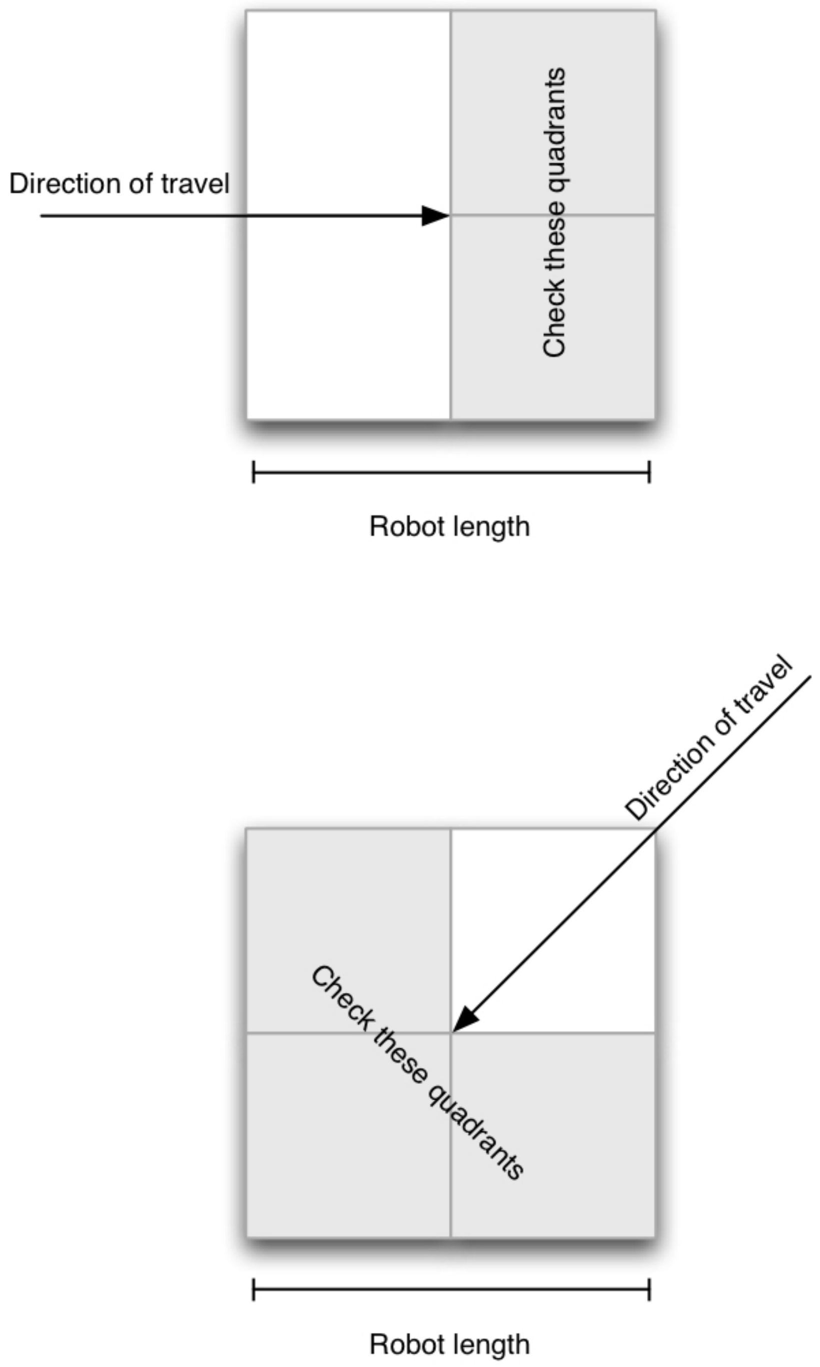


Figure 10.4: Checking that a point has space around it to fit the robot. The grey squares are the quadrants around the point that the algorithm will check.

and checking that every point in-between those two is sufficiently close to the line. If they are, it adds the next point in the list to the line and repeats until it finds a point that does not fit. At this point, it considers all points thus far as a single line, and then starts again from this point. In this way, the algorithm constructs $O(n)$ lines, and on each iteration checks $O(n)$ points for their proximity to the line. As such, in the worst case it is an $O(n^2)$ algorithm, but performs significantly better in practice.

Algorithm 9 Route Smoothing

```
Require: Array PathPoints, ConfidenceLevel
  RoutePoints ← new Route
  start ← 0
  end ← 2
  RoutePoints.add(PathPoints(start))
  while end < PathPoints.size() do
    Line ← new Line(PathPoints(start), PathPoints(end))
    {Construct a line between the two points we are considering}
    if ∀ Point p ∈ PathPoints[start..end] : p.distanceFromLine(Line) < ConfidenceLevel
    then
      {If all points between the two Path Points are on, or close to, this line, then see if we
      can smooth this route further}
      end ← end + 1
    else
      {If the points aren't on the line, then we can only smooth up to the previous end
      point, so add this to our route}
      RoutePoints.add(PathPoints(end - 1))
      start ← end
      end ← end + 2
    end if
  end while
```

10.6 Navigating

Once the route has been planned, the path points are converted into the required series of turns and movements by the EKF. These motions can be passed (with a bit of unit conversion: the SLAM code generally works in millimetres and radians, whereas motion is concerned with meters and degrees) to the *MotionController* class, whose operation will be discussed in the *Devices and Porting* section.

Chapter 11

Simulator

The simulator tool was developed by the former years Engineering team for use with their robot. However it was incompatible with the SLAM software being developed and therefore needed to be modified. The simulator was also slow and very buggy so other all round major improvements were also desired.

11.1 Changes to existing simulator tool

The changes made to the existing simulator tool ranged from structural changes to efficiency changes and new functions. Changes included:

- Incorporating SLAM
- Improving the interface
- Improving the simulator's overall architectural design
- Adding collision detection

11.2 Incorporating SLAM

To make the simulator useful for the project, a new SLAM function had to be able to be called from the simulator tool and be able to communicate with the tool, as it was unable to do this with the existing tool. The SLAM function needs to be called with references to a LIDAR sensor, a motion controller and a tilt sensor. The SLAM function polls the

LIDAR sensor for its data, uses the motion controller to move the robot and polls the tilt sensor for information on the robot's pitch,roll and heading. All these interfaces needed to be simulated so that they matched what the SLAM function expected. It was decided that the `datastore` class in the simulator should hold all information about the state of the robot and its surroundings.

This means that there would be two LIDAR classes, one to provide the simulated LIDAR data to the `datastore`, and another that could be passed to the SLAM function as an interface to the `datastore`. The first class first retrieves the robot's position from the `datastore` and uses this with the internal map to get a representation of the area around the robot. The class then converts this into suitable LIDAR data format and puts this into the `datastore`. The second class then just needs a single method `poll()` that is called by the SLAM function and retrieves the latest LIDAR data from the `datastore`.

The motor controller was trickier to design as it had to implement an interface to control the position of the robot. It was decided that the main position class called `orientation` would control all changes to the robot's position and therefore the motor controller device which is passed to the SLAM function could just interface between SLAM and the `orientation` class. When the SLAM calls the motor controller devices `setMotorVel` method, the motor controller device calls the corresponding method in `orientation` which then changes its internal speed variables. `orientation` is also a runnable thread which runs every 10th of a second, each run the robot's position is retrieved from the datastore and changed using the speed variables, then the `datastore` is updated with this latest robot position.

The `orientation` class also calculates the simulated robots pitch and roll, which is also added to the robot's position. This can easily be retrieved by the tilt sensor which is passed to the SLAM function.

11.3 Interface Improvement

To initialise the robot, values had to be inputted via the command line which resulted in being long, tedious and repetitive for the user. This is solved by implementing a simple GUI interface, so that map location could be browsed to and the robot's starting data could be input easily and all at the same time.

11.4 Improvement of Architectural Design

The previous simulator did not follow proper design principles and thus was too inefficient, therefore there was a redesign of how the simulator tool ran and communicated within itself. The `datastore` was designed as an overall container of information about the robot's state and the state of the world around it. The `datastore` could then be passed to classes (such as sensors) which would then have access to all this information as soon as it was updated. The `datastore` also kept a log of the robot's position over time. It was decided that the position would be logged twice per second. To do this the `PositionLogger` class was changed to be a runnable thread that ran every half a second and logged the current position within the datastore. These positions are then used after the simulator has finished to print out the snapshots of the robot on the map at each time interval.

11.5 Adding Collision Detection

Collision detection was something that needed to be added to the simulator for testing so that it was known if the SLAM software was navigating the robot into a wall. It was decided that the ideal place to implement collision detection was within the `orientation` class when the new robot position was being calculated. To check if a collision had occurred the robot's length and it's width is retrieved first, then the co-ordinates of the four corners of the robot are calculated using it's length and it's width and current position. For every pixel within a box just surrounding the robot, there can be a check for whether that pixel represents a wall and whether that pixel is within the robot using the lines created from the corners of the robot. To check whether or not a point (x,y) is within a square defined by $(p1x, p1y) \rightarrow (p2x, p2y) \rightarrow (p3x, p3y) \rightarrow (p4x, p4y)$ then calculate 4 values from these points using Algorithm 10.

A rough outline of the interactions between several key components of the simulator is shown in Figure 11.1. The figure shows the simulator process starting up three other processes: the LIDAR sensor, SLAM and Motor encoder. These three processes are linked together by a another entity: the `RobotPosition`. The simulated LIDAR sensor uses the `RobotPosition` and the internal map within the simulator to create LIDAR scans. These scans are supplied to the SLAM software which will use them to ultimately direct the robot in a certain direction. For this it supplies the simulated motor encoder with instructions

Algorithm 10 Check point inside square

```
a1 = y - p1y - ((p2y - p1y)/(p2x - p1x))*(x - p1x);
a2 = y - p2y - ((p3y - p2y)/(p3x - p2x))*(x - p2x);
a3 = y - p3y - ((p4y - p3y)/(p4x - p3x))*(x - p3x);
a4 = y - p4y - ((p1y - p4y)/(p1x - p4x))*(x - p4x);
if a1 >= 0 && a2 >= 0 && a3 >= 0 && a4 >= 0 then
    return true
else
    return false
end if
```

which the encoder takes and changes the `RobotPosition`. Within each of these interactions there also needs to be many other processes working away which are not shown here. For example any collision (`RobotPosition` hits a wall) needs to be detected but this shows the general outline of the proposed simulator. The output of the simulator is a list of images which show the map and the robot's position on each map image, plus the LIDAR scans every timestamp which is user defined.

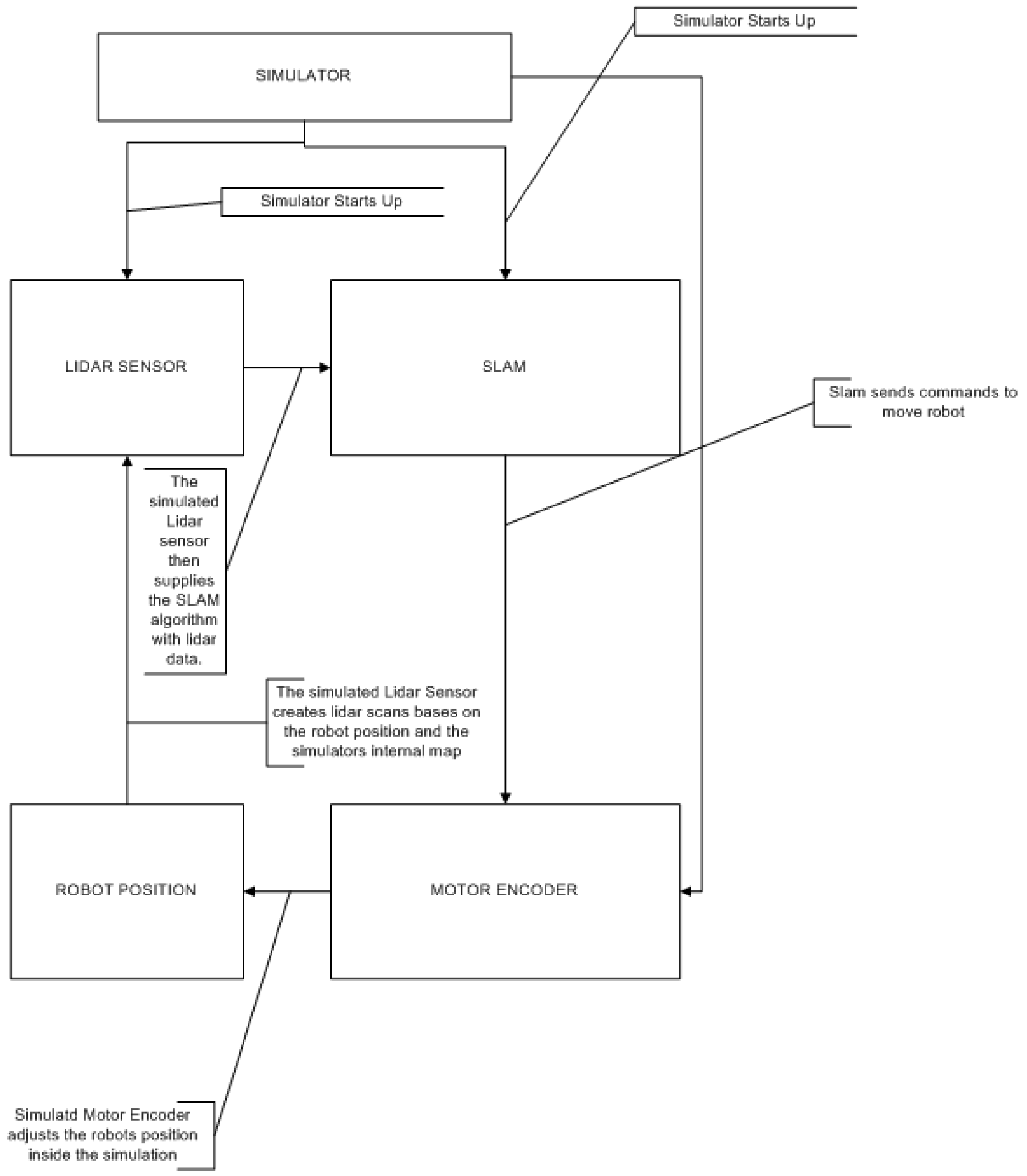

Simulator Starts Up
SIMULATOR
Simulator Starts Up
LIDAR SENSOR
SLAM
Slam sends commands to move robot
The simulated Lidar sensor then supplies the SLAM algorithm with lidar data.
The simulated Lidar Sensor creates lidar scans bases on the robot position and the simulators internal map
ROBOT POSITION
MOTOR ENCODER
Simulatd Motor Encoder adjusts the robots position inside the simulation

Figure 11.1: Interaction of the simulator's components

Chapter 12

GeoTIFF

The GeoTIFF module is used for exporting the environment map produced by SLAM to the GeoTIFF image format in accordance with the RoboCup Rescue competition rules. For the implementation of the module, we utilise the GeoTools open source toolkit and the Java Advanced Imaging (JAI) package. In this section, the steps in the export process will be described.

12.1 The Map Export Process

The export to GeoTIFF proceeds in a number of steps as shown in a high-level flowchart in Figure 12.1. The individual steps are described in the following subsections.

12.1.1 Internal Map Representation

The input to the GeoTIFF module is an internal representation of the map produced by SLAM. The map is represented as a 2-dimensional array of `MapState` elements. A `MapState` element is an enumerated type, representing a point in the map as being in one of the possible states, such as *unmapped, wall, obstacle, victim*, etc.

12.1.2 Creating a `TiledImage` with the Map

Since GeoTIFF is an image format with associated geographical data, an image representation of the map needs to be created. The recommended image representation is a `TiledImage`, which is a Java representation of an image with a sample model for individual

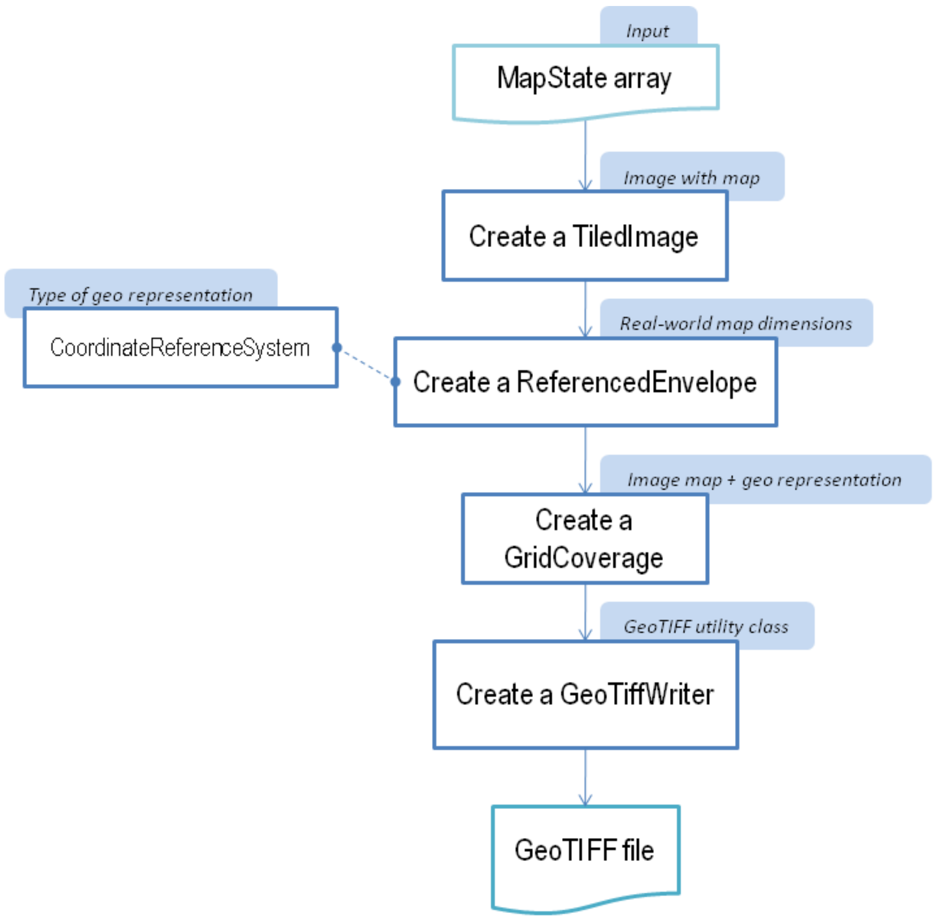

Input
MapState array
Image with map
Create a TiledImage
Type of geo representation
CoordinateReferenceSystem
Real-world map dimensions
Create a ReferencedEnvelope
Image map + geo representation
Create a GridCoverage
GeoTIFF utility class
Create a GeoTiffWriter
GeoTIFF file

Figure 12.1: GeoTIFF Export Process

image tiles of a set width and height. Java Advanced Imaging provides methods for creating and manipulating tiled images.

The input data consists of enumerated types, which lead to a decision to create a custom colour model, mapping each of the enumerated values of `MapState` to a colour in the RGB scheme. In this way, different information on the map can be encoded in different colours. In addition to rendering the information from the input array, other information is shown on the map, such as an underlying chess-board-style grid, which adds convenient scale information for the user and which is required by the competition rules. Also, a small scale indicator and a compass are placed on the image.

12.1.3 Creating a `ReferencedEnvelope`

A `ReferencedEnvelope` is an envelope, defined by a 2-dimensional rectangle, and an associated coordinate reference system. In essence, the envelope describes a region in the real-world (in real-world scale) and assigns a coordinate reference system with it, which allows for geospatial data to be modelled and projected onto the real-world coordinates. GeoTools provides methods for creating referenced envelopes, as well as grid coverages, discussed in the next step.

12.1.4 Creating a `GridCoverage`

A grid coverage is essentially a data structure with a 2-dimensional matrix of map values and associated information about the geographical positions of the values. In general, a grid coverage does not need to be created using an image representation of the map, as pure numerical values could be used as well. However, such a representation would lack colour information and would lead to a GeoTIFF file containing only geographical information and no map visible in a conventional image viewer. In our case, it is created using the image representation of the map and a referenced envelope, which provides the real-world scale of the map.

A grid coverage can be directly queried for specific positions within the map and information about the real-world geographical locations can be determined. In our scenario, real-world coordinates are set within a ‘simulated’ real-world region which simply provides a space of the desired number of square meters, starting from coordinate (0,0) meters in the top-left corner and spreading to (*max. width*, *max. height*) meters in the bottom-right corner of the map. In most practical applications, such as GIS systems, the region of the map is set to real-world locations, usually defined by latitude and longitude of the region on Earth being mapped.

12.1.5 Writing the GeoTIFF file

A grid coverage represents enough information for an export to the GeoTIFF format. In order to export the map, a `GeoTiffWriter` is created, which is a special implementation of a `GridCoverageWriter` that outputs a file in the GeoTIFF format.

12.2 The Map Viewer GUI

For the purposes of showing an intermediate map produced by SLAM during the robot's movement in the environment, a GUI window has been developed. The `MapViewer` window makes use of the GeoTIFF module's code to render a `TiledImage` from the `MapState` matrix produced by SLAM. It essentially produces a similar image representation of the map as in the GeoTIFF module and displays the map in the client GUI to the user. Examples of the map produced by `MapViewer` can be for in the Route Planning section, such as Figure 10.1.

Chapter 13

Victim Identification

13.1 Victim Identification: General Description

13.1.1 Overview

The Victim Identification component of the robot software deals with the task of identifying potential victims in the simulated environment using a variety of detection methods. It is designed to employ a number of different techniques to collect evidence from the robot's environment, to collate the evidence and pass the overall results to other components, such as SLAM and the client GUI. The detection methods make use of input services, such as a web camera, a infrared camera and a CO_2 sensor. This section will provide a high-level overview of the Victim Identification components and their operation.

13.1.2 Operation of Victim Identification

Victim Identification employs a number of different detection techniques, each of which is located in its own subcomponent. On the whole, Victim Identification can be divided in two general parts:

1. Main control classes

2. Detection classes

Since the majority of image processing and detection work is implemented in C language, a main Java control class has its corresponding main C class. While interaction with other

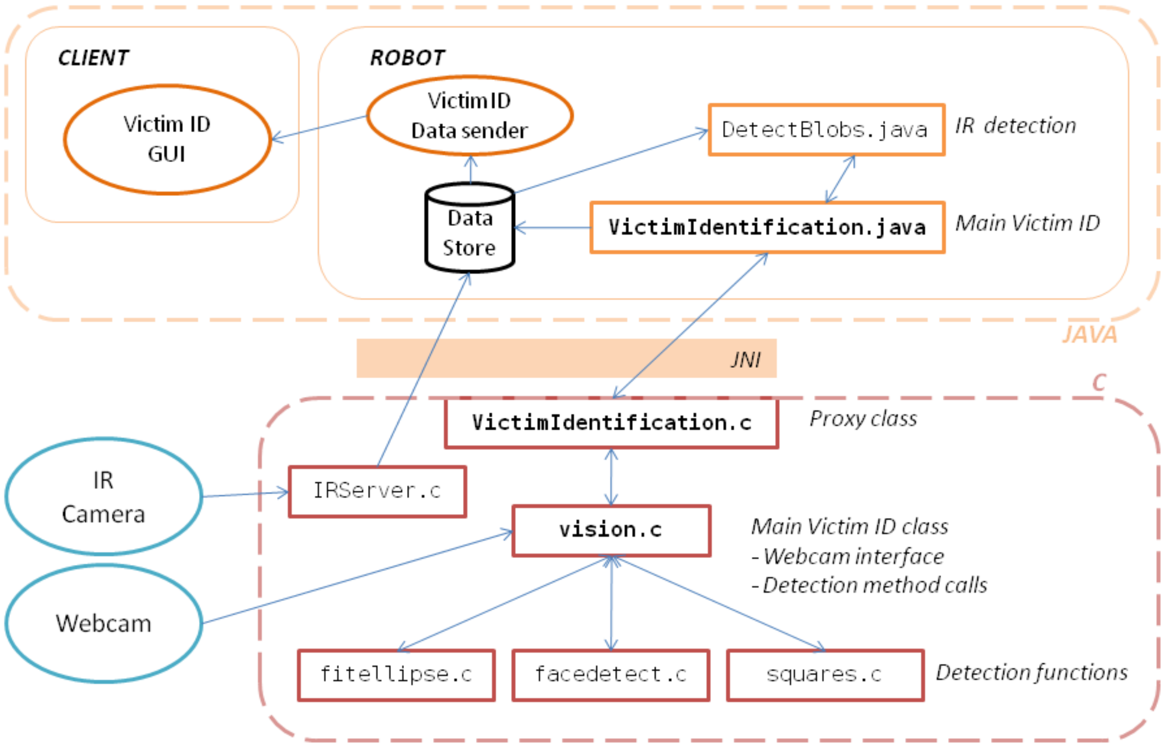


Figure 13.1: Victim Identification - component diagram

components of the robot's software is done through the main Java class, interaction with the image processing and detection code in C is done through the main C class. The component diagram in Figure 13.1 illustrates the structure of the Victim Identification classes, interaction among them, as well as with other parts of the robot code. More detailed descriptions of each class can be found the the following Implementation section.

Overall control flow of Victim Identification is managed through the main Java class. This class provides methods to other components, such as SLAM, which may trigger the detection process. This way, lower level calls within the Victim Identification component are encapsulated and hidden away from external users. When detection is triggered by an external call, the main Java class will issue a series of commands, such as retrieving a frame from the camera and calling a number of detection methods. Also, it will retrieve detection results from the relevant methods, evaluate these results to formulate an appropriate response for SLAM and collate these results into a Victim Identification result object which is returned.

When the detection process is triggered, the high level flow managed by the main Java class is the following:

	Action	*Component*
1.	Read in configuration file	`VictimIdentification.java`
2.	Grab an image from the web camera	`vision.c`
3.	Detect holes and eye-charts	`fitellipse.c`
4.	Detect faces	`facedetect.c`
5.	Detect squares	`squares.c`
6.	Detect heat signatures	`DetectBlobs.java`
7.	Retrieve images with results marked	`fitellipse.c, facedetect.c, vision.c, squares.c`
8.	Compute angle for detected objects	`fitellipse.c, facedetect.c, vision.c, squares.c`
9.	Evaluate detections and decide on positive ID of victim	`VictimIdentification.java`
10.	Produce a result object for SLAM	`VictimIDResult.java`

Table 13.1: Parameters for Hole and Eye-chart Detection

To evaluate whether or not a victim has been identified or not, `VictimIdentification.java` passes the results into a function that returns true when:

- Largest infrared blob size > 2500
- At least one face is detected

The result object is passed to a SLAM function which then calculates the actual angle of the detected objects relative to the robot's bearing, by taking the gimbal bearing into account.

13.1.3 Implementation

This section gives brief descriptions of the individual classes as part of Victim Identification, such as their main purpose and interactions with other components.

Detection Classes in C

These classes implement the actual detection algorithms and are written in the C programming language. They make use of image processing programming and utilise the external OpenCV library. The set of all C programs are compiled separately from the rest of the project as a library, which is then imported and accessed from Java code.

`fitellipse.c, facedetect.c, squares.c`

1. Detection algorithms
2. Input: image frame from the web camera
3. Task: detect the presence of an object of interest, such as a face, an eye-chart or a hole in a box
4. Output: number of objects detected and their relative positions in the image. Also return the modified image (e.g. binary image in ellipse detection, image with a circle around the face in face detection, etc.)
5. Called by: `Vision.c, VictimIdentification.java`
6. C Header file: `vision_func.h`

Control Classes

`VictimIdentification.java`

1. Main Java class for Victim Identification
2. Interface to C code
3. Method calls for:
   (a) Grabbing webcam frames
   (b) Detection methods
   (c) Retrieving results
   (d) Retrieving modified images by each detection method

4. Output: Returns detection results, positions of detected objects and a bearing for SLAM if a potential victim is seen
    (a) Slam stores returned results in DataStore.
    (b) Results in DataStore are further sent to RobotClient via `robot.communication.VictimIDResultSender`
5. Called by: SLAM

`VictimIdentification.c`

1. Proxy class between VictimIdentification.java and vision.c
2. Interface between C and Java code through Java Native Interface (JNI)
3. Provides method declarations
4. C Header file: `VictimIdentification.h`

`Vision.c`

1. Main C class for Victim Identification
2. Webcam interface grabbing frames from the camera
3. Interface to function calls for all detection methods
4. Interface to function calls for retrieving detection results
5. C Header file: `vision.h`

`IRServer.c`

1. C class to retrieve raw image from infrared camera and send to java code
2. Interfaces to Java code through socket
3. corresponding Java class IRDevice.java

Victim Identification GUI On the client side, a graphical interface was developed in order to show the detection results from Victim Identification. These include mainly images from the web camera, modified images from the detection classes with marked victims, as well as summary statistics about detection results. An illustration of the design for the Victim Identification GUI used at the design stage is shown in Figure 13.2.

The GUI is developed as an external frame with a number of panels showing the results from each detection method. The GUI can be launched from the main client-side GUI for

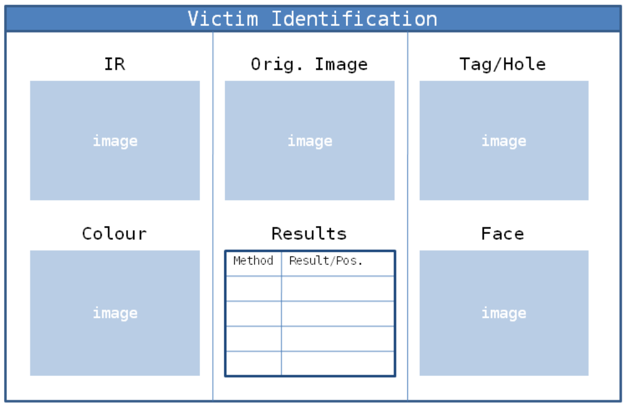


Figure 13.2: Victim Identification GUI Design

the robot and it runs on its own thread. Data from the robot is sent to the GUI in the same way as other data to the client, in the form on response objects. Response objects are processed on the client by the `ResponseProcessor` class, which also updates the relevant controls within the GUI with new information as it is received.

13.2 Detection of Eye-charts and Holes in Boxes

In the following sections, methods for detecting boxes with a round hole, as well as detection of eye-charts with a tumbling 'E', will be described. The reason for discussing these problems in the same section is that both problems were tackled using the Ellipse fitting technique. Both detection tasks are initiated in the same way and specific detection results are only determined after applying Ellipse fitting. In this section, we will concentrate on additional techniques and algorithms that have been introduced in our project to achieve Eye-chart and Hole detection. Details about the Ellipse fitting technique are discussed in the Research section.

13.2.1 Hole Detection

The problem:

Victims in the competition arena may be located in closed cardboard or wooden boxes, which have a round hole at the front. The robot might not be able to see inside the box from a distance. However, it could notice the existence of the box and move towards the hole for a better view of its contents. The task for Hole detection is to detect a round hole at the front of the box and report it to Victim Identification.

Some of the properties of the holes assumed by Hole detection are:

- The hole is round in shape
- From a distance, the inside of the hole appears darker than the outside
- The hole is located around the centre of the box, implying that there is a margin between the hole and the box edge
- Multiple holes may be observed at the same time

Approach:

The Hole detection algorithm proceeds in the following main stages:

1. Fitting ellipses
2. Contour density and ellipse size ratio checks
3. Colour inside the hole checks
4. Colour outside the hole checks
5. Detection Result

In the following sections, these stages will be discussed in turn.

1. Fitting Ellipses

Due to the shape of the hole, the technique of ellipse fitting appeared suitable for an initial analysis of the input image. As mentioned in the Research section, the hole may be seen from various angles depending on the robot's relative position to the hole. Therefore, it may appear as an ellipse rather than a circle when seen from the web camera. In order to apply ellipse fitting, we first need to pre-process the input image by finding all the contours in the image. This task is done by converting the original image to greyscale, creating a binary image based on a threshold on the amount of black colour and finally finding all contours in the binary image. In summary, the first stage of Hole detection proceeds as follows:

Algorithm 11 Ellipse Fitting

```
Given an input image, convert it to greyscale.
Create a binary image based on a threshold on the amount of black colour.
Apply a function to find contours. Contours are returned as sets of points.
for all contours do
  if num. contours points > minimum then
    Apply ellipse fitting.
    if ellipse found then
      Record ellipse.
    end if
  end if
end for
```

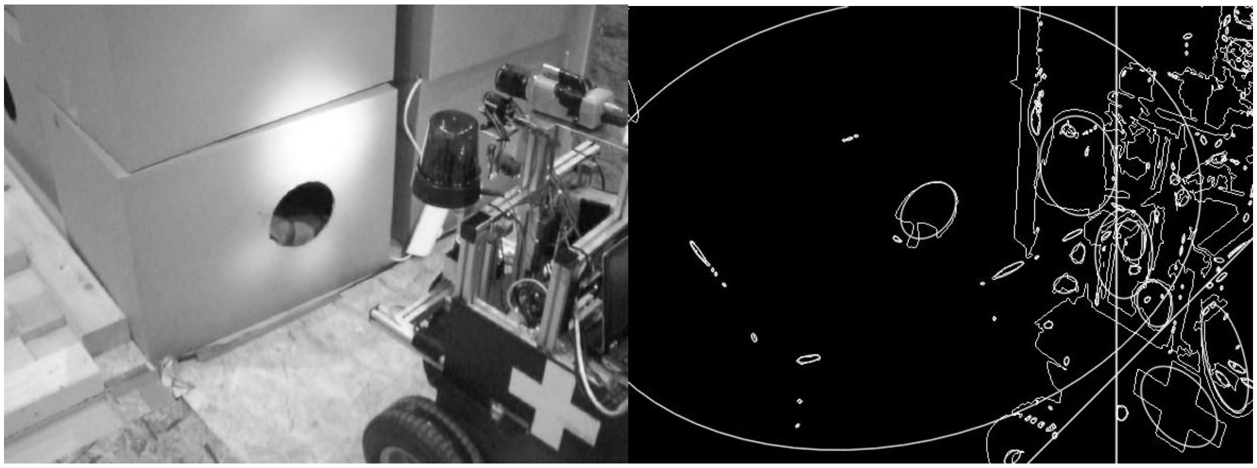
After running the above algorithm, we obtain a number of ellipses fitted to contours in the image. There may, however, be many more ellipses found than those of interest, as illustrated in Figure 13.3.

For this reason, additional steps are needed to check if an ellipse could potentially approximate the object of interest. The following stages were introduced to perform these checks in a 'candidate elimination' fashion.

2. Contour density and ellipse size ratio checks

During initial examinations of ellipse fitting results, it was observed that a high number of ellipses were fitted around relatively small and unimportant contours. A solution to this problem was to increase the minimum number of points a contour was required to have to be considered for ellipse fitting. In theory, as few as 5 points are necessary in order to fit an ellipse around them. For Hole detection, the minimum number was increased to 100, since contours of the holes were relatively long, clear, connected and consisted of hundreds of points.

Another initial observation was that many un-needed ellipses were very narrow. The approach taken to eliminate these ellipses was to calculate a ratio between the ellipses' height and width and to eliminate ellipses with a ratio below a set threshold. The ratio is calculated as



(a) Original image (b) Ellipses fitted to image contours

Figure 13.3: Application of ellipse fitting to an image

$$Ratio(w,h) = \begin{cases} \frac{w}{h} & \text{if } w \leq h \\ \frac{h}{w} & \text{if } h < w \end{cases}$$

where w and h correspond to the width and height of the ellipse. Figure 13.4 illustrates the calculation of the ratio. The ratio has a value between 0 and 1, where 0 would mean an ellipse with the width or height of 0, and 1 would be a perfect circle. The default threshold value was chosen to be 0.4, based on observations. The ratio threshold is also one of the parameters of Hole detection that may be changed externally by the user.

3. Colour inside the hole checks

This step aims to eliminate candidate ellipses which cannot correspond to a hole, based on the contrast between the colour inside and outside the hole. It is assumed that the inside of the hole is much darker than the outside than its surrounding. Since we have a binary thresholded image available to us, this contrast simply becomes binary. Now the inside of the hole will appear as mostly black, while the outside as predominantly white in colour. This fact is used in the following check, which examines the amount of black colour within the ellipse. If this amount is low, the candidate ellipse will be eliminated.

In order to analyse the colours within the ellipse, a colour histogram of the region is calculated. The region for the histogram needs to have a rectangular shape and needs to

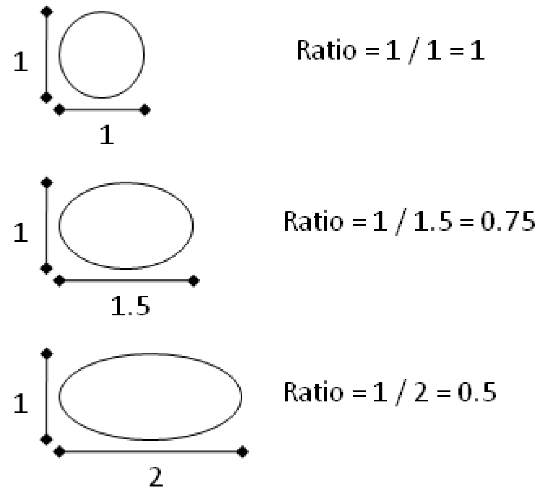


Figure 13.4: Calculation of the ellipse size ratio

be located within the ellipse. This step is done by defining a rectangle within the ellipse, which is dependant on the ellipse's dimensions. The following method is used to define the coordinates of a rectangular region within the ellipse:

$$C_{R\_in} = C_E$$
$$w_{R\_in} = \frac{max(w_E, h_E)}{2}$$
$$h_{R\_in} = \frac{max(w_E, h_E)}{2}$$

where E is an ellipse, R_in is the inner rectangular region, C is centre point, w is width and h is height. The new rectangular region is within the ellipse, as can be seen from Figure 13.5. Since we have limited the minimum size ratio of the ellipse to 0.4, even if the ellipse was rotated by 45^o, the rectangle would still be within the ellipse. If no limit was set on the size ratio, the rectangle could reach out of very narrow ellipses rotated by around 45^o.

Using the rectangular region within the ellipse, a histogram is calculated and the percentage of black colour within the region is determined. To determine the percentage of black colour, the following formula is used:

$$Bpc_{in} = \frac{Bval_{in}}{Bval_{in} + Wval_{in}}$$

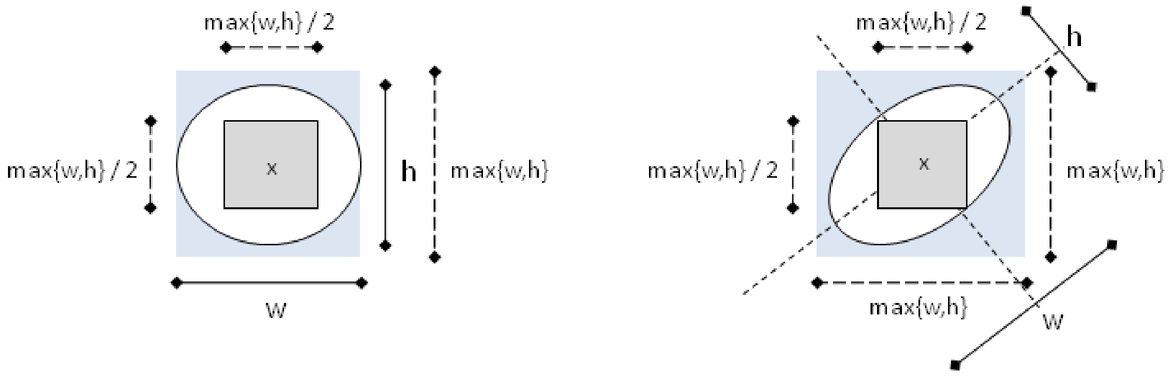


Figure 13.5: Placing a rectangle within an ellipse

where Bpc_{in} is the percentage of black colour, $Bval$ and $Wval$ are the absolute amounts of black and white colours, respectively. If Bpc_{in} is below a set threshold, the candidate ellipse is eliminated. The threshold is set by default to 90%. This leaves some tolerance for small amounts of white colour within the ellipse which could be due to places within the hole, which appear lighter and thus have become white in the binary image.

4. Colour outside the hole checks

As the next step, the area around the hole is examined. The area around the hole should be predominantly white in the binary image, due to the assumptions made about the position of the hole on the front of the box. The assumption is that there is a margin around the hole which is predominantly light in colour and thus appears white in the binary image. Again, a colour histogram may be used in this step. However, since the area within the hole is now not of interest to us, we must perform an extra step to eliminate the black colour within the ellipse, which would lead to a biased analysis of the outer area. The inner area may simply be set to white, so that only black colour outside the ellipse is considered.

Firstly, the region of the ellipse is set to white colour. Rather than only filling the inside of the ellipse to white, a bounding rectangle is drawn around the ellipse and this region is set to white. The reason for the use of a bounding rectangle lies in the way the ellipses are sometimes fitted around the hole. Some parts of the hole may stretch outside the ellipse area, and hence dark regions may appear directly outside the ellipse boundary. The use of a bounding rectangle is a potential solution for this problem and has been successful in our

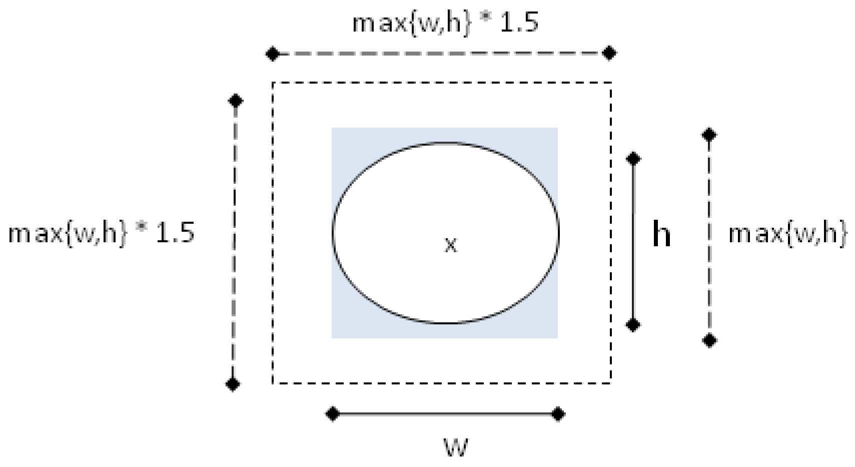


Figure 13.6: Placing a rectangle around an ellipse

observations.

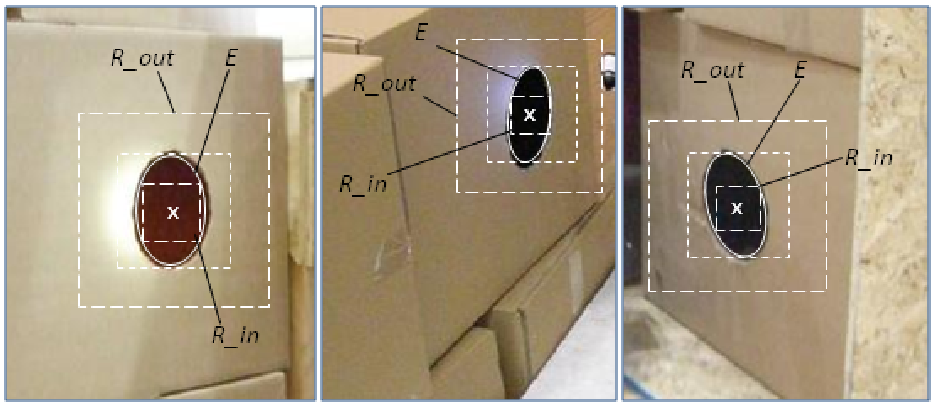
Secondly, a new rectangular region is defined outside the ellipse, which will be used to calculate the colour histogram. The following method is used to define the coordinates of the rectangular region:

$$C_{R\_out} = C_E$$
$$w_{R\_out} = max(w_E, h_E) * 1.5$$
$$h_{R\_out} = max(w_E, h_E) * 1.5$$

where E is an ellipse, R_out is the outer rectangular region, C is centre point, w is width and h is height. The new rectangular region defines a margin around the ellipse, as may be seen from Figure 13.6. This size of the margin showed the best results in our observations, since a margin too small would not provide a strong enough evidence for the elimination of false candidates and too large a margin might stretch past the boundary of the box and eliminate good candidates. Figure 13.7 illustrates the use of the outer boundary and inner rectangle in real images.

Thirdly, using the outer rectangular region, a histogram is calculated. The percentage of black colour is determined as:

$$Bpc_{out} = \frac{Bval_{out}}{Bval_{out} + Wval_{out}}$$



(a) (b) (c)

Figure 13.7: Placing inner and outer rectangles to ellipses

where Bpc_{out} is the percentage of black colour, $Bval$ and $Wval$ are the absolute amounts of black and white colours, respectively. If the percentage of black colour in the region was more than a set threshold, the candidate ellipse would be eliminated. The threshold is set by default to 4% of black colour. This leaves some tolerance for small amounts of black colour around the hole, possibly due to noise or the texture of the background.

5. Detection Result

Any candidate ellipses left at this stage are classified as a hole. The number of detected holes, as well as their positions in the image are returned. Also, the binary image and the image with contours may be retrieved for displaying to the user.

13.2.2 Eye-chart Detection

The problem:

Simulated victims in the competition arena may be accompanied with an eye-chart placed on the wall behind the victim. An eye-chart typically contains letters or symbols in rows of decreasing symbol size. The characters on the eye-chart may be either different letters of the alphabet or simply a tumbling letter 'E' printed in various rotations (by 90^o, 180^o and 270^o). The latter eye-chart type, also referred to as the "tumbling E", is found in the competition arena specification. Figure 13.8 is an example of such an eye-chart.

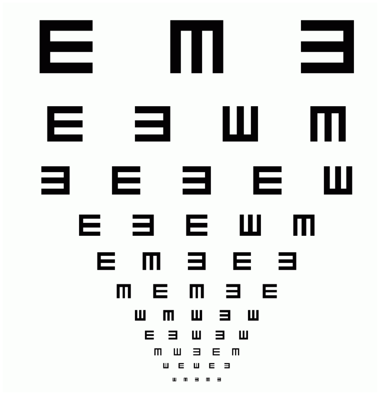


Figure 13.8: Example of an eye-chart

Approach:

Similarly to the Hole detection problem, Eye-chart detection is tackled in our project using ellipse fitting as the first step in the classification process. Since the symbols on the eye-chart are printed in dark colour on a white paper, contours may easily be found in the binary image, which is created as described in the previous section. These contours are usually very clear, with no or very little noise, meaning that ellipses can easily be fitted around the contour of each symbol. As the next step, further geometric checks can be employed to confirm or reject the presence of multiple symbols ordered in rows, which would correspond to the pattern seen in eye-charts.

In the next sections, the stages of the Eye-chart detection algorithm will be described.

1. Ellipse fitting and initial checks

At this stage, ellipses are fitted to a binary thresholded image. This stage is identical with stage 1 of the Hole detection algorithm. As in Hole detection, very narrow ellipses may be ignored since they cannot describe symbols on the eye-chart. In fact, the tumbling E symbols have a very similar width and height, resulting in nearly circle-shaped ellipses. Narrow ellipses may then be eliminated by the size ratio as described in stage 2 of Hole detection.

2. Distance-based check

An important property of eye-charts is that symbols on the eye-chart are located at regular distances from each other. This distance may differ between rows, but it remains constant between symbols in the same row. This fact becomes an important criterion when deciding whether a group of ellipses corresponds to symbols on the eye-chart. To perform this check, distances between all candidate ellipses are calculated. If a group of ellipses exists, where each ellipse has the same distance from its neighbours, it will be considered for a potentially positive classification.

Distances between a pair of ellipses is calculated by Euclidean 2-D distance as

$$d(e_1, e_2) = \sqrt{(x_{e_1} - x_{e_2})^2 + (y_{e_1} - y_{e_2})^2}$$

where $d(e_1, e_2)$ is the distance between ellipse 1 and ellipse 2, x_{e_i} and y_{e_i} are the x and y coordinates of the ellipse centres for ellipse i.

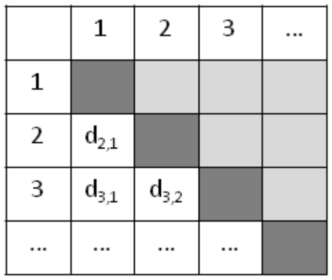
The distances between all pairs of ellipses are calculated and stored for convenience in a matrix data structure. In the matrix, cell (x, y) contains the distance between ellipse x and ellipse y. Since the distance between two ellipses is symmetric, there is no need to calculate the distances in both directions. The computation time can thus be reduced by a half. The process of calculating distances between ellipses is then given in Algorithm 12.

Algorithm 12 Pair-Wise Distance Calculation

```
for $i = 0$ to $\#ellipses - 2$ do
  for $j = i + 1$ to $\#ellipses - 1$ do
    if $i = j$ then
      continue
    end if$distances[j][i] = d(e_1, e_2)$
  end for
end for
```

The resulting matrix looks as illustrated in Figure 13.9.

Next, the distances in the matrix are investigated in order to find 3 or more ellipses of the same or nearly same distance. As the distance between symbols in the eye-chart is not known beforehand, a nested loop needs to investigate each ellipse for whether it has 2 neighbouring ellipses at the same distance. If such a case is found, the 3 ellipses become candidates for a positive classification and are subjected to further checks. To allow for a slight displacement of the ellipses, a tolerance factor for the distance, ϵ_d is introduced. This factor is also a parameter for Eye-chart detection that may be changed by the user.



	1	2	3	...
1	■	░	░	░
2	$d_{2,1}$	■	░	░
3	$d_{3,1}$	$d_{3,2}$	■	░
...	...	...	...	■

Figure 13.9: The distances matrix

3. Angle-based check

At this stage, there is a triple of ellipses equidistant from the middle ellipse. However, it is not yet confirmed whether the ellipses are located on the same line, as the symbols are located in rows on the eye-chart. It is possible that they are in different parts of the image and purely happen to have the same distance from the middle ellipse. For this reason, a check based on the angle between the 3 ellipses is performed. If a, b, c are the 3 ellipses' centres, b being the middle ellipse and a, c the outlying ellipses, the angle is measured between a and c with b as the angle centre. The angle is computed from the normalised dot-product of the 2 vectors $\vec{ab}$ and $\vec{cb}$:

$$dot(\vec{ab}, \vec{cb}) = \binom{x_{\vec{ab}}}{y_{\vec{ab}}} * \binom{x_{\vec{cb}}}{y_{\vec{cb}}} = x_{\vec{ab}} * x_{\vec{cb}} + y_{\vec{ab}} * y_{\vec{cb}}$$

The angle between the 2 vectors should be around 180^o, meaning that the dot product should be approximately -1. To allow for a slight displacement of the ellipses, the dot product may vary by 0.03, corresponding to +/- 14^o. This tolerance factor, ϵ_a is an external parameter for Eye-chart detection.

Figures 13.10 and 13.11 illustrate the distance-based and angle-based checks between the 3 ellipses. The first figure shows the ideal situation and the latter shows the incorporation of tolerance factors in the checking process.

The overall algorithm to perform the distance-based and angle-based checks on a high level is given in Algorithm 13.

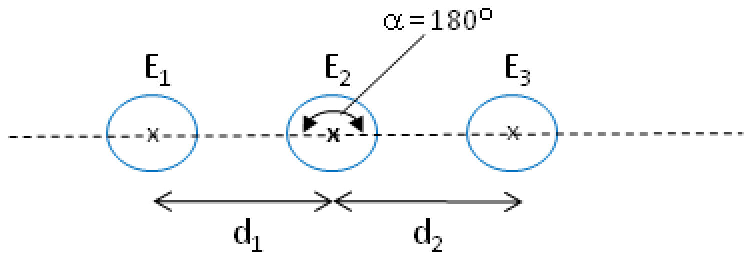


Figure 13.10: Distance and angle checks during eye-chart detection

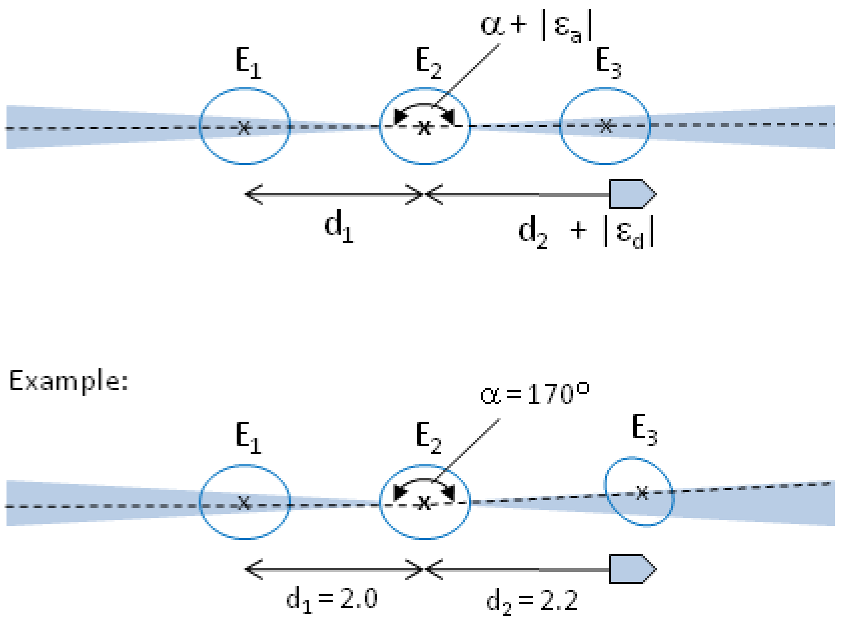


Figure 13.11: Including a tolerance factor for distance and angle checks

Algorithm 13 Distance-based and angle-based checks in Eye-chart detection

```
for $b = 0$ to $\#ellipses - 1$ do
  for $a = 0$ to $\#ellipses - 1$ do
    if $a = b$ then
      continue
    end if
    $tmp\_dist \leftarrow distances[a][b]$
    for $c = a + 1$ to $\#ellipses - 1$ do
      if $a = c$ or $b = c$ then
        continue
      end if
      if $|distances[b][c] - tmp\_dist| < \epsilon_d$ then
        $dot \leftarrow dot(\vec{ab}, \vec{cb})$
        if $|1 + dot| < \epsilon_a$ then
          {Positive classification!}
          break
        end if
      end if
    end for
  end for
end for
```

4. Detection Result

If there is a positive classification during the last stage of checks, the result success and the positions of the Eye-chart ellipses in the image are returned. Also, the binary image and the image with contours may be retrieved for displaying to the user.

Discussion: Alternative solutions

During the design and implementation of the Eye-chart detection algorithm, alternative ways to confirm if ellipses belong to an eye-chart have been considered. Initially, finding a line between two ellipses and checking if any other ellipse lies on or near that line was considered. However, this approach was found to be more tedious than the angle-based check, which essentially serves the same purpose while allowing control of the relative angle (or indirectly the relative distance) by which a third ellipse is allowed to divert from the straight line.

incorporating the size of the ellipses into the checking process was also considered. In this way, pairs of ellipses of greatly varying sizes would be discarded, since the symbols on the eye-chart have the same size in each row. This process would, however, involve additional computation time and its benefit in the classification process would be more for disqualification purposes. The distance-based and angle-based checks would still need to be performed to confirm a positive classification. In fact, it was observed that the two checks were sufficient to eliminate any false candidates and since no false positive classifications were observed with the existing algorithm, the size-based check would be unnecessary.

13.2.3 Implementation

The Hole and Eye-chart detection algorithms are implemented in the C language. Since both methods use ellipse fitting and share a large part of the initial computational tasks, they are located in the same file, `ellipse.c`. The main program flow starts with the ellipse fitting task and continues on to Hole detection within the same algorithm. Candidate ellipses for Eye-chart detection are stored in a data structure for later examination. After the Hole detection algorithm finishes, the program proceeds to Eye-chart detection, using the stored candidate ellipses. Figure 13.12 illustrates the high level flow of the program.

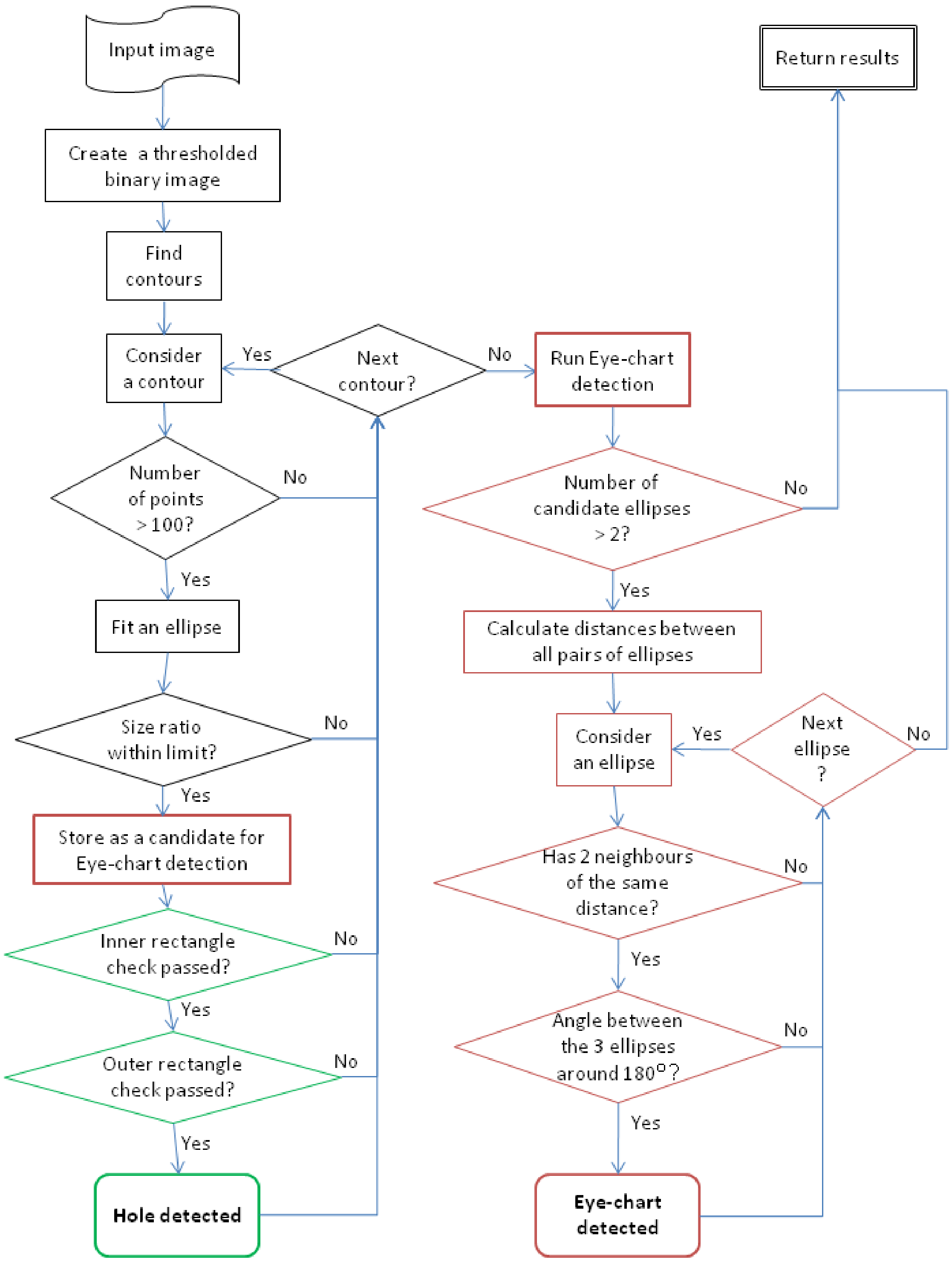

Input image
Create a thresholded binary image
Find contours
Consider a contour
Yes
Next contour?
No
Run Eye-chart detection
Return results
Number of points > 100?
No
Yes
Fit an ellipse
Size ratio within limit?
No
Yes
Store as a candidate for Eye-chart detection
Inner rectangle check passed?
No
Yes
Outer rectangle check passed?
No
Yes
Hole detected
Number of candidate ellipses > 2?
No
Yes
Calculate distances between all pairs of ellipses
Consider an ellipse
Yes
Next ellipse ?
No
Has 2 neighbours of the same distance?
No
Yes
Angle between the 3 ellipses around 180°?
No
Yes
Eye-chart detected

Figure 13.12: Flowchart of Hole and Eye-chart detection

Parameters:

A number of parameters may be specified for Hole and Eye-chart detection, as mentioned in the descriptions of both algorithms. The user may access a configuration file, `vision.conf`, located in the robot's root folder, to review and change the following parameters:

Parameter	*Range*	*Default*	*Description*
thresh_val	0-255	70	Value of threshold to create a binary image
ell_size_ratio	0-1	0.4	Ratio of the sizes of the ellipse. Value between 0 and 1.
inner_blk_pct	0-1	0.9	Min. percentage of black colour within the ellipse
outer_blk_pct	0-1	0.04	Max. percentage of black colour around the ellipse area
tag_dist_tol	≥ 0	2.0	Tolerance factor for distance variation between 3 ellipses (ϵ_d)
tag_angle_tol	0-1	0.03	Tolerance factor for angle variation between 3 ellipses (ϵ_a) $0 \Rightarrow$ no tolerance, $1 \Rightarrow 90^o$

Table 13.2: Parameters for Hole and Eye-chart Detection

13.3 Face Detection

13.3.1 Overview

The aim of this module is to detect faces of simulated victims. In the RoboCup Rescue competition, dolls are used to simulate victims, so a method for detecting doll faces was needed. As in common face detection, the required module would analyse a given image, identify any faces and return their positions in the image.

As described in the Research section, it was decided to use a Haar classifier based on the Viola-Jones method for the face detection function. Essentially, a Haar classifier can be built separately using training and testing image samples and may be used independently of any platform or programming language. In this project, it was decided to build a Haar classifier and use it from within the Face Detection module. On a high level, the Face Detection module works as shown in Algorithm 14.

Algorithm 14 Face Detection

```
Haar classifier is loaded into memory
Image is converted to greyscale
Equalise the histogram of the image, in order to normalise the brightness and increase the contrast in the image
Apply the Haar classifier
for all detected faces do
    Return its position in the image
    Draw a circle around the region of interest
end for
Return the relative horizontal position in +/- percent, of the largest face in the image
```

13.3.2 Building the Classifier

The process of building the Haar classifier consisted of the following stages: (1) collection of positive sample images, (2) data preparation, (3) collection of negative images, and (4) training the classifier. In general, the recommended procedure in building the classifier as suggested in [25] was followed. In the following sections, each of the stages in building the classifier will be described.

1. Training dataset - Positive images

As the first stage, suitable training data had to be collected for classifier training. In our case, baby dolls would be used as simulated victims and were to be detected by our classifier. There was little material containing images of the simulated victims from the data available from past competitions. For this reason, the method of positive sample collection was by internet research.

Our main target were images:

- Depicting the head of a doll
- Potentially containing other body parts of the doll
- The face was shown from the front, or only turned by a small angle
- The face was shown upright, or only turned by a small angle in a clockwise/anticlockwise direction

- The face was visible and clear of any additional objects such as glasses, hands, etc.
- A variety of shades and types of light in different images was preferred

As the main source, websites of online shops and auctions were used. Many of the websites provided convenient search facilities to enable a systematic search process. By the end of the search process, around 190 positive images were collected.

In addition to that, the dataset was enhanced with a number of pictures taken using a real doll, which had been obtained for the purposes of this project. The doll was photographed in different light settings and angles relative to the camera. From these pictures, 11 were included in the dataset.

Overall, the training dataset consisted of the following positive images:

- Images collected from online resources: 190
- Images created using a doll: 11
- Images from past RoboCup competitions: 2
- Total positive images: 203

2. Data preparation

After collecting positive images, the dataset needed to be normalised in order to provide consistent input for the training program. This stage mainly involved image cropping, which was needed so that the input images would only contain the region of interest in the image, in this case the face of the doll. In this way, any unnecessary information in the image, such as hair style and colour, would be ignored. Figure 13.13 shows an example of image cropping used.

In an early stage of data preparation, the region of interest used for cropping extended over the whole head of the doll. This approach created some problems, such as the differences in hair style and colour, which would lead to a bias and potential errors in the classifier. Also the colour of the background around the head differed dramatically. As a result, it was soon realised that the region of interest should only contain the face itself.

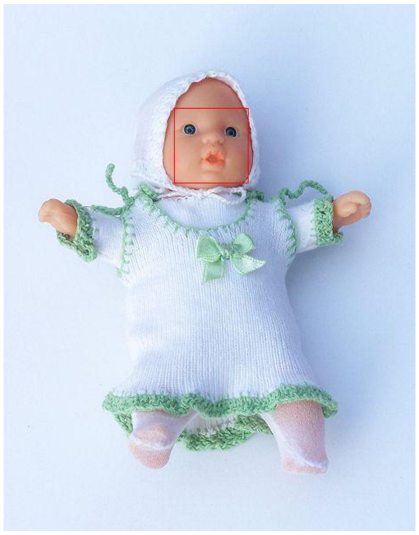


Figure 13.13: Data preparation: image cropping to only keep the region of interest

3. Negative images

As well as positive images, negative images needed to be collected to provide negative samples to the training program. These images could essentially depict any object other than a doll's face. The choice of negative images, however, would affect the classifier built using these samples. For example, a high number of images of objects found in the competition arenas, such as walls, corners, robot parts, etc., would decrease the probability of false classifications in these environments. For this reason, a high proportion of the negative images was taken from materials about past competitions.

In the real application, the robot would be using a web camera as image input. Due to the nature of typical views seen using a web camera, which are usually parts or sections of the whole view as seen by a human eye, original negative images were also cropped to contain parts of objects. Also, sections of walls and corridors were selected, which would resemble typical views as seen with the web camera. This part became especially useful during the building of the classifier, as early versions of the classifier made false detections on walls and other plain textures. For this reason, the number of negative images was increased with wall segments having wooden or painted textures.

In total, the number of negative images used was 120.

4. Creating training samples

This stage involved converting both positive and negative images in the dataset into a format as required by the Haar training program. The training program requires all positive samples to be of the same size and organised in a specific folder structure. A utility program in OpenCV, `createsamples`, could be used to automatically resize the positive images. Kuranov et. al. [15] suggest that using the image size of 20x20 pixels produced the best achieving classifiers. Therefore, we have followed the same practice and used 20x20 as the size for our positive images.

A 'vector' file was also produced for both positive and negative samples using the same `createsamples` utility. The vector file would be used as the main image descriptor for the training program, providing detailed information about the images, their locations, as well as information about their dimensions

5. Training the classifier

At this stage, the input data for building the classifier was ready. The actual process of training the classifier was done using the `haartraining` program as part of OpenCV. The program takes the vector files as input, together with parameters used for training and performs the training process. With regards to the parameters used, they include:
- Number of positive and negative samples: these were 203 and 120, respectively.
- Number of training stages: 20 stages were used, based on the example in [15].
- Other parameters were set to their default values, such as 'number of splits' = 2, 'minimum hit rate' = 0.995, 'maximum false alarms' = 0.5 and 'weight trimming' = 0.95.

The training was performed remotely on a machine in the Department of Computer Science at Warwick. Remote access was chosen due to the anticipated long running time of the training program. [25] reports that training took several days when using thousands of training images. In our case, training took around 8 hours.

After training was finished, output of the training program was converted to XML. The resulting XML file would contain the description of the newly produced Haar classifier.

6. Initial tests and second training iteration

Initial tests with the new classifier showed promising results when tested on the training images. However, when tested with live camera input, false detections would occur, often in places such as walls or other texture-like areas. It was found that very few of the negative images in the training dataset contained texture-like features. For this reason, a higher number of negative images were chosen to be images of walls and boxes, such as from past competition arenas, as well as manually made images from a web camera used in a room.

With the negative samples updated, based on the observations of the first classifier, a new training was conducted. The second version of the classifier was trained after around 8 hours. The second version was again tested with a web camera and its performance was found to be better than of the first version. For this reason, the second version was selected to be used in the Face detection module of Victim Identification.

13.3.3 Implementation

As part of the overall Victim Identification component, Face detection is implemented in the C language. The code is located in `facedetect.c` and is responsible for operating the Haar classifier, loading an input image, performing minor pre-processing tasks, calling the classifier to detect any faces in the image and returning the detection results.

Parameters:

The face detection program uses an external XML file with the Haar classifier definition, `dollface.xml`. The XML file is located in the same folder as the C code for face detection, "robot/vision".

13.4 Square Detection

As with the other Victim Identification image processing modules, the square detection process will be implemented using the OpenCV library for C. Many of the functions and algorithms mentioned in the research section for this module have been implemented in OpenCV, though others not at all.All the primary image processing functions mentioned have been implemented. The square detection algorithm is supplied with an image from the webcam on top of the robot; the algorithm then detects the squares of interest within this image and returns a list of positions indicating where these squares are. The figure 13.14 outlines this process in a rough flowchart and each part will be explained in more detail.

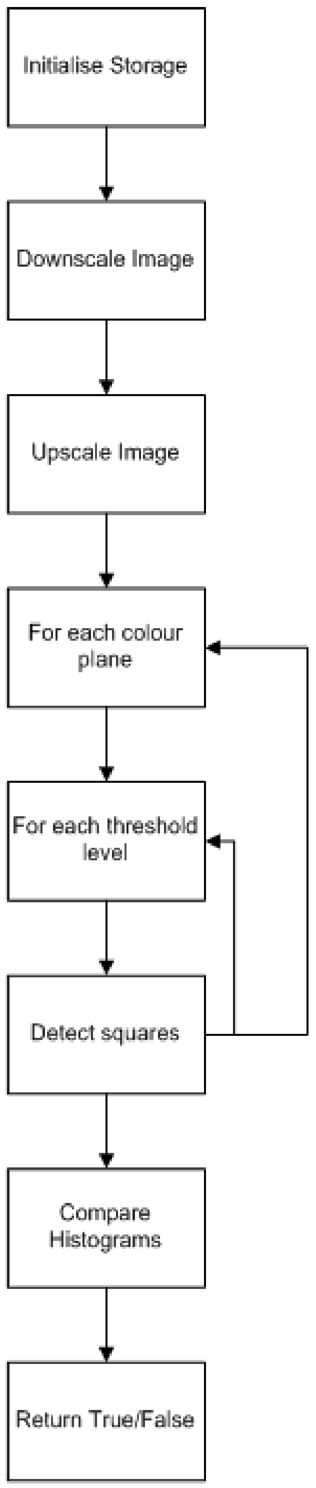


Figure 13.14: Flowchart of Square Detection

13.4.1 Initialise Storage

In OpenCV it is important to declare and initialise all storage, especially to hold the many images that will be needed. Image storage is needed for

- Input image
- Greyscale images
- Temporary scaling images
- Sequence of detected squares

13.4.2 Downscale Image

OpenCV provides the inbuilt function `cvPyrDown`.

13.4.3 Upscale Image

OpenCV also provides the inbuilt function `cvPyrUp`.

13.4.4 Colour Planes

Our images will be using 3 colour planes (RGB) therefore we need to extract them one at a time. This can be done using the OpenCV function cvSetImageCOI which is able to extract each plane in turn.

13.4.5 Threshold Level & Detect Squares

To give as great a chance as possible to detect squares, several thresholded levels between 0 and 255 can be used. For threshold level 0, the Canny algorithm is used which is implemented in OpenCV as `cvCanny`. This function takes as input two threshold levels which can be modified with testing.

The binary image of edges then needs to be dilated using OpenCV's `cvDilate`. For other threshold values `cvThreshold` instead of `cvCanny` can be used.

The threshold type will be `CV_THRESH_BINARY` to output a binary image. The maximum value for the threshold will always be 255, but the lower threshold can be set between 1 and 255 to give as great a chance to detect squares as possible.

After the binary image is obtained, `cvFindContours` can be used to find connected segments in the image. This method returns a list of edges pointed to by `first_contour`.

After this, it is possible to go through this list and use `cvApproxPoly` with `CV_POLY_APPROX_DP` as the method of approximation. This method implements the Douglas-Peucker algorithm mentioned in the research section, this algorithm turns a sequence of connected segments into a list of minimal vertices that approximates those segments. This returns a sequence of polygons which can then be checked to find out if they are squares or not. To check whether they are squares, it can be checked whether the number of vertices is equal to four and whether or not the contour is convex. It is also possible to to discard squares that are too big or too small. A square will need to be at most 70% of the image and consist of at least 1000 pixels.

After the contour has passed this test the inner angles of the contour can be calculated by returning the smallest angle between each pair of adjacent edges. Edges are stored in a `cvSeq` of vertices, an OpenCV type of sequence. This can be traversed and for each 3-tuple of vertices the angle can be calculated with the OpenCV method `angle()`.

Then each angle can be checked to see if it is 90 degrees. If so then the contour is accepted as a square.

13.4.6 Compare Histograms

A colour histogram of each square can then be created by setting the region of interest within the image to the area defined by the square using `cvSetImageROI` and then using `cvCreateHist` to create the colour histogram of that image. This can then be compared to an array of pre-calculated histograms using `cvCompareHist` and the Chi-square test. This outputs a value that represents whether or not the square can be positively contribute to identifying a victim.

13.5 Infrared Blob Detection

There are several infrared components that will be needed for blob detection to work

- `Robot.hardware.IRDevice` : Collects the infrared raw data from the image server and generates a BufferedImage which is put into the `datastore`

- `Robot.vision.IRBinaryServer` : Starts up `BinaryDevice` and is used for sending binary image data)
- `Robot.vision.BinaryDevice` : Converts the `BufferedImage` in the `datastore` to a binary Image
- `Robot.vision.DetectBlobs` : Runs blob detection on the binary image
- `Robot.vision.Binary` : Class for storing the binary Image
- `RobotLibrary.support.Blobs` : Class for holding blob victim ID data for sending to client

13.5.1 IRDevice

This classes function is to retrieve the raw infrared image data from the OpenCV image server, turn this data to a `BufferedImage` and store it in the `datastore`. `IRDevice` creates a socket and input and output streams for the socket to receive the image data. This data is stored as an array of fixed length and is transformed into a `BufferedImage` by going through each 3-tuple in the array (i.e each pixel). The class then transforms the 3-tuple into a pixel colour for the corresponding pixel in the `BufferedImage`.

13.5.2 IRBinaryServer

This class will be started up from the main robot class, it can be described as the hub of infrared binary communication between the robot and the client. When the class is instantiated it will create a new `BinaryDevice` thread. Then as `IRBinaryServer` runs it will continually await incoming connections on a socket, when a connection is made a new `IRBinaryCameraClient` thread is created and added to an internal list. The `BinaryDevice` created earlier will call `IRBinaryCameraClient`'s `SendImageData` method to send a raw binary image to all clients stored in the internal client list.

13.5.3 BinaryDevice

This class is created and run from `IRBinaryServer`. Its function is to transform the stored infrared image in `datastore` to a binary image. It does this using Algorithm 15. Once the binary image is created, it can be stored in `datastore` and also sent to clients through the parent class, `IRBinaryServer`.

Algorithm 15 Convert To Binary

```
Require: image
  Apply down-sampling and then up-sampling steps of Gaussian pyramid decomposition
  for each pixel do
    if pixel.red *a + pixel.blue *b +pixel.green *c > T then
      Binary.pixel = 1
    else
      Binary.pixel = 0
    end if
  end for
```

13.5.4 DetectBlobs

This class is the main functional class for blob detection. `DetectBlobs` provides methods to retrieve a `Blob` array from the latest infrared image and also return the largest blob detected. The `DetBlobs()` method gets the latest binary image from the `datastore` and then runs the scanline flood fill based algorithm upon the image. The `Blob` class is a serialisable class (so it can be sent to the Victim Identification GUI on the client), that holds all data on a blob, i.e its size and location.

Chapter 14

Devices and Porting

14.1 Serial Devices

Many of the devices the robot would be using, such as the LIDAR Scanner and the Motor Control boards, use serial interfaces. In order to save on code repetition, a class which implemented many useful methods, such as connection handling and data sending and receiving, was created based upon a similar class produced by WMR team some years ago. Minor modifications were required to make the connection methods more stable, however the majority of the class was untouched.

A recurring issue was attempting to deduce which port names were assigned to which devices. In the majority of cases, the `udev` and `udevadm` commands proved invaluable for determining device information from port names. In a few particularly annoying cases, trial and error by connecting and disconnecting devices and observing changes to the port list sufficed.

14.2 Phidget Devices

As well as the Serial devices, a Phidget board [2] was used on the robot to control data from the SONAR sensors and from the CO_2 sensor. This board provided an easy to use API with a single call to connect to the device and a simple event-driven technique for retrieving data from the devices.

Most data can be read from the board directly. However, the CO_2 sensor requires a warm-up time, after which it begins reading a constant value. A detection of CO_2 then corresponds to a drop in this value. In order to turn this information into a useful boolean detection / no-detection value, the system first determines the constant output of the device by taking an average of the first 100 readings. Subsequent readings can then be subtracted from this average to give a value which increases when CO_2 is detected. A threshold value can be set so that the system can automatically detect when the readings from the device change to signify the presence of CO_2 in the environment.

14.3 Motion

All robot motion is carried out by the *MotionController* class, which is an abstraction of the *Motor* device class. It includes functions which allow the user to ask the robot to move forwards through a certain number of meters, or to turn through a number of degrees. *MotionController* then determines the speed at which the robot should move and the duration of movement. How it does this shall be discussed in the next section.

The *Motor* device talks directly to the AX3500 motor control board, which in turn interfaces directly with the two motors, with capability for independent control of each motor. It accepts commands in the range -127 to +127, where 127 is the maximum speed in either the positive (forwards) direction or negative (backwards) direction, and 0 is stationary. Thus, the motors can travel a certain distance at a percentage of their maximum speed by sending a value in this range to each motor, and then at some time later sending a command with value 0. One feature to note here is that the design of the robot is such that one motor faces the opposite direction to the other. A result of this is that one motor operates more efficiently than the other, since motors are more efficient moving forwards than backwards. The motor control board can compensate for this by applying a scaling factor to the commands based upon their relative efficiency. One of the tasks that was performed in porting the code to the robot was to adjust this compensation value through trial and error, moving the robot over long distances and observing how it changed its path.

The motor control board allows us to operate the robot in a 'Closed Loop Speed Control' mode, which uses a feedback loop from encoders on the motors to ensure that the robot travels at a certain constant speed. This is particularly useful in our situation, since it allows us to know with some accuracy how quickly the robot will be moving. It is with this knowledge that we can calculate the timings for moving.

14.3.1 Moving Forwards

Straight-line motion is achieved simply through using $velocity = distance/time$ to calculate the time the robot must travel at a given speed to move the desired distance. The speed is set to be a fraction of the motor's maximum speed, around 50% or so, since the robot need not move too fast, and more accurate speed control can be obtained at lower speeds since the effects of acceleration and deceleration are minimised. The motor's maximum speed can be calculated by knowing its maximum revolutions per minute and the distance the tracks move in one rotation - the circumference of the rotor to which the motor is attached. Since the gearing on the motors is complex, it was not known accurately what the maximum RPM would be, and so an estimate through timing and counting was taken, which provided results accurate to around 1cm error over a movement of 1m; small enough that the EKF should be able to minimise those errors with ease.

14.3.2 Turning

Turning proved to be more of a challenge for the robot, largely as a result of its twin-tracked design. In order for these to turn, one motor must be move in the opposite direction to the other, causing an amount of slippage between the tracks and the surface, which causes the robot to turn. This slippage is highly variable from surface to surface, and is further complicated by factors such as debris which can affect slippage non-uniformly. A dead-reckoning approach to turning, while perhaps accurate on one surface, will thus require re-calibration for each new surface the robot encounters, and it will not account for non-uniform slippage in the driving surface. Thus, solutions which were surface-independent were sought.

Compass An Ocean Server 5000 tilt sensor and compass is mounted on the robot, largely with an aim to providing tilt information for when the robot encounters slopes. Since it also contains a compass, a feedback loop was created so that `MotionController` can know through how much the robot has turned, and stop the robot when it has turned the required amount. While in theory this was a good idea, numerous issues were encountered with the compass with regards to obtaining accurate measurements. This was due to the magnetic fields created by the motors, fans and other electronics inside the robot. After much testing, the compass was mounted on a carbon-fibre pole extending above the robot, to minimise the interference from any of the robot's components. This alleviated many problems and meant that the bearing estimates provided by the compass were more accurate. However, these estimates had to be improved through the use of Scan Matching, since there would still be approximately 10-20 degrees error in the worst case from using the compass alone.

Matching Scans A second feasible solution to this problem that is facilitated by the regular walls of the environment is that the robot can determine how far it has turned through matching the positions of walls throughout the turn. This is relatively easy to do since wall-extraction code already exists. In order to do this, the algorithm must match walls between one scan and the next accurately. These matchings must be computed efficiently, so that the frequency of the feedback loop is high enough to maintain turning accuracy. This matching algorithm must take into account the fact that as the robot turns, it will see less of some walls and more of others. The entire Scan Matching algorithm can be found in Algorithm 16.

It works by first rotating the original scan by the estimated bearing change and extracting wall maps from both scans. It then attempts to produce a matching of these walls using their relative bearings and the distance between endpoints as metrics. From these matches, the algorithm can produce a more accurate value for the change in bearing by taking an average of the difference in bearings between the walls it has matched. Throughout the algorithm, the fact that the walls rotate in the opposite direction to the robot is accounted for.

Algorithm 16 Scan Matching for Turning Estimation

```
Require: InitialScan, FinalScan, RotationEstimate
  InitialScan.rotate(RotationEstimate)
  InitialWalls $\leftarrow$ Ransac.extract(InitialScan)
  FinalWalls $\leftarrow$ Ransac.extract(FinalScan)
  Matches $\leftarrow$ Match(InitialWalls, FinalWalls)
  BearingChange $\leftarrow$ 0
  for all Match $\in$ Matches do
    BearingChange $\leftarrow$ BearingChange + (Match.FinalWall.bearing $-$ Match.InitialWall.bearing)
  end for
  return RotationEstimate $-$ BearingChange
```

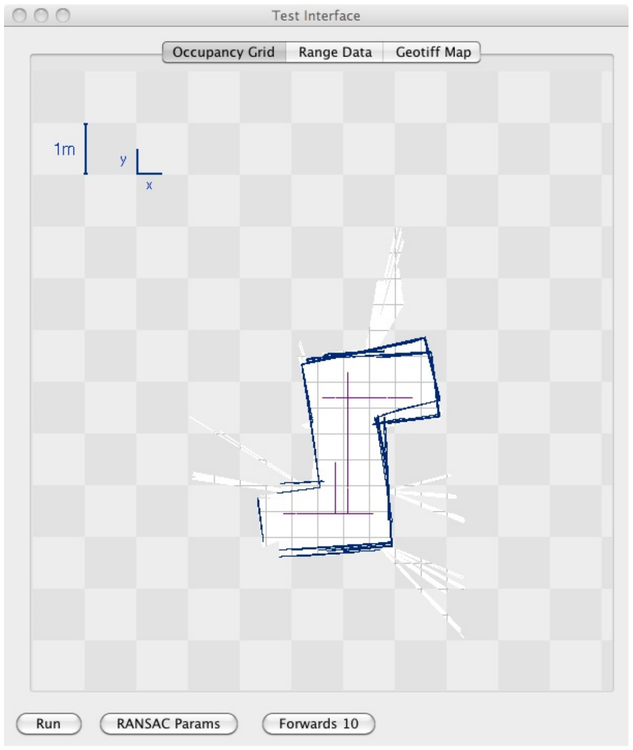


Figure 14.1: The raw map produced using only odometry estimates. Note the angles of walls, which should be perpendicular, are skewed by inaccurate position estimates.

Although crude, the scan matching algorithm was able to clean up some of the test data retrieved from the testing arena. In Figure 14.1, we see the raw map produced by using only odometry estimates. The turns should all be 90 degrees, the errors in odometry can be clearly seen since the walls are not aligned properly. In Figure 14.2, we can see how by applying Scan Matching, the map can be significantly improved as a result of the reduction in error from the odometry estimates.

Although the algorithm produced good results when applied to turning, linear translations such as moving forwards or backwards could not be estimated accurately. This feature would be very useful indeed, since it would mean the SLAM algorithms no longer relied upon the inherently inaccurate odometry data for localisation estimation. In the *Future Work* section, we will discuss possible extensions which would allow for this.

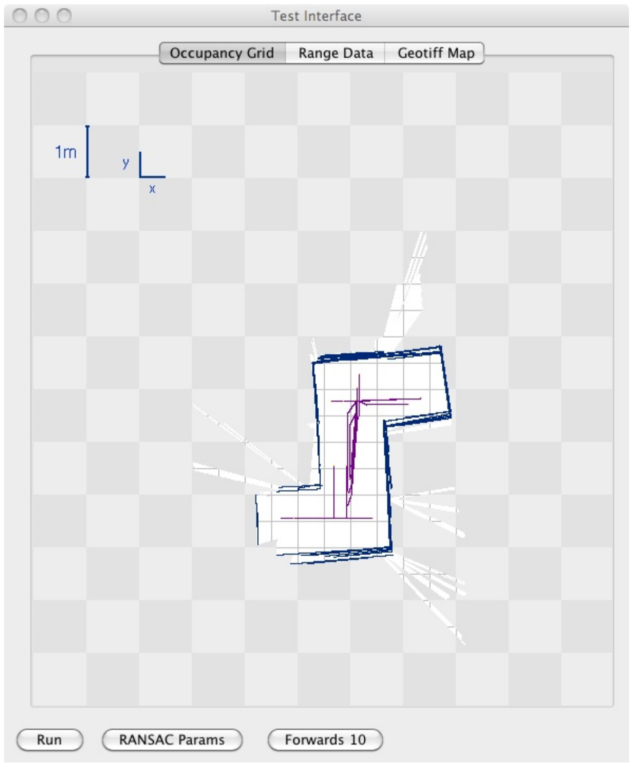


Figure 14.2: The raw map produced with Scan Matching applied on the turns of the robot. Note how the walls are much more aligned than in the odometry based map.

14.3.3 180 Degree Turns

As a result of an uneven weight distribution, the robot is unable to rotate on a fixed point in the centre of the footprint. This means that when the robot turns, some motion forwards or backwards is expected. From the point of view of mapping, this is catered for in the error minimisation, however it does pose problems when the robot is placed in tight spaces, since it may not have enough room to turn around. In order to allow the robot to turn in these environments, collision detection around the sides of the robot was implemented using the LIDAR scanner. If the robot detects a collision in the direction it is turning, it can move backwards, using a rear-facing SONAR to ensure no collisions occur, and then continue to turn.

14.3.4 Avoiding Collisions

Although the route planning algorithm contain checks to ensure the robot avoids collisions, it was useful to add low-level collision avoidance into the motion controller, since the estimated location of the robot may not be accurate and as such obstacles may be present on the robot's route. Low-level collision avoidance was implemented using two sensors: the LIDAR scanner and a rear-mounted SONAR sensor. The LIDAR scanner, with its wide field of view and high refresh rate, allowed the robot to check for collisions both in front and to the sides, so that obstacles were avoided in both straight line motion and during turns. The rear-mounted SONAR allowed the robot to ensure that it didn't drive into walls whilst turning on slopes, since the robot would tend to slide down slopes whilst rotating. These tests were implemented during a loop which executed whenever the robot was in motion.

Chapter 15

Testing

15.1 Slam Simulation

To allow the team to test the SLAM algorithms without the need of using the robot, LIDAR scans were logged using the robot in the test arena. The scans were taken at regular intervals of 25cm movements using manual control of the robot. This allowed representative data to be taken, since the motions would have the same inaccuracies as when the robot is running fully autonomously. These scans were used to create a "SLAM Simulation", which mimicked all the actions of SLAM except it would use the logged LIDAR scans and execute the motions taken when logging the data. This was incorporated into a test GUI, so that the results of the algorithms could be seen in real time. This allowed us to test algorithms such as Scan Matching, JCBB and the EKF using realistic data, but without needing to consider battery life or access to the robot. Screenshots of this SLAM simulation can be seen in the diagrams from the *Planning* and *Porting* sections, Figures 14.2, 14.1, 10.1 and 10.2.

15.2 Feature Extraction

Testing of feature extraction was performed on the RANSAC algorithm first, and then on the geometric post-processing algorithm.

The logical structure of the RANSAC algorithm was tested by verbose output to the console, to ensure correct broad operation. Then using a few small sets of example points, with varying levels of noise and numbers of outliers, extensive test output was produced to be checked by eye. Defining a formal/mathematical testing procedure would probably have been infeasible due to the qualitative nature of the correctness requirements, and certainly would not have been worth the time necessary to implement. Initially the output would vary wildly from one trial to the next running the same scan, and useless long strings of small walls were common, and much work on optimising algorithm parameters was required to reduce this variation.

Once it became apparent that further parameter tweaking was yielding very limited results, the geometric post-processing algorithm was developed and tested. This component was tested using sample output from the first RANSAC testing phase, again analysing the processed output by eye to determine optimal threshold parameters for the joining and merging of walls. The algorithm was then tested in the test arena.

15.3 Data Association

As with most of the SLAM algorithms, Data Association was tested using the SLAM Simulation interface. At each iteration as the map was built, the JCBB algorithm would be run on the latest set of observations against the current `WorldState` object. With a verbose output, the hypothesis could be checked by hand to ensure that the associations were correct, and that no spurious pairings were present. At first, the hypotheses contained erroneous pairings, which were due to incorrect parameters and a result of the way Java handles passing objects between functions. This was due to the recursive nature of the algorithm whereby it passes `Hypothesis` objects recursively so that they may be built incrementally. Because of the way Java passes objects by reference, all the hypotheses had to be cloned into new objects before they could be sent down the recursion tree, so as the ensure that each branch of the tree dealt with only its own hypothesis and no interference occurred. This added complexity onto the algorithm, since within each hypothesis all pairings and features must also be cloned. Unfortunately, without changing to a language that supports passing objects by value, this problem could only be solved by adding computational overhead. As well as this, parameter tweaks for both the bearing and distance confidence were required to ensure the algorithm performed correctly.

15.4 State Estimation

Testing of state estimation was performed on the EKF first and then on the hypothesis application wrapper.

The mathematical model derivations in the EKF could be checked using a spreadsheet with numerous test cases, but the partial derivations for the various required Jacobians had to be checked purely by hand. This testing stage was extensive to make ensure no mistakes were present. Particular problems including handling singularities in the model equations and loss of orientation information when computing angles. There were also issues with rounding errors when performing calculations in the Java language. A few example scenarios were produced manually to examine the ability of the EKF to compensate for odometry error using corner observations. For some models it was possible to compare the numeric results of covariance computation to known working EKF implementation in flat 2D.

The logical structure of the hypothesis application method was tested using verbose console output, to ensure the algorithm was proceeding in the correct manner and to isolate and fix exceptions. Tests were performed with manually created wall maps and hypotheses at first, with the conversion procedure for each endpoint observation group studied for any clearly erroneous output. The system was then tested in the test arena, on manual navigation control initially.

15.5 Planning

The main technique for testing planning was to use the Slam Simulation interface and ensure that at each point of the simulation, the planning algorithm would produce a sensible path. Figures 15.1 and 15.2 show examples of the planning algorithm in the Slam Simulation interface. Once the algorithms had been refined, testing was carried out on the robot. Initially, the robot was placed in a relatively clear environment, with very basic mapping capabilities, and was set off to observe which paths it took. These tests also ensured that the robot's collision avoidance was robust, since the basic mapping meant that the robot would often plan paths that took it towards an obstacle. Once collision avoidance and path planning were verified, the robot was placed in the test arena, and its motions observed in competition-esque settings.

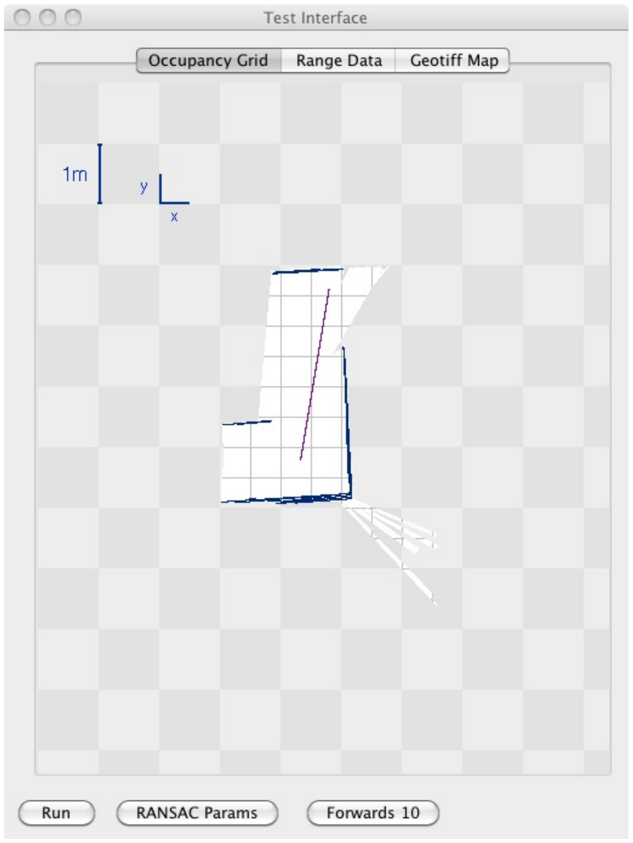


Figure 15.1: Planning a path based upon the current map.

This testing revealed a major issue with the `OccupancyGrid` class, which meant that all the maps were being reflected in the x-axis, and as a result the paths, although sensible, did not take the robot in the correct directions. This was caused by the origin for the Occupancy Grid being in the top left of the map, rather than the bottom left as was expected. It was solved by inverting the y-coordinates in a small set of functions, and ensuring that all map read and write operations used these methods. This meant that the maps became much more accurate, and that the robot began navigating sensibly.

15.6 Victim Identification

15.6.1 Hole Detection

Positive images: Testing of the Hole detection algorithm was done using positive images containing the area of interest. In total, 5 images of boxes with round hole openings from

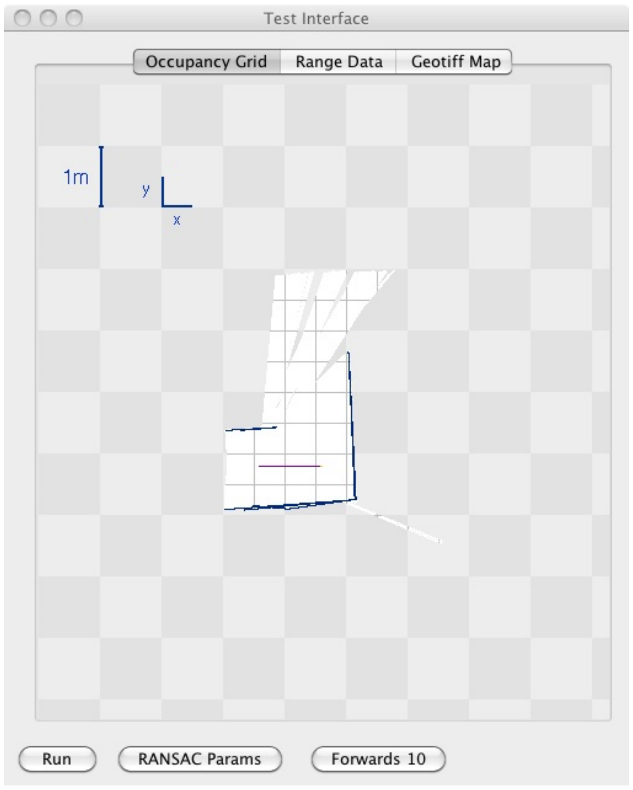


Figure 15.2: Planning a path based upon the current map.

RoboCup Rescue competitions were found in past materials. Also, 2 images of a wooden box with a round hole opening were found on the web. Testing proceeded by displaying a positive image on a computer screen and by pointing a web camera on the screen. The Hole detection algorithm was then run on the live camera input and detection results were displayed to the user.

Parameters for the Hole detection algorithm used for testing were:

- Value of threshold to create a binary image: 70
- Min. ratio of the sizes of the ellipse: 0.4
- Min. percentage of black colour within the ellipse: 0.9
- Max. percentage of black colour around the ellipse area: 0.04

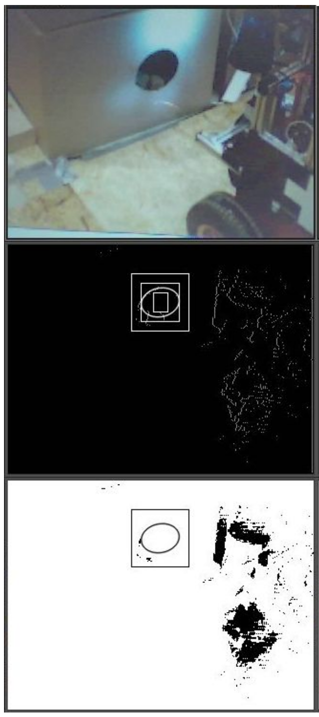
Success criteria was a positive detection of a hole in the box in the image. Example results of tests with positive images are shown in Table 15.1. Image names starting with a "c" were taken from past competitions, "w" are images from the web.

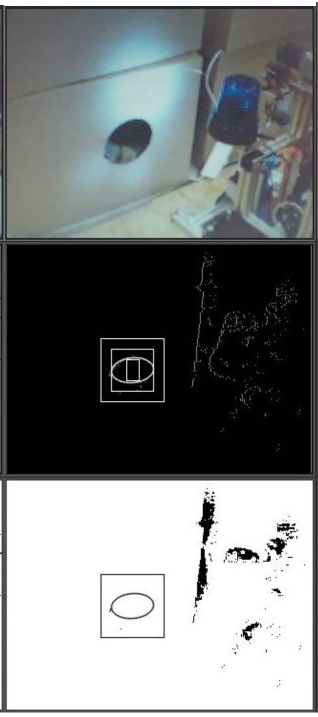
Image No.	*Test result*	*Comment*
c1	Correct	-
c2	Correct	-
w1	Correct	-
w2	Correct	Two holes detected, one of which not of a round shape.

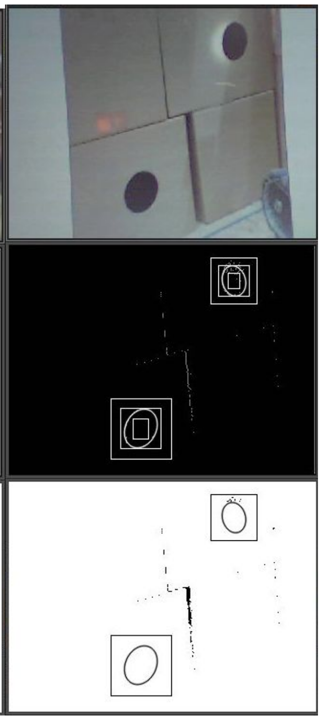
Table 15.1: Hole detection test results

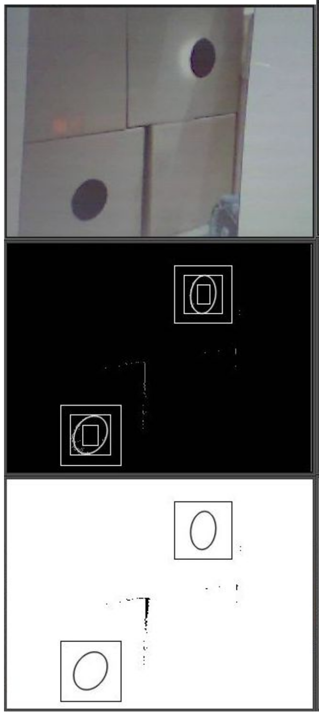
Examples of test results are shown in Figure 15.3.

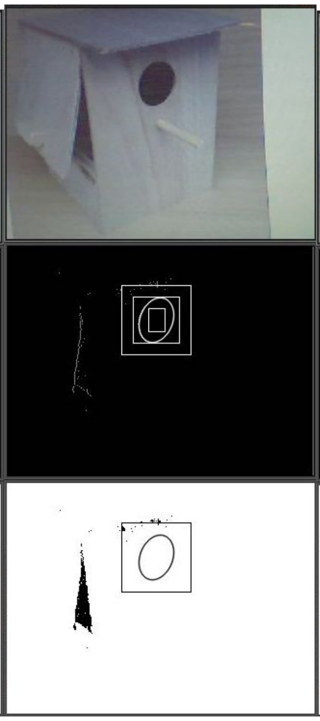
Negative images: Tests with negative images were also conducted. For this task, negative images collected as negative samples for the Face detection training were used. In total, 100 tests were conducted and no false detections were observed. Due to the nature of the Hole detection algorithm, however, it was found that a different kind of images should be used as negative samples, namely images which contain patterns similar to a hole in a box. This essentially involved images containing a dark shape surrounded with a light margin, such that the Hole detection algorithm would consider them a positive instance, based on the checks in the detection process. The web camera was used for this task and special objects


(a) Image ‘c1’


(b) Image ‘c1’


(c) Image ‘c2’


(d) Image ‘c2’


(e) Image ‘w1’

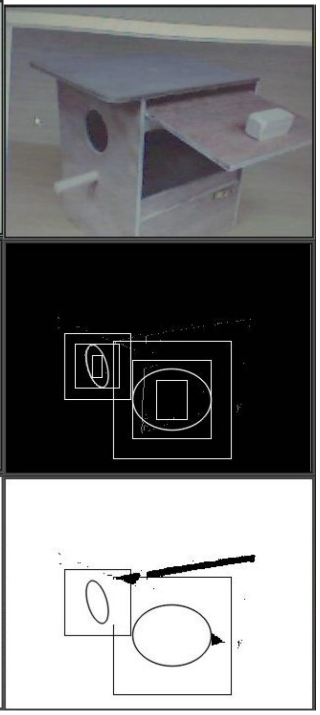
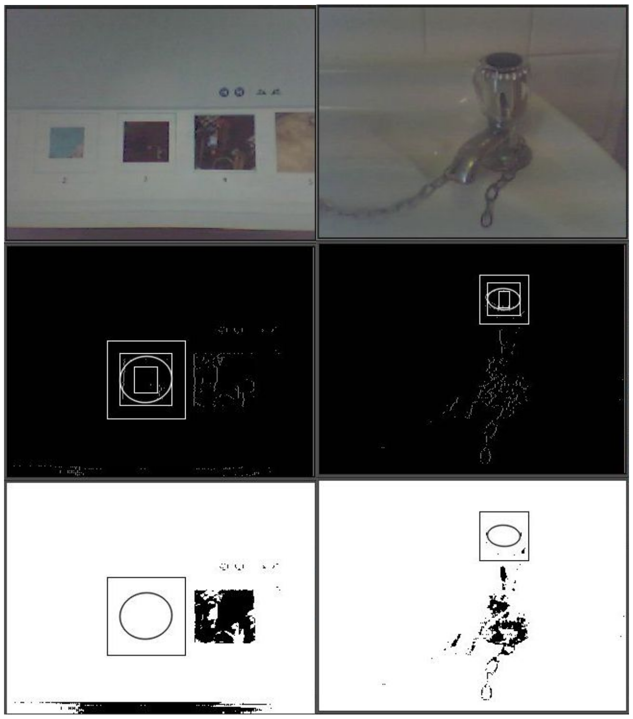

(f) Image ‘w2’

Figure 15.3: Hole detection: correct classifications



(a) A false detection made in the case of dark icons on the computer screen.

(b) A false detection made on water tap having a dark round button at the top.

Figure 15.4: Hole detection: incorrect classifications

having the said properties were found. Figure 15.4 shows examples of false detections in these tests.

Discussion:

The Hole detection method showed successful results in detecting the areas of interest in positive images. As for negative images, false detections were made in instances where a pattern appeared with similar properties to the hole detection problem. In cases of false detections, an object was detected, although it might not have a round shape. This is mainly because of the use of ellipse fitting, which may fit ellipses to non-round contours. This behaviour may be useful in cases when a part of the hole in the box is illuminated and the hole contour does not have a perfectly round shape. To eliminate false detections, however, additional checks may be employed to ensure that the detected object is indeed a hole in the box. The use of colour detection may come in useful when confirming the result, for example by detecting a specific colour of the box. Eventually, the amount of false detections in the arena will depend on the presence of objects similar to holes in boxes in the competition arena. It is important to note, though, that false detections are not of a very high concern, since they only provide an incentive for the robot to move towards the detected object and look for other evidence of a victim. On it's own, Hole detection does not constitute victim identification.

On the whole, the Hole detection algorithm can provide useful information based on the tests, since it has a high success rate in identifying holes in boxes, which serves as a trigger for the robot to move towards that direction and investigate the site for the presence of a victim.

15.6.2 Eye-chart Detection

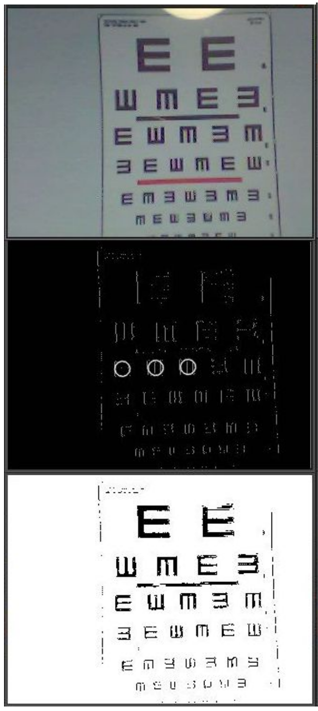
Positive images: Testing of the Eye-chart detection algorithm was conducted used a set of 5 images with eye-charts. 3 of the images come from past RoboCup Rescue competitions, 2 additional images were obtained from the web. A similar testing method with a web camera was used as for Hole detection. Additionally, for Eye-chart detection testing, tests were conducted with the web camera rotated, to test if rotated eye-charts would be detected.

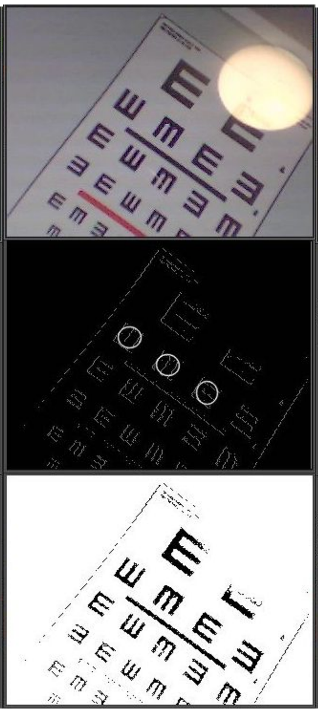
Parameters for the Eye-chart detection algorithm used for testing were:

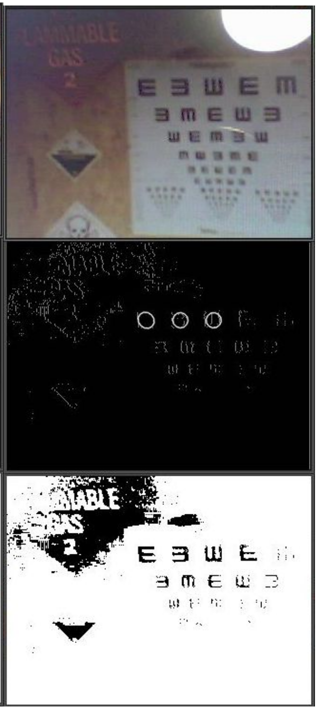
- Value of threshold to create a binary image: 70
- Min. ratio of the sizes of the ellipse: 0.4
- ϵ_d: 2.0
- ϵ_d: 0.03

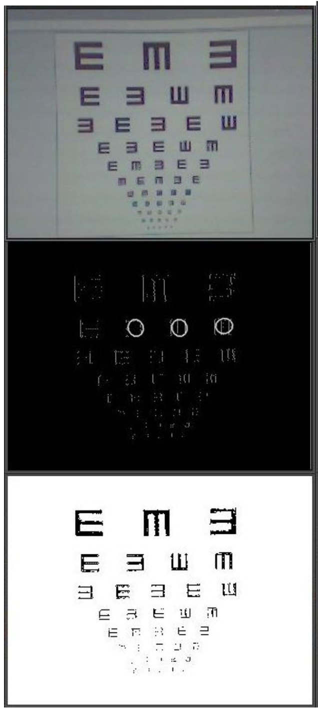
Test results: Tests were successful for all the positive images. Tests with the webcam rotated were also successful in all instances. In some cases during testing, the eye-chart was not detected due to the threshold setting being too low for the E symbols to appear fully in the binary image. However, further movement with the webcam around the image resulted in a successful detection. Examples of the results are shown in Figure 15.5.

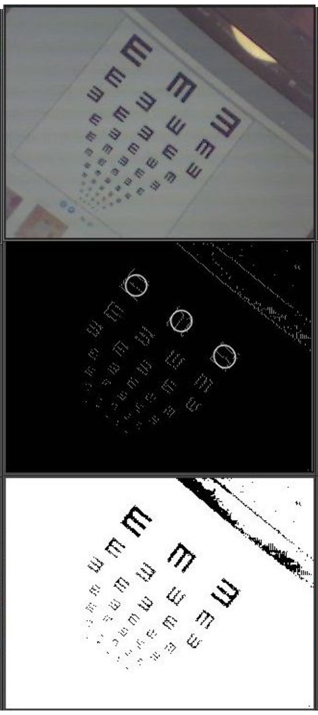
Negative images: Both the Eye-chart detection and Hole detection algorithms run in parallel. This means that Eye-chart detection was tested using negative images during the whole process of Hole detection testing, using the Hole detection testing images, as well as


(a) Test 1


(b) Test 2


(c) Test 3


(d) Test 4


(e) Test 5

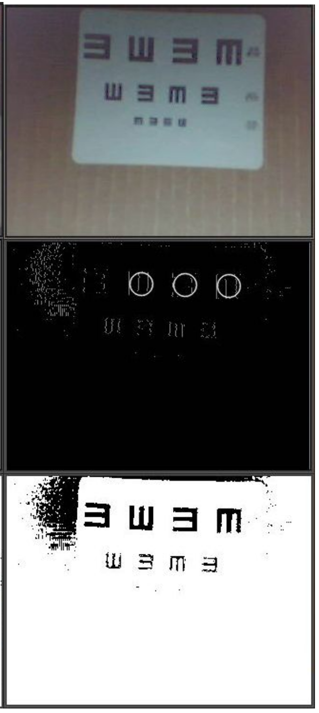

(f) Test 6

Figure 15.5: Eye-chart detection testing

images from the room the testing was conducted in. During this process, there was no false Eye-chart detection.

Discussion:

Based on testing results, it can be concluded that the Eye-chart detection algorithm provides a robust detection method and achieves good results. As it can be seen from the tests, even rotated eye-charts can be detected with the same success as eye-charts in a normal upright position. The method was also successful in achieving practically no false detections.

The only potential weakness may be in the fact that sometimes the threshold value for the binary image resulted in the 'E' symbols not appearing very clearly, and the Eye-chart detection not being able to fit ellipses around them. However, this is mainly an issue of adjusting the threshold value based on the current lighting conditions in the environment.

On the whole, a positive detection using the Eye-chart method can provide strong evidence in the victim identification process.

15.6.3 Face Detection

Testing of the performance of the Face detection classifier was conducted in two ways:

- Testing with new images downloaded from the web
- Testing with a doll and a web camera

Data was newly collected or obtained from a web camera for testing purposes, in order to observe the number of correct and incorrect face detections in new images. The results of tests with the two methods are presented.

Testing with Still Images

Positive images: New positive images were downloaded from online resources for this part of testing. The criteria for selecting the images to use for testing were similar to the collection of training images, which mainly meant images of dolls where the doll face is not rotated in a significant way, since the classifier had not been trained for rotated faces. The same applies to the fact that the face needs to be visible from the front and not from a side. Apart from that, images of a real doll were taken using a digital camera and used in the tests.

In total, 17 tests with positive images were conducted, from which 12 were from the web and 5 were taken using a real doll. Table 15.2 provides an example test results. Image names starting with an “w” are images from the web, whereas images starting with a “d” are taken using the real doll. A classification is considered correct if a face has been detected in the image that contains one.

Image No.	*Test result*	*Comment*
w1	Correct	-
w2	Correct	-
w3	Correct	-
w4	Correct	-
w5	Correct	-
w6	Correct	-
d3	Correct	Face detected, however in a wrong place in the image

Table 15.2: Face detection tests: positive still images

Negative images: Images not containing a doll’s face have were also downloaded from the web to test any false detections. In total 10 negative images were supplied to the classifier. Table 15.3 shows the results of the negative image tests for which there has been a false positive classification. A classification is considered correct in these tests if a face has not been detected since the image does not contain one.

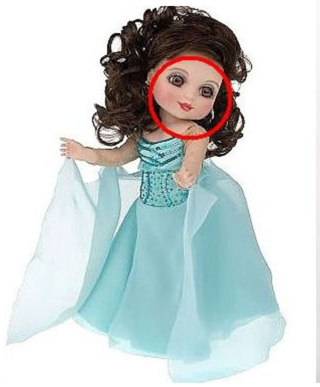
Image No.	*Test result*	*Comment*
n9	False	Several false positive face detections
n10	False	False positive detection

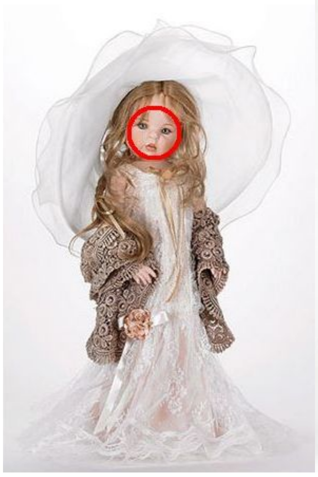
Table 15.3: Face detection tests: negative still images

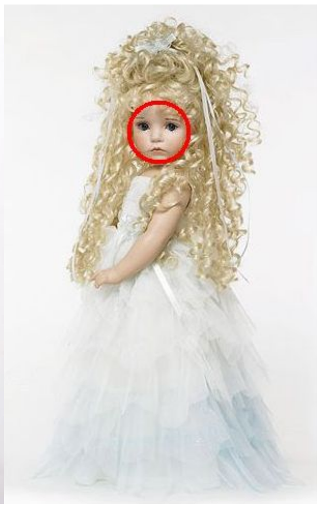
Figure 15.6 gives examples of test results using still images, both positive and negative.

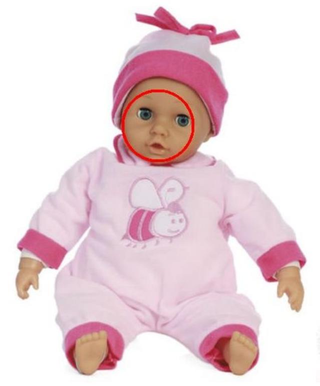
Testing with a Web Camera

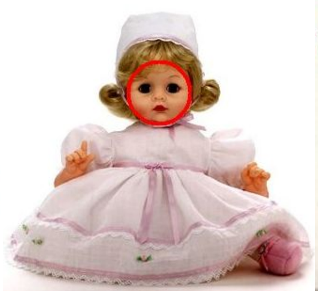
In this part of testing, a real doll was placed in front of a web camera and face detection results have been observed. Different positions of the doll relative to the camera and different distances from the camera were tried. Also, different lighting conditions were set up to

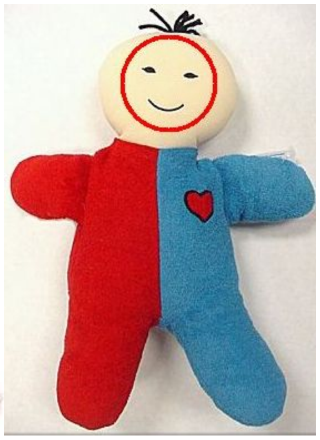

(a) Image ‘w1’

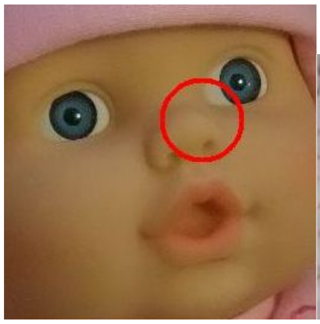

(b) Image ‘w2’

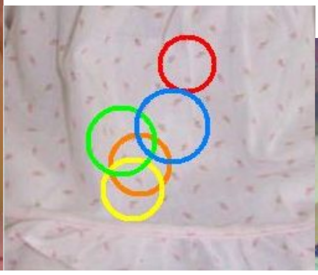

(c) Image ‘w3’

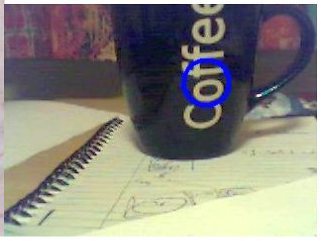

(d) Image ‘w4’


(e) Image ‘w5’


(f) Image ‘w6’


(g) Image ‘d3’


(h) Image ‘n9’



(i) Image ‘n10’

Figure 15.6: Face detection: tests with still images

investigate the effect of lighting in the environment on the face detection results. As this testing method works with real-time images from a camera, it is difficult to provide detailed test results. In general, the tests can be divided in 4 groups, based on the lighting conditions:

1. Darker foreground, lighter background
2. Equally lighted foreground and background
3. Lighter foreground, darker background
4. Dimmed foreground and background

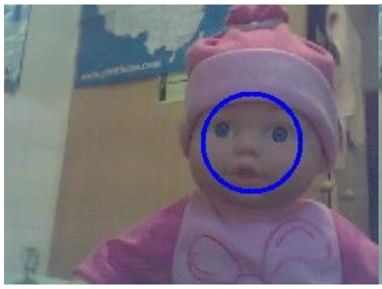
Examples of successful face detections in each of the lighting conditions are shown in Figure 15.7.

In general, it was observed that the best detection results were in the case of a lighted foreground and darker background. In contrast, the worst results were in the case of darker foreground and more illuminated background. The better illuminated face of the doll, the more often a face was detected. With regards to false face detections, hardly any were observed during the testing in the testing room, even when capturing scenes without any doll present. A few false detections, however, occurred when a doll was present in the camera's view and the detected region was in an area other than the doll's face.

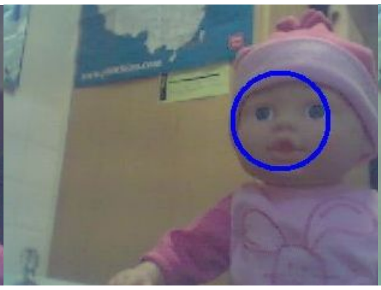
Discussion:

During testing, it was observed that false detections sometimes occurred in the presence of a strong pattern or texture in the image, which mislead the classifier. There have been false detections on some occasions when using the web camera, for example in the presence of a doll, when the classifier wrongly detected a face where there was none. On the whole, though, the number of correct detections strongly outnumbered false detections.

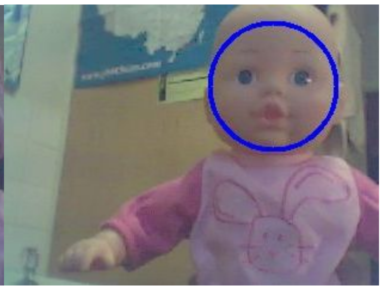
It is worth noting that detections made by the classifier when using the web camera seemed rather 'careful', meaning that the classifier seemed 'strict' in not making too many detections. This fact is probably due to the relatively small number of images used in the training process, since commercial face detection classifiers often use thousands of positive images during training and a similar number of negative images. An enhanced database of images would therefore be useful for creating a refined and more powerful doll face detection classifier in the future.



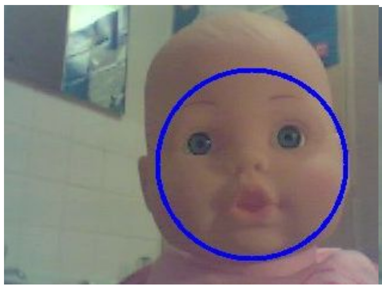
(a) Darker foreground, lighter background #1



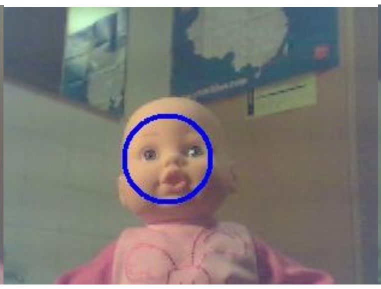
(b) Darker foreground, lighter background #2



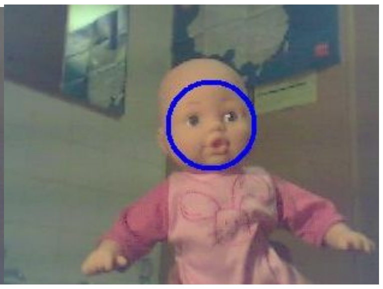
(c) Equally lighted foreground and background #1



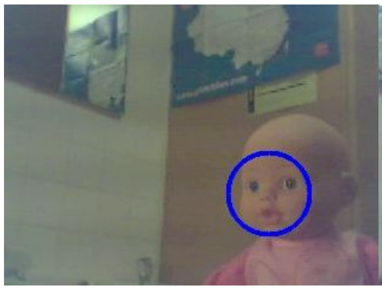
(d) Equally lighted foreground and background #2



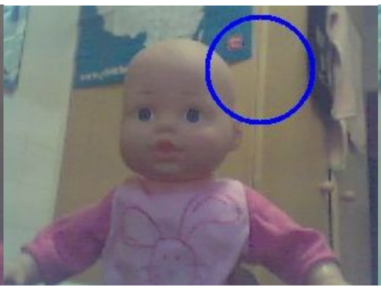
(e) Lighter foreground, darker background #1



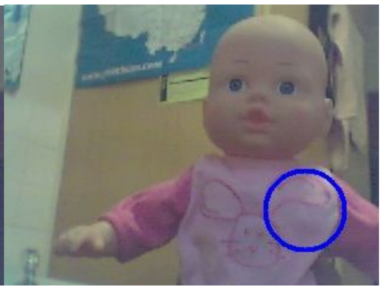
(f) Lighter foreground, darker background #2



(g) Dimmed foreground and background



(h) False classification #1



(i) False classification #2

Figure 15.7: Face detection: tests with a web camera

15.6.4 Infrared Blob Detection

To make sure Blob Detection gave as accurate results as possible, a sequence of manual and iterative tests were performed. The infrared binary image was tested in real time to make sure the threshold value was set such that background heat and noise was kept at a minimum. This was done using multiple humans to make sure only body heat or greater was detected and shown in the binary image. The threshold was changed manually to the highest value that it could be, within a margin, such just that body heat was detected.

To make sure that the Blob Detection algorithm returned suitable values, a simple test was used whereby a fixed number of heat objects were put in view of a the camera with a cold background. Objects were then added and removed from this view and the output of `DetectBlobs()` was checked correspondingly. The output contained the number of individual blobs detected and the horizontal angle to each blob. Another test undertaken was to slowly move one heat emitting object across the camera's field of view in a horizontal direction. The output of `DetectBlobs()` was then checked to make sure the angle field was changing accordingly.

The results of these tests were extremely positive and proved the accuracy of the Blob Detection algorithms.

15.6.5 Square Detection

Positive images: Testing of the square detection algorithm was done using positive images containing squares of interest. Pictures from the previous year's RoboCup competitions were used as input to the Square Detection function and the results were displayed to the user. As well as images from the competitions, live webcam feed was tested with the basic square detection algorithm without colour histogram checking to test that squares in various situations and backgrounds were detected.

Parameters for the square detection algorithm used for testing were:

- Value of high threshold for Canny: 200
- Min. size of square: 1000 pixels
- Max. ratio of size of square to image : 0.8

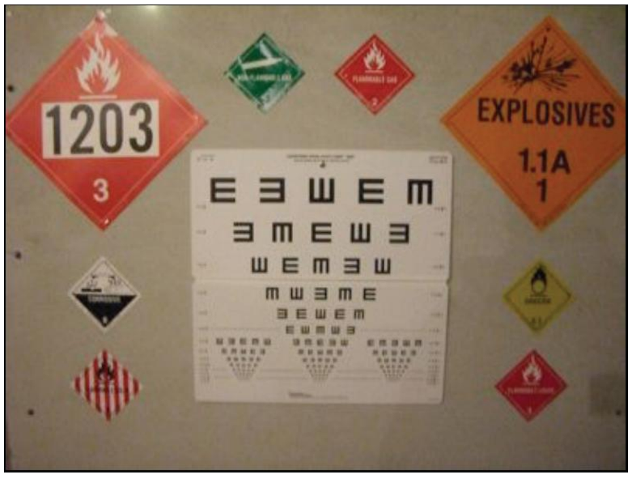


Figure 15.8: Competition Sign Example

Success criteria was a positive detection of a square in the image. Example results of tests with positive images are shown in Table 15.4. An example image from the competition is shown in Figure 15.8.

Image Type.	*Test result*	*Comment*
Competition	Correct	All signs detected, and only signs were detected
Webcam	Correct	Various squares were detected, some false positives

Table 15.4: Square detection test results

Negative images: Tests with negative images were also conducted using the webcam stream as input to the square detection algorithm. A relatively low number of false positives were detected. Only 2 false positive squares were output by the algorithm in the whole stream. These false positives came about when non-squares were partially out of the field of view and so formed a square with the edge of the frame. This was deemed acceptable for the competition, although improvement of the square acceptance methods would be an excellent candidate for future work.

Overall it was shown that the Square Detection algorithm gives a confident test of whether or not a square of interest is within an image.

15.7 Simulator

The simulator was tested with the SLAM functions that were implemented at the time. Testing the simulator simply took the form of running it and looking at the output images that were created to check if the results were those that were expected. The simulator tool provided to the team proved to contain a greater number of bugs than expected, and positive results when testing with SLAM were sporadic. This meant the development of the simulator took more time than was planned for and it was felt the importance of Victim Identification greatly superseded that of the simulator tool. Therefore development of the simulator was put aside before it could be fully completed.

All this meant that testing of SLAM within the simulator was not fully carried out and so all testing needed to be carried out on the robot.

Chapter 16

Conclusion

16.1 Competition

The team were able to secure funding for travel to the competition, although due to financial and timing constraints only one team member was able to attend. This allowed James Griffin to be on hand to fix problems with the robot, since the Engineering team were not trained in the use and the maintenance of the autonomous software. This also meant that the autonomous robot could be adapted to any new rules or terrain changes which the team were previously unaware of. This would prove to be very important. The rules of the competition were announced on the first day; the team were only able to work to the previous year's specification up until this point. These rules, along with insufficient testing of many sections of the software, would ultimately lead to a number of failed runs of the robot and a disappointing final result for the autonomous section.

16.1.1 Competition Environment and Rules

The terrain for the autonomous section differed from the team's expectations. Instead of slopes whose peaks are perpendicular to the direction of travel (i.e. forwards along a corridor, with peaks across the path rather than along it), the peaks were along the centre of the paths, meaning the robot would travel either on the peak of the slopes or along a wall at an angle. Figures 16.1 and 16.2 show images from the competition illustrating this issue. Due to the tracked nature of the robot, with low ground clearance, the robot would often find itself grounded with its base on a peak and no tracks touching the ground, or with one

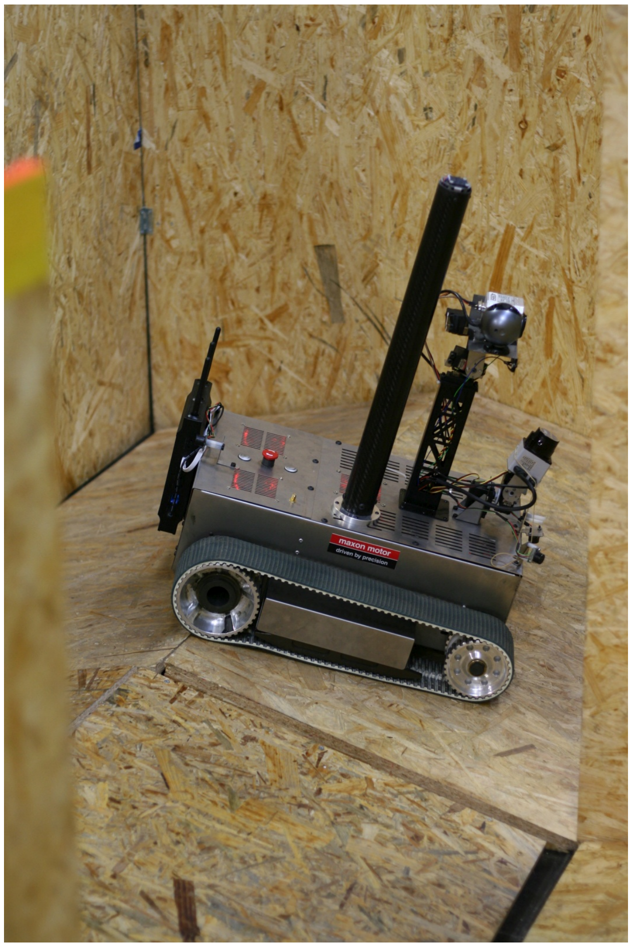


Figure 16.1: The robot navigating the competition arena.

track caught in the join between ramps at the peak. As well as this, the robot would often cross peaks during turns, meaning its position was even more difficult to estimate. Without a working EKF for localisation, the robot was incapable of producing a map that was of sufficient quality to submit for marking. As well as this, the precise colour format of the maps was not made available to the team until the competition itself, so changes to the map rendering software were required last minute at the competition.

16.1.2 Testing

One of the major problems was a lack of testing on the robot itself. This is a problem that cannot be attributed to any one group or individual, since it was due to a combination of late hardware availability and lack of suitable testing environments, which are due to issues

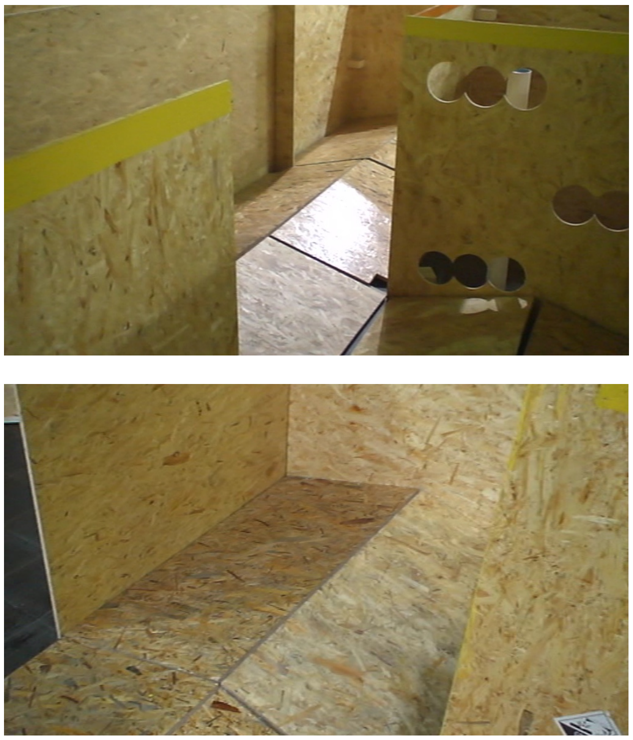


Figure 16.2: The sloped floors in the competition arena.

on both Computer Science and Engineering team halves.

16.2 Project Outcomes

The project was unique in that the objectives were derived from the competition's rules, which were not available in full until the competition itself. The slow release of information lead to the addition of victim identification as an objective at a relatively late stage. The majority of the project's initial objectives were met, as well as the development of victim identification techniques, with the exception of research into experimental SLAM techniques, which unfortunately could not be completed in the time available. The success of these individual sections shall be discussed here.

SLAM The algorithms for SLAM have been carefully chosen and implemented to the extent that the robot is capable of mapping environments of a similar complexity to those present in the competition. All individual SLAM sections have been built and tested thoroughly with the exception of State Estimation which, although the EKF seemed to operate correctly, the hypothesis wrapper layer has yet to be fully integrated into the solution. The code produced stands as a solid framework for further development, as will be discussed in the *Future Work* section.

GeoTIFF The current software can produce maps in almost complete accordance with the competition specifications, with full support for the GeoTIFF format.

Simulator Further development of the simulator was undertaken, however further modifications are required before it is suitable for use as a test environment for the SLAM algorithms, as shall be discussed in the *Future Work* section. Unfortunately due to unforeseen problems with the original simulator's state and due to timing issues, some of our objectives for the simulator were not met and therefore the simulator as a whole is incomplete.

Victim Identification Individual victim identification functions have been developed and thoroughly tested, however the integration of the entire solution could not be completed adequately in time for the competition. Further work in the area of victim identification shall be discussed in the *Future Work* section.

16.3 Further Work

Following discussions with other competitors at the RoboCup Rescue Competition, a number of areas of future work were identified. These are outlined below.

16.3.1 Scan Matching

Odometry estimation is an important part of the SLAM process. In this project, we relied upon the estimates provided by devices such as the motor control board and the compass, both of which provide information which is only accurate in certain perfect situations, and both of which require calibration in order to perform in different environments. This situation is not ideal, since the error ranges provided by these methods can be too large for SLAM to cope with. To this end, teams in the competition implemented advanced Scan Matching techniques, which can take two successive LIDAR scans along with an odometry estimate and produce an accurate estimate of the movement the robot has executed in terms of coordinate change and rotation. Algorithms such as Metric-based Iterative Closest Point

Scan Matching can provide this information, and investigation into the use of these could lead to the production of much more accurate mapping and localisation through SLAM.

16.3.2 Simulation

One issue the team faced throughout this project was that of robot availability. Since the robot was being constructed from scratch this year by the Engineering team, the robot was unavailable for testing for much of the year whilst it was being built and mechanically tested. When the robot was available for testing, limited battery life became an issue, since the batteries would run for around an hour and then require charging; a process which could take upwards of two and a half hours. The simulation environment provided by the Engineering team and extended by us was unfortunately not capable of accurately mimicking real-world situations with scanning errors, slippage, slopes and so on. As a result, it was difficult for the team to test the SLAM code thoroughly. Development of a full simulation environment which could take account of these real world factors and could simulate the robot with sufficient accuracy that little porting was required when the algorithms are to be run on the robot would allow teams to test the code not only more often, but also on a wider range of environments. This would not only produce more stable and general code, but would also mean the robot is less susceptible to surprise competition arena changes, such as the slopes this year.

16.3.3 User Interface

The current user interface provides minimal interaction with the robot beyond starting and stopping the SLAM process. Other teams demonstrated that it is incredibly useful to have an interface which provides real-time feedback from the robot, displaying information such as the current LIDAR scan super-imposed over the map, a 3-dimensional representation of the map in the robot's orientation allowing the operator to observe the robot's location easily, and streams from the victim identification sensors so that the operator may identify when the robot is missing victims and why. Along that line, some interfaces allowed the operator to run the robot semi-autonomously, such that the operator could specify a location on the map and ask the robot to navigate to that point. This can be useful in situations when the robot has not identified a victim or has missed a region of the environment from the map. Such a user interface would not only allow the operator to be more informed about the robot's current state, but would also aid testing and debugging on the robot.

16.3.4 Run Logging

Teams at the competition were able to log all of the robot's activity from runs, which allowed them to use this data for future testing and playback to identify issues and pitfalls with the robot in certain arenas or configurations. Such a feature would allow testing and debugging of the robot to be carried out using data directly from the robot, but without using the robot itself, thus saving time and effort.

16.3.5 2.5-Dimensional Scanning

Although the map produced by teams was always 2-Dimensional, it can be very useful to scan the environment in 2.5D, i.e. obtaining height information up to a certain height, but not fully 3-Dimensional. These 2.5D scans allow the robot to detect areas it should not explore, such as areas with drops and step fields. As well as this, the production of 2.5D maps would be a great advantage in the competition.

16.3.6 Map Format

The competition requires that maps display areas of the environment that have been "cleared" by the robot. This should be represented through the use of a gradient of colours based upon how confident the robot is that an area is unoccupied, usually determined through maintaining a record of how many times an area has been scanned. Other minor adjustments, such as labels for each victim and the starting position of the robot, should be added. The competition organisers, Johannes Pellenz (pellenz@uni-koblenz.de) or Adam Jacoff (adam.jacoff@nist.gov) should be contacted for the exact specifications.

16.3.7 Robot Platform

Few modifications should be made to the robot platform next year, so that any prospective team working on Autonomous software has a robot for testing purposes throughout the development cycle. One modification that should be considered essential is to improve the ground clearance of the robot, so that it can fare better on polished, steep slopes such as at the competition. This could be achieved through tracks similar to the ones on the tele-operated robot, which incorporate rubber pads that raise the robot up and improve traction.

16.3.8 Code Framework

Improvements in code interfaces between individual software components would be desirable for more structured inter-component communication. This will allow for more extensibility and a clearer component organisation.

16.3.9 Victim Identification

Motion Detection Motion detection is a widely used technique in image processing and may be a useful addition to the capability of Victim Identification. If victims in the simulated environment have moving parts, motion detection might provide strong evidence for their discovery.

Viola-Jones / Haar classifiers In this project, a Haar classifier has been implemented to detect faces of dolls in the environment. There is certainly scope for further work in this area and for refinement of the accuracy of the detection. For example, classifiers could be trained, so that they can deal with rotated dolls (lying sideways, rotated face, etc.). However, the Viola-Jones method is not only applicable to face detection. Classifiers could be trained to detect other objects of interest, such as eyes, hands, Hazmat labels, tags, etc.

Colour-based detection Colour-based detection was used in the square detection portion of this project, however a variety of applications of colour-based detection methods could be employed. These may include detection of hands based on colour, which could be further analysed using the edges of the hand. Colour detection might also be used to detect face colour, or colour of any objects placed around the victim. The colour signature of the area in which the victim is located with all its surrounding objects, could be analysed and detection methods based on the signature might be developed.

Infrared Blob Detection In this project, a infrared camera was successfully utilised to detect and analyse blobs of heat. Further work in this field would be to rank blobs based on the heat intensity. This would mean that the closest heat source could be identified and given priority. It also means victim situations could be described in more detail for the competition.

Intelligent evaluation of detection results The task of collating and evaluating results from various detection methods and producing a 'classification' on the presence of a victim provides scope for future work. Methods could be further developed to produce and adjust confidence in the discovery of a new victim. The confidence that a victim has been found could be evolving with the robot moving closer to the victim. After initial evidence is observed, the robot could move towards the victim's potential location and any additional evidence could increase the confidence value. This task would involve deep integration with SLAM and the route planning components.

Also, methods for more intelligent evaluation of the detection results could be developed, potentially using artificial intelligence (AI). These would enable the robot to 'learn' which combination of results typically leads to the presence of a victim. For example, some detection methods may produce inaccurate results on their own, but when combined with results from other methods, they could lead to more accurate classifications. When working in a lab environment, these interactions could be studied and eventually incorporated in the classification process. An interesting problem might be to train such a meta-classifier based on real scenarios with victims in a simulated environment. An even more interesting problem might be to enable the robot to train its own meta-classifier in a completely new environment in 'real time'. In other words, 'learning on the fly'. During the training, a user might manually confirm victim classifications made by the robot, which would gradually 'learn' about the best combination of results. This scenario would of course require a certain number of different detection methods available and time for sufficient training of the robot in a new environment. An advantage of this method would be a much increased adaptability of the robot to a new environment.

16.4 Conclusions

This project has demonstrated the requirement for access to the robot hardware throughout the project lifecycle, as well as the need for regular integration of components and testing of the full solution well in advance of the competition. Although the software developed this year was unsuccessful at the competition, it is hoped that with further development Warwick

Mobile Robotics will be a competitive entry against the other RoboCup Rescue autonomous teams.

Bibliography

[1] Geotiff (official website). http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf.

[2] Phidgets (official website). http://www.phidgets.com/.

[3] Geotiff - a standard image file format for gis applications. http://partners.adobe.com/public/developer/en/tiff/TIFF6.pdf, 1992.

[4] T. Bailey and H. Durrant-Whyte. Simultaneous localization and mapping (slam): part ii. *Robotics Automation Magazine, IEEE*, 13(3):108 –117, sept. 2006.

[5] H. Choset and K. Nagatani. Topological simultaneous localization and mapping (slam): toward exact localization without explicit localization. *Robotics and Automation, IEEE Transactions on*, 17(2):125–137, 2001.

[6] Frank Dellaert College and Frank Dellaert. The expectation maximization algorithm. Technical report, 2002.

[7] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, 1972.

[8] H. Durrant-Whyte. Localization, mapping, and the simultaneous localization and mapping problem, 2002.

[9] Walter Gander, Walter G, Gene H. Golub, and Rolf Strebel. Fitting of circles and ellipses - least squares solution, 1994.

[10] Shuzhi S. Ge and Frank L. Lewis. *Autonomous Mobile Robots: Sensing, Control, Decision-Making, and Application.* CRC Press, 2006.

[11] B. Hiebert-Treuer. An introduction to robot slam (simultaneous localization and mapping). "http://dspace.nitle.org/handle/10090/782", 5 2007.

[12] S. Julier and J. Uhlmann. A new extension of the kalman filter to nonlinear systems. In *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls, Orlando, FL*, 1997.

[13] Kurt Konolige. Improved occupancy grids for map building. *Auton. Robots*, 4(4):351–367, 1997.

[14] Andol Li. Hand detection using opencv. http://www.andol.info/hci/830.htm, 2009.

[15] Rainer Lienhart, Er Kuranov, and Vadim Pisarevsky. Empirical analysis of detection cascades of boosted classifiers for rapid object detection. In *In DAGM 25th Pattern Recognition Symposium*, pages 297–304, 2003.

[16] Stefan Winkvist Michael Tandy and Ken Young. Competing in the robocup rescue robot league. http://wrap.warwick.ac.uk/2718/, 2010.

[17] Jose Neira and Juan Tards. Data association in stochastic mapping using the joint compatibility test, 2001.

[18] S. Martinelli A. Tomatis N. Nguyen, N. Gchter and R. Siegwart. A comparison of line extraction algorithms using 2d range data for indoor mobile robotics. *Autonomous Robots*, 23(2):97–111, 8 2007.

[19] NIST. Performance metrics and test arenas for autonomous mobile robots. http://www.isd.mel.nist.gov/projects/USAR/.

[20] Roger Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2001.

[21] Sren Riisgaard and Morten Rufus Blas. Slam for dummies. http://ocw.mit.edu/NR/rdonlyres/Aeronautics-and-Astronautics/16-412JSpring-2005/9D8DB59F-24EC-4B75-BA7A-F0916BAB2440/0/1aslam_blas_repo.pdf.

[22] Diego Rodriguez-losada and Javier Minguez. Improved data association for icp-based scan matching in noisy and dynamic environments, 2007.

[23] Paul L. Rosin. Further five-point fit ellipse fitting. *Graph. Models Image Process.*, 61(5):245–259, 1999.

[24] Mike Ruth. Geotiff faq version 2.3. http://www.remotesensing.org/geotiff/faq.html, 2005.

[25] Naotoshi Seo. Opencv haartraining (rapid object detection with a cascade of boosted classifiers based on haar-like features). http://note.sonots.com/SciSoftware/haartraining.html, 2008.

[26] R.C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *The International Journal of Robotics Research*, 5(4):56–68, 1986.

[27] R.C. Smith and P. Cheeseman. A stochastic map for uncertain spatial relationships. In *The International Symposium on Robotics Research*, number 4, 1988.

[28] G. Welch and G. Bishop. An introduction to the kalman filter. Technical report, Chapel Hill, NC, USA, 1995.

[29] G. Zunino. Simultaneous localization and mapping for navigation in realistic environments. In *In Proc. IEEE Conference*, pages 67–72, 2002.