Title

An Empirical Evaluation Of Map Building Methodologies in Mobile Robotics Using The Feature Prediction Sonar Noise Filter And Metric Grid Map Benchmarking Suite

Author

Shane O'Sullivan

Master of Science

University of Limerick

Supervisor

J.J. Collins

Submitted to the University of Limerick November 2003

Abstract

Many successful autonomous robot control architectures demonstrate competency at tasks such as obstacle avoidance and wall following. These architectures have been largely based on elements of Brooks' subsumption paradigm which partly advocates minimal use of internal state to ensure tight integration of the sensory-motor control loop. However many of these are merely reactive systems. In order to carry out more complex tasks, some forward planning is required. This means that, unlike purely reactive systems, some inner state is needed to record facts about the environment and to determine what steps need to be taken in order to achieve the goal.

In order to develop a strategy for completion of a goal, for example path planning, some internal representation of the surrounding environment, a map of some sort, is required. The objective of this research is to compare and contrast various methods of map-building using sonars, including the traditional grid based probabilistic methods and gaussian models, as well as the more recent methods using refined specularity models, information-redundancy prevention and Bayesian update theory. Platform independent modules have been developed to test and compare the approaches put forward by Elfes, Matthies, Moravec and Konolige. Theoretical evaluations of the map-building approaches advocated by Thrun and Crowley are also undertaken within.

One of the greatest difficulties of map building is in dealing with the noisiness of sonars, which often leads to inaccurate maps. A method called Feature Prediction is presented that compensates for this noise using algorithms to predict features in the environment before the robot senses them.

There is little agreement between researchers as to what exactly constitutes a good map, and there are therefore many benchmarking methodologies in the literature. A number of these are discussed within, as well as two new tests that calculate the usefulness of the map when used by a robot to navigate.

Acknowledgements

First and foremost, thanks to my parents who, besides feeding me for so many years, bought me my first ZX81, and taught me to program BASIC on a Spectrum, planting the seed of frustration (that damn computer WILL do what I tell it!!) that has driven me to being a programmer.

Secondly, I must thank JJ Collins my supervisor, who thoughtfully waited until a year into the research before telling me that he originally considered this whole thing to be impossible, as well as providing me with numerous ideas and some much needed direction.

Thirdly, thanks to the other members of the robotics group in the University of Limerick, Mark Mansfield, Gerard Conway and Dave Haskett, for all the insightful conversations, and with whom I developed the distributed framework on which to run and test my software.

Finally, thanks to Ciara for everything else.

Table Of Contents

Table of Figures

Chapter 1: Introduction

1.1 Research Objective

The purpose of this thesis is to implement, compare, and contrast the various paradigms currently in use for mapping an environment with a mobile robot. The methods tested include:

- Probabilistic Occupancy Grid theory put forward by Moravec and Elfes [46].
- Bayesian based Occupancy Grid methods by Matthies and Elfes [42].
- Konolige's MURIEL [33] method for eliminating incorrect sonar readings.

Theoretical evaluations are also carried out on work by other researchers. These include:

- Crowley's hybrid method [15] of extracting features from a local grid map.
- Elfes' Inference Grids [20].
- Thrun's automatic learning of sensor models [57, 58] and confidence values using neural nets.

The thesis also presents an algorithm called Feature Prediction for filtering noisy sonar readings in order to build more accurate maps.

The methods being evaluated enable a robot to model its environment using a number of paradigms, for example a 2D grid [15,33,42,47], or a topological graph [57,58]. This thesis examines the strengths and weaknesses of the available methods, as well as suggesting improvements. It determines which method, or what combination of multiple methods, best enables a robot to accurately model its surrounding environment through the construction of a map.

Many different types of sensors are available to researchers in mobile robotics, and these are discussed later. This thesis is concerned with map building using sonar ultrasonic sensors for many reasons, as discussed in chapter 2. However a very extensible architecture is presented that facilitates the incorporation of additional sensory inputs.

While mobile robots are used here as a test bed for the various map-building strategies, they are just one of a number of possible platforms. This thesis is therefore not a review of possible robot architectures and their pros and cons, rather one robot architecture is designed and discussed, and one simulator is used, the Pioneer 1 robot and Saphira simulator by ActivMedia Robotics. All algorithms performing map building are evaluated using this architecture and simulator.

The process of evaluation involved building a number of modules for the purpose of map building. These modules are architecture independent and work with any robot configuration. To ensure real-time operation with limited resources, as well as to promote extensibility, these modules were deployed in a distributed framework using CORBA (see Appendix A) although they are also capable of being deployed on a single machine. This framework has, and will continue to be, added to by other researchers in the robotics group to include path-planning capabilities, neural net based mapping techniques, as well as pursuit and evasion algorithms to create a highly functional mobile robot.

The contribution of this thesis is as follows.

1. It provides an empirical evaluation of a number of map building methods.
2. The Feature Prediction algorithm has been developed to enhance map building algorithms, specifically to identify noisy specular readings.
3. Development of a suite of benchmarking techniques for map building algorithms.
4. Design and implementation of platform-independent robot control architecture, with support for threading, multiple clients and callbacks, and can be run on a single machine or distributed over a network.

1.2 The Importance of Map Building and Localisation

Map building is important for a number of reasons. Primarily, an accurate model of a robot's surrounding environment enables it to complete many complex tasks more quickly and reliably than without such a model. While a variety of robotic activities are possible to perform with minimal inner state [5], for example wall following or collecting soda cans [7], many other activities require considerable forward planning for them to be completed within and acceptable timeframe. Examples of such activities include path planning, searching for objects [59] or places, or vacuum cleaning a floor. Without an internal map of the environment, a robot cannot plan a path to a place not currently sensed by its sensors. Neither could it effectively search for an object or place since it may search the same place repeatedly and never know that it was retracing its steps.

Another advantage of map building is that it potentially provides a good framework for integrating information from many different sensors, possibly of heterogeneous types, from many different positions and directions, into a single knowledge source, from which intricate plans can be created [42, 59]. This is beneficial as it compensates for many weaknesses inherent in the various sensors used in map building. For example, if both vision and sonars were used in conjunction, the vision system may not detect a white wall, whereas a sonar is more likely to detect it, or the sonars might miss a hole in the ground, but the vision system will possibly detect it. If the information from both sets of sensors is integrated into the same map, then they can be used to either confirm or dispute each other, resulting in a more robust, less error-prone model of the environment. This in turn leads to more successful plans being created that require fewer changes to be made to them during their execution.

One of the more prominent difficulties is mapping an environment is in knowing the exact position and orientation the robot was in when it received the sensor readings. If the wrong position is used, then the incorrect part of the map will be updated, leading to large errors. Information regarding the distance the robot has travelled, and in what direction, is usually calculated from measuring the number of times each wheel has turned, also called odometry. Unfortunately, occurrences such as wheel slippage and angular drift (when the robot’s odometry claims it is going in a certain

direction, but it is slightly wrong) can lead to the estimation of the robot's position being extremely erroneous. It is therefore necessary to use the robot's other sensors to correct this error in *pose* (position and orientation) estimation. This is called localisation.

Localisation is used in an attempt to solve two related problems:

- *Global localisation*, the ability of the robot to decide where it is in relation to the rest of the environment. Global localisation is necessary when the robot has a previously generated map of its environment. When it is first turned on it must be able to know where it is within that map in order to incorporate its new sensor readings into the correct area of that map.
- *Position tracking*, when the robot decides where it is in relation to where it was. Position tracking is used to compensate for the problems mentioned earlier of wheel slippage and angular drift caused by things such as differences in wheel pressure and slippery surfaces.

When performing map building with mobile robots, a method called Simultaneous Localisation and Map-Building (SLAM) is usually performed. Using this method, a map is build as the robot traverses an environment, while a localisation routine runs in parallel, continually updating and correcting the robot's estimated pose.

The issue of localisation, despite its necessity, is beyond the scope of this research. A flexible framework, detailed in Chapter 4, has been developed which allows the easy addition of services to the overall robotic architecture, and it is intended that other researchers in the UL robotics group will add localisation modules to complement the mapping modules developed for this thesis, just as they have added pursuit and evasion and path planning modules.

1.3 Problems Inherent in Map Building

There are two primary difficulties when it comes to building maps of an environment.

1. How to translate sensor readings into knowledge about the environment. This includes modelling the data returned by a sonar as an object in the world, as well as compensating for the considerable amount of noise in sensor readings. A large portion of this thesis is dedicated to these topics, with Chapter 2 discussing various approaches taken by researchers in the past [15, 20, 33, 42, 47, 57, 58, 59], and Chapter 3 presenting a new method called Feature Prediction for estimating the reliability of sonar readings.
2. Dimensionality of an environment. The more information that is stored about an environment the more use can be made of it, but also the more memory and computation time is required. This thesis focuses on two-dimensional maps, which is a tractable problem using today's desktop workstations. However there is work currently underway [47] into full three-dimensional maps that store huge amounts of data.
3. Estimation of the robot's position. As explained in the previous section, errors in measurements taken of the number of times wheels have turned lead to error in the robot's estimation of its pose accumulating over time to eventually rendering a map useless. Localisation routines can be used to correct these errors. However, these are not examined in this thesis.
4. Dynamic environments. Environments can change over time, doors open, chairs move, people move. The map that a robot built initially may, after a period of time, no longer be valid. This is an open problem in the domain of robotic map building, and one that has yet to be solved.

This thesis examines various solutions to the first map building issue listed above, that of how to interpret the sonar readings received by the robot. The issue of localisation is beyond the bounds of this work. However, it must be stressed that localisation is essential to the area of map building, which is illustrated in Chapter 6, Experimentation, where the quality of the map produced in simulation (without odometry error) being far superior to the map produced when a real robot is used to map an area without localisation routines.

This thesis therefore deals with three main issues:

1. Compare and contrast different map building strategies, some theoretically [15, 59], and others both theoretically and experimentally [33,42,46].
2. Developing a method for recognising the type of reflection being exhibited by the sonar beam and assigning confidence measures to each sonar readings using an algorithm called Feature Prediction.
3. Designing and implementing a comprehensive suite of benchmarking methods for evaluating the fitness of a map generated by a given map building method.

1.4 Interesting Properties of Intelligent Robots

Early attempts at imbuing a machine with intelligence met with what was then considered to be some amount of success – computers could play draughts [26, 55], play chess [48], and manoeuvre in very simple static domains such as the blocks world, for example the robot Shakey developed at SRI [49]. At the time, the possibilities seemed endless. As Herb Simon, a noted AI researcher said in 1957:

> *"… There are now in the world machines that think, that learn and that create. Moreover their ability to do these things is going to increase rapidly until – in the visible future – the range of problems that they can handle will be coextensive with the range to which human mind has been applied."*

However the AI community soon hit a brick wall, and could go no further. The reason for this, argues Brooks [6], is that by simplifying the world in which the machine operated, researchers were ignoring the very problem that they should be attempting to solve. Both checkers and chess are finite problems within a clearly defined world and the blocks world presented simplistic shapes that a robot could recognise, such as cubes and rectangles. Brooks argued that to infer true intelligence upon a machine, we have to imitate the process of evolution of humans. By this he meant that instead of taking away all the difficult properties of the world in order to allow a robot to navigate, as was done in the blocks world approach, we should first tackle problems such as obstacle avoidance, wall following and other so-called simplistic actions.

Brooks argues that it is better to try to achieve insect-level intelligence, a realistic goal, than to attempt to imitate humans and failing. This means that before the robot is able to model the world, it should first be able to exist in it safely, and to carry out simple tasks. The reason the early robots failed was that they tried to reason everything about their surroundings, and while they were busy doing that, they crashed into something. This mode of operation is called the Sense-Plan-Act cycle (SPA), and is expanded upon in section 1.5. Brooks advocated less of a dependence on inner state, with almost no time spent between sensing and action meaning that robots can now avoid obstacles, follow walls, and even collect soda cans [7] in a dynamic environment. He coined the now famous phrase "*the world is its own*

model" to explain that rather than try to understand the world, we should be using it, and that the only way to create true intelligence in a machine, is to create a machine that can operate in a real world environment. He also argues that the field of robotics is an ideal test bed for such development since all other applications of artificial intelligence, whether game playing, medical diagnosis or whatever, have had the complexity of the world removed a priori by a human.

The goal of early AI researchers of making a machine that 'thinks' and 'understands' like a human has all but been abandoned due to its apparent impossibility. Instead, largely as a result of the work carried out by Brooks and the MIT Artificial Intelligence lab, the focus has shifted to creating machines that can carry out tasks that humans may find mundane, such as vacuuming a floor, or tasks that are too dangerous for humans, such as deep sea exploration or scouting other planets. These are being done through the field of robotics, and with more success than ever before.

1.4.1 What is Artificial Intelligence?

Possibly the simplest, and least controversial, definition of Artificial Intelligence (AI) is the study of how to build and/or program computers in order that they can do the kind of things that minds can do. One problem with this definition is that it assumes that computers can do what minds do, i.e. diagnose, advise, and understand. This problem can be avoided by saying that AI is the development of computers whose observable performance has features, which, in humans, we would attribute to mental processes [3].

Many systems have been developed that are in line with the above definition, such as systems for medical diagnosis [29], navigation and image recognition. However the holy grail of AI research is not merely to create systems that can carry out complex functions, but to create systems that comprehend what it is that they do [3]. As the method of teaching often used with children states: learn first, understand later. This means that in order to understand our environment we must first know it, and it is this learning step that this thesis is concerned with. Building a map of the environment a robot must operate in is a method of organising and validating the information the robot can extrapolate using both its sensors and past information. The problem of

comprehending the stored information is beyond the scope of this work, and is, for the foreseeable future beyond the abilities of the scientific community as a whole.

1.5 Mobile Robotics

As mentioned earlier, this thesis is not concerned with the particulars of the various robot architectures available. The mapping techniques discussed herein can be applied to a number of different platforms, for example hand held devices used in underground cave mapping or indoor building modelling. However, it is useful to have an understanding of the platform used in experimentation in order to comprehend the experiments performed and the results obtained. As such, here follows a brief introduction to the field of robotics.

In mobile robot navigation, four different approaches have been proposed.

(i) Traditional approach – Sense Plan Act (SPA) cycle
(ii) Subsumption / Reactive architecture
(iii) Hybrid architecture
(iv) Deliberative approach

In the traditional approach, perception, planning and execution follow each other in that exact order (see Fig 1.1). This theoretically enables robots to deal with complex problems, but it is inflexible and its top down behaviour makes it almost impossible to implement in real life.

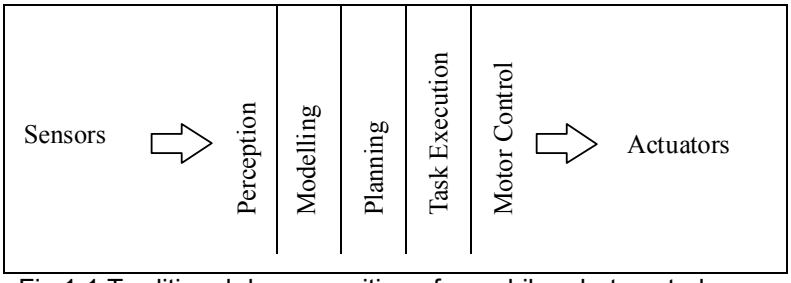


Fig 1.1 Traditional decomposition of a mobile robot control system into functional modules

Brooks [10] criticised the traditional approach, saying that there were too many steps, and hence delays, between sensing and action. Brooks also felt that the symbol system used in the traditional approach to represent the world was a bottleneck. Since all

information must flow through the symbol-based system, processes couldn't run in parallel, which leads to slow deliberative behaviour, which is not well suited to dynamic environments. The SPA approach also suffers from the fact that trying to extract artefacts from sensory channels is a complex problem. When just incomplete information is available, as often happens, the choice has to be made whether or not to make assumptions as to the properties of an artefact, which can lead to errors in the planning stage.

Traditional symbol systems are also under constrained [28], which means that they can, and do, represent anything that is logically possible, including a huge number of highly unlikely concepts. Thus, time is wasted reasoning about events that can never occur, a situation referred to as the frame problem. As Fodor put it [22], The "frame problem", he writes, is: "Hamlet's problem: when to stop thinking"

To solve these problems, Brooks created his *Subsumption Architecture* (see Fig 1.2), which was a behaviour-based, layered architecture with a bottom up approach. Each layer in this architecture represented a behaviour, with all layers running in parallel, with very little interaction between them. This worked well with low-level behaviours, such as obstacle avoidance and wall following, but did not allow for higher-level functions such as learning or planning.

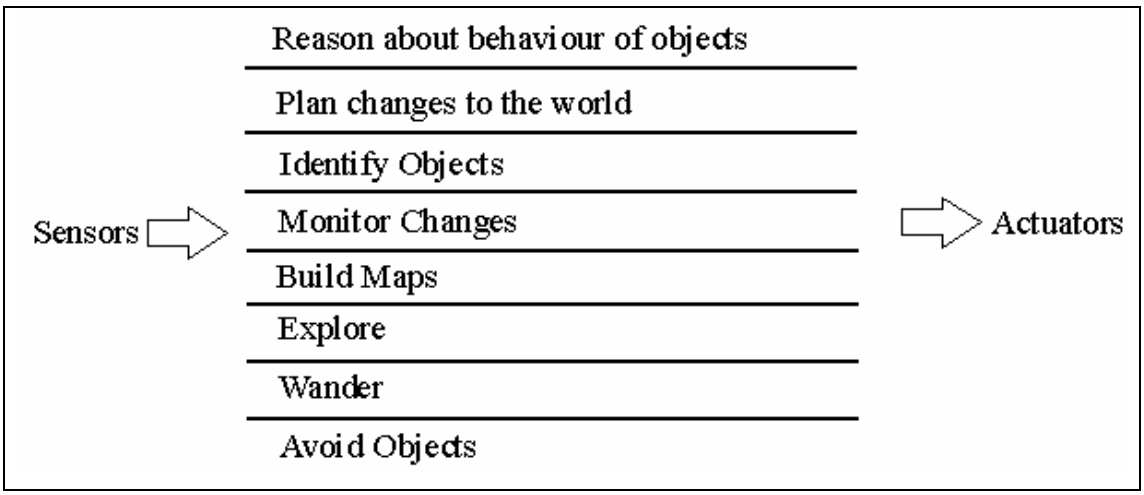


Fig 1.2 Decomposing the robots activities into asynchronous behaviours with the subsumption architecture

Because of this lack of higher functions, a third approach was proposed, a hybrid architecture. This involves using a behaviour based, reactive system for low-level control, and a central planning device like the traditional approach for higher-level behaviours like planning and mapping. An interesting development of this method is the use of a deliberative layer to perform higher-level functions like mapping and navigation. This layer can be implemented through neural networks and/or genetic algorithms, among others.

The mobile robot RHINO [59] designed by Thrun and his team of researchers used an architecture similar to the 3T architecture. Their distributed modular architecture ran simultaneously on multiple computers, both on-board and off-board. Each computer ran one or more higher level modules, with low level behaviours residing on the robot itself. The higher level modules, whether their function be map building, localisation, path planning or something else, executed as an 'any-time' algorithm. This means that the module was not constrained by real time considerations, and that whenever it completed processing the information given it, it would report its results to the appropriate controller module. This design strongly influenced the robot control framework designed for this thesis as laid out in Chapter 4.

The distributed robot control framework developed for this thesis and extended by the robotics group in UL is designed is such a way as to evoke emergent behaviour from a robot. Emergent behaviour is when a system made up many individual parts, each of which is self-contained and capable of performing its function regardless of the presence of any other modules, displays what seems to be intelligent behaviour because of the interaction between the modules. It uses the three-layer architecture or 3T [24], which is quickly becoming the de facto standard for mobile robotics. The 3T architecture is a further development of the subsumption architecture and consists of a *skill* layer, a *sequencing* layer and a *planning* layer, which are the bottom, middle and top layers respectively. The skill layer tightly couples the sensors to the actuators, with the bare minimum of internal state. It implements primitive behaviours, such as wall following and obstacle avoidance and is identical to the *reactive* type of system Brooks advocated [5]. The sequencing layer decides which behaviour should be active at any given time, arranging a number of different behaviours in order to carry out

useful tasks. The sequencing layer contains some internal state, but it must operate in real time. As Gat [24] states:

> "*The sequence should not perform computations that take a long time relative to the rate of environmental change at the level of abstraction presented buy the controller.*"

What the rate of environmental change is exactly is open to discussion, but is generally accepted that the process should complete within one clock cycle of the robot. The planning layer is where most of the time consuming operations take place, and it usually doesn't work in real time. This layer is where all the path planning, localisation and map building take place.

The Saphira architecture designed by Kurt Konolige [32] was chosen for experimentation, and was used for obstacle avoidance and basic movement, although the robotic mapping services are designed to work hand in hand with any other architecture. Saphira acts as both the bottom and middle layers of the 3T architecture. Saphira uses a version of the Procedural Reasoning System (PRS) behaviour sequencing language, Colbert, to string together basic actions, for example wandering or following an object. The map building, path planning and pursuit-and-evasion services sit on the top of the basic motion layer as the planning, or deliberative, layers. All of these services act asynchronously to the others to achieve their own goal, and it is possible to remove any of the upper layers without affecting the lower layers. They do not work in real-time, and may lag behind the operation of the robot, but this is to be expected with the deliberative layer.

1.6 Research Methodology

The methodologies used in the completion of this thesis included:

- Literature review – A complete review of the previous research in the area of map building and localisation in mobile robots. The result of this work can be seen in chapter two where the work done by previous researchers such as Moravec, Elfes, Konolige, Thrun and others is explained and analysed in depth.
- Design and implementation of robotic control architecture.

- Simulation – Both development and experimentation made much use of the Saphira simulation environment to speed up implementation of the test suites.
- Embodied Experimentation – Experimentation took place on a Pioneer 1 robot in a variety of environments including hallways and a small enclosed environment.
- Development of comprehensive suite of benchmarking techniques for determining the quality of a map.

1.7 Overview of Thesis

Chapter two contains an overview of map building, as well as a theoretical analysis of work by Alberto Elfes, Larry Matthies, Hans Moravec, Sebastian Thrun, Kurt Konolige and James Crowley.

Chapter three details the Feature Prediction algorithm designed as part of this body of research to filter out incorrect sonar readings.

Chapter four outlines the Saphira architecture the experimental modules operated on, as well as describing the various map building services used in experimentation. It also outlines the overall distributed framework these modules are part of.

Chapter five details the suite of benchmarking techniques applied to the maps in order to test their fitness, based on a number of different criteria.

Chapter six describes the experiments carried out and the results obtained.

Chapter seven presents the conclusions that can be drawn from the theoretical analysis and experimentation results, both simulated and real world.

1.8 Publications

Four papers based on research presented in this thesis have been accepted for publication at the Ninth International Symposium on Artificial Life and Robotics (AROB) 2004 in Oita, Japan. The titles are as follows.

1. *Linear Feature Prediction for Confidence Estimation of Sonar Readings in Map Building* – O'Sullivan, S, Collins, J. J., Mansfield, M, Eaton, M, Haskett, D. This paper is based upon research discussed in chapter three.
2. *Developing an extensible benchmarking framework for map building paradigms*.– Collins, J.J., O'Sullivan, S, Mansfield, M, Eaton, M, Haskett, D. This paper is based on research presented in chapter five.
3. *A Quantitive evaluation of sonar models and mathematical update methods for Map Building with mobile robots* – O'Sullivan, S, Collins, J.J., Mansfield, M, Eaton, M, Haskett, D. This paper is based upon the empirical experimentation results presented in chapter six.
4. *Developing a statistical baseline for robot pursuit and evasion using a real world control architecture*.– Mansfield, M, Collins, J.J, Eaton, M, O'Sullivan, S, Haskett, D. This paper's experimentation results were derived using software built upon the software architecture described in chapter four.

Chapter 2: Map Building

2.1 Map Building – An Overview

Many biological entities sub-consciously and deliberately create maps of their environment using highly developed senses like sight and touch. Depth perception gives us the ability to tell the relative distance of objects, we can recognise shapes and distinguish colours. We are able to recognise locations and plan paths between different places, as well as knowing the spatial relationships between these locations. Most importantly we can update this cognitive map almost instantly based on new information, for example a new door being installed, or a previous open door being locked. Unfortunately, as with other attempts to mimic humans and animals abilities with machines [4,49], getting robots to map their environments, recognise places and know the distance from one place to another is quite difficult and the subject of much study.

There are two main questions that need to be answered when it comes to map building:

1 What kind of map should be built – a metric grid-based map (Fig 2.1 (b)), a topological graph-based map (Fig 2.1 (c)), or a combination of both? Part of this also has to do with how new information should be integrated with old information to produce an up-to-date map.

2 How should the sensors be interpreted – i.e. what information can be extracted from a sonar or laser range reading, or a camera image, and how does one deal with inaccuracies or errors in the readings?

To briefly answer the first question, there are many factors that go in to deciding what type of map to use. These include ease of construction, equipment availability/cost, storage requirements, processing power requirements and suitability to the task at hand. Metric (grid based) maps, which store information regarding the obstacles in an environment and the spatial relationship between them, are relatively easy to construct but need a lot of storage and can be computationally hungry. Topological maps, which are graph-based structures that usually only record the existence of recognisable places and the paths between them with no distance informaiton, are

more difficult to build but are very quick to plan paths with. These are described in detail in section 2.3. There are a number of different map-update procedures advocated by different researchers, from the relatively simple methods used by Moravec and Elfes [46], to more mathematically robust Bayesian update procedures used by Matthies, Elfes and Konolige [20,33,42]. Other approaches do not use a mathematical formula at all; instead they train a neural network to calculate the probability that a certain space in the environment is occupied [58]. These too are discussed at length later in the chapter.

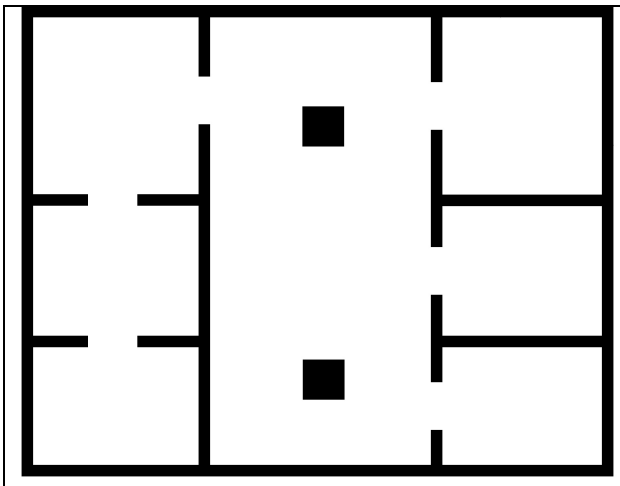


Fig 2.1 (a) A real-world environment with seven rooms. Areas shaded black represent obstacles in the environment, white areas represent freespace.

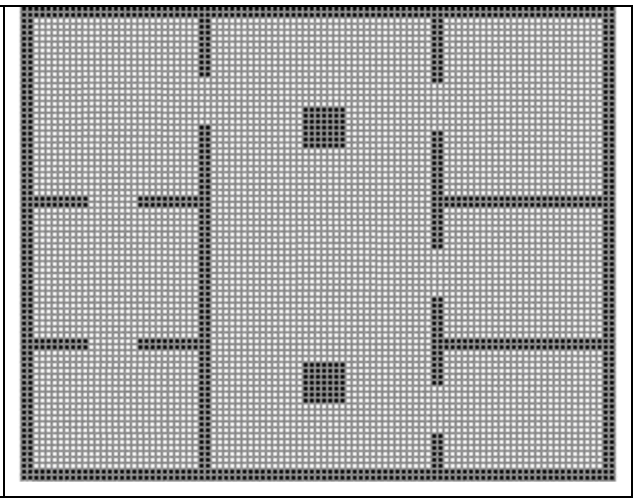


Fig 2.1(b) A metric map of Fig 2.1 (a) with seven rooms connected by doors. Space is divided into equally sized squares with obstacles/walls shaded black and empty spaces blank.

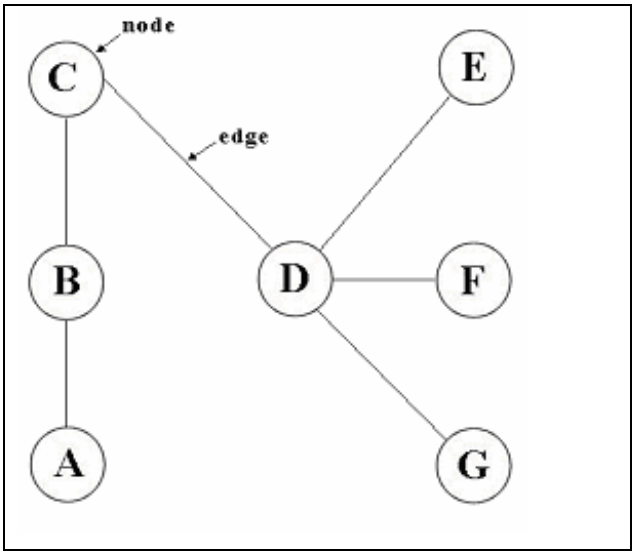


Fig 2.1 (c) A topological map of the same area as Fig 2.1(a), with nodes representing places such as rooms, and edges representing navigable paths between these places, i.e. doors.

As to the second question, how to interpret the sensors, there are a number of approaches for interpreting sonar readings, and the discussion and analysis of these is the main contribution of this masters. Some approaches use gaussian error models of the sonars [33, 42, 46] which assume that if a sonar beam bounces off an object then the object is most likely at the centre of the beam and at the distance reported by the reading. Another approach taken is to use neural nets to interpret the sonar readings and give a figure representing the probability of occupancy of a particular place in the environment [58]. These approaches, in their various forms, are discussed later in the chapter.

2.2 Maps and Robot Architectures

Each of the three most used robot architectures, SPA, 3T and subsumption, allow for the ability to create a map or a model of the robot's surrounding environment.

2.2.1 Maps and the SPA architecture

Early research into autonomous robotics tended to use a three-step approach to navigation: first the robot attempts to use its sensors to build a model of the world, and second the robot uses the model to plan its actions, and finally it would act on those plans. This was the Sense-Plan-Act cycle. One of the better known examples of this is the Shakey STRIPS method [49]. For each action the robot took, it created a table representing its knowledge about the initial and goal states, calculated the distance between these states and applied an operator to shorten that distance. Whenever an action worked and brought the robot closer to its goal the action was added to a list of viable actions to use, and whenever an action failed it was added to a list of unsuccessful actions. In this way it created a type of model of things it had *learned*, actions that it should take in certain states, and actions it should not. While there were numerous problems with these early attempts, such as inaccurate models that came from assuming that the sensors were perfect, their main downfall was their speed, or lack of it. The robots moved in very short steps, with long delays between steps. The creation of the model of the environment got in the way of the operation of the robot, which led to a change in thinking regarding if and when an environment model should be updated.

2.2.2 Maps and the Subsumption Architecture

During the 1980's some researchers became disillusioned with this state-based SPA approach, with perhaps the best-known being Rodney Brooks at MIT. They attempted to minimise the amount of processing between sensing the environment and acting upon it's findings, and through their research into behaviour based robotics [5], developed robots using a subsumption architecture. These robots could simulate insect-level intelligence, performing simple tasks such as wall-following and obstacle avoidance. The subsumption architecture divided its various functions into various layers, all of which could work concurrently to each other. In this way the (optional) map building modules did not prevent the lower level behaviours such as obstacle avoidance and wall following from performing as they should.

2.2.3 Maps and the 3T Architecture

The 3T architecture, as outlined in Chapter 1, is made up of three layers. The bottom two layers are used for real-time control, whereas the top layer contains all the time consuming operations that cannot guarantee completion of processing on one set of readings before the next set of readings are received. It is in this top layer that map building algorithms reside.

2.3 Map Types

Once it is decided to use a map, the next decision is what type of map to choose. This decision depends on the tasks the robot is required to perform. The different types of maps can be broadly separated into four groups. These are

- Full metric maps.
- Topological map.
- Metric Topological maps.
- Recognisable locations

Metric maps describe the environment by subdividing it in one of two ways.

(i) By feature – the map contains a list of primitive features, e.g. cylinders, cubes, and the properties of each [36]. For example, a wall can be represented as a line between point *A* and Point *B* (Fig 2.2), or a cabinet can be represented as a rectangle. The distance and orientation of the shapes in relation to each other

is known. This type of map is comes as standard with the Saphira robot simulator used in experimentation, and an application of it is discussed later in relation to the work of Crowley [17].

(ii) By area – the map is divided into regions, with the properties of each region (usually whether or not it is occupied) being recorded (Fig 2.1 (a)). These regions are generally represented as a grid with each square of the grid being the same size as all other squares.

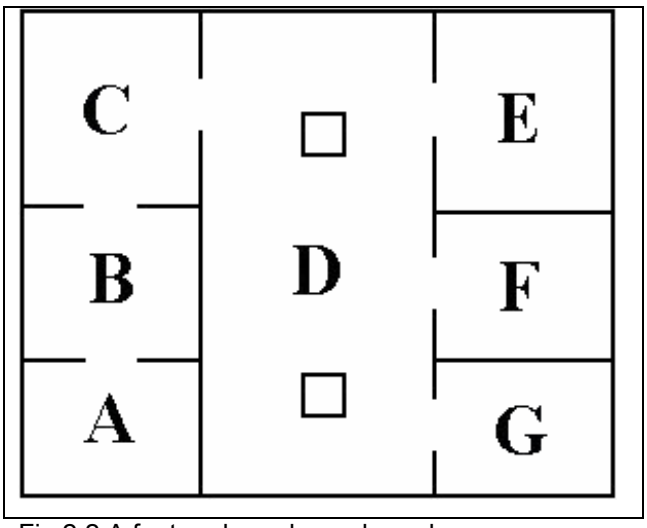


Fig 2.2 A feature based map based on same environment as Fig 2.1.

Metric maps are useful when there are few landmarks, e.g. open spaces, no walls. Area based maps are typically used when the application is focused on the use of free space. Feature based maps are used when the properties of the obstacles are of concern, and is often used for self-localisation because it is useful to know the orientation of objects when estimating the robots position.

Topological maps record links between recognisable locations, and are usually represented as an undirected graph (Fig 2.1b). A location is anything that the robot might be able to recognise, such as a room, or a distinctive object. If a link exists between two locations it means that it is possible to travel from one location to the other. A link can therefore be a door, or a corridor, and can only be created once the robot has actually navigated from one location to another.

Topological maps are very useful when it comes to planning a path from one location to another. An algorithm such as Dijkstra's can be used to quickly find the least number of nodes necessary to visit in order to complete the journey. This is considerably faster that path planning in a metric map, which usually consists of using an algorithm like A-Star to find the shortest path. Unfortunately the path chosen for the topological map may not be the shortest however, due to the fact that it is not

known how far one location is from another, just that it is possible to reach one from the other.

Topological maps are more difficult to construct than metric maps due to the difficulty in recognising locations. For example, in an office environment many offices may look the same, and a robot can easily mistake one for the other. One method used to solve this problem [58] is to combine the relative ease of building metric maps with the speed of planning afforded by topological maps. Building a grid-based map of the environment, then applying an algorithm to infer a topological map from it, does this. The method used for the conversion is discussed later.

Metric Topological maps are similar to topological maps, but provide additional distance information about the paths between locations so that instead of just knowing that it is possible to travel from location *A* to location *B*, we also know that they are 20 metres apart. This makes the path planning algorithms more optimal due to the fact that the algorithms don't just count the number of different locations they have to visit before reaching a goal but can now judge the distance between places.

For example, if we had three locations, *A*, *B* and *C* with links from each location to both others, and we wanted to travel from *A* to *C* in Fig 2.3. It is possible to go from *A* straight to *C*, or from *A* to *B* to *C*. A simple topological map would count the number of locations it had to visit and choose the path *AC*. However a metric topological map could tell us that the path *AC* is 100 metres, and the paths *AB* and *BC* are 20 and 10 metres respectively, resulting in a much shorter path.

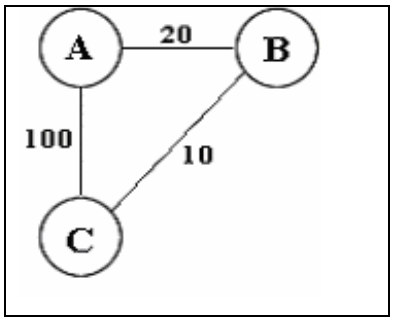


Fig 2.3 Metric topological map

Metric topological maps can also help with landmark disambiguation i.e. they help to tell similar landmarks apart. For example, if locations *B* and *C* look similar to the robot, but it knows that it was just in location *A* and had to travel 100 metres to get here, it is in location *C* since *C* is 100 metres from *A*, whereas *B* is 20 metres from *A*. A simple topological map would just know that it can get from *A* to both *B* and *C* and not be able to differentiate between the paths used to get where it currently is.

Recognisable locations are the least useful type of map. Using these, the robot knows where it *is*, but knows nothing about how to get anywhere else. Some methods for recognising a location include using image recognition to identify something in the environment, such as a distinctive object recorded earlier, matching sensor readings against stored readings, using a Self Organising Map (SOM) to monitor the position of the robot by recognising similar groupings of data received by the sensors. Using only recognisable locations enables the robot to know that it has been in this position before, which can be useful in some situations. For example, a robot could be trained to recognise all recharge positions in a building, and each time it sees one it checks its battery power and decides whether or not it needs to use the recharge point.

2.4 Previous Research in Map Building

Much research has taken place in the area of feature-based maps. The Stanford Cart, built by Hans Moravec during the 1970's [45], used stereo vision to determine the location of features in 3D space, but was prone to mistakes and far too slow. The cart could only move one metre every 15 minutes or so and this was done in fits and starts. It would move a metre, stop and take some pictures, and process them at length. This processing involved applying an algorithm called the *interest operator* to the pictures, which picked out distinctive regions or objects in the pictures. Next the *correlator* algorithm attempted to find these objects in previous pictures. Finally the *camera solver* estimated the position of the objects based on their movement from picture to picture. Obviously this is a huge amount of processing and the limited computers at the time could not handle the processing load.

Moravec has recently revived the project [47] with a new robot and far more processing power. His ongoing project using three cameras capable of colour stereoscopic vision in combination with distributed computing (part on-board, part off-board) is yielding impressive results, reproducing the real world in a reduced-dimensionality 3D map. It goes even further than the Stanford cart, taking the images from three cameras at the same time and using the colour of objects to correlate their position. While the processing load is prohibitive meaning that the robot currently moves in slow steps, the algorithms are being designed with future CPU speeds in

mind, and will soon be able to run in real time using an on-board computer and an off-board workstation connected via radio.

Crowley [15] simplified Moravec's three dimensional model by assuming that the world could be modelled in 2D space rather than 3D, and modelled the world as a collection of line segments using a rotating sonar for its sensor. His view was that for robot navigation it didn't matter if there was a book on a shelf, all that mattered was whether or not the floor space was occupied. Crowley's maps used both the area based metric approach and the feature based approach. The map is first built using a grid representation (area based) for ease of construction, then lines are extracted from it for compactness and processing speed. Crowley's mapping algorithm kept two maps, a global map and a local map which only contained what the robot could currently sense. The local map is constantly being matched against the global map to localise the robot and compensate for wheel slippage, and is also being integrated into the global map to provide new information. Crowley's work will be discussed at length later in the chapter.

In part due to the inability of the Stanford Cart to operate in real time, Hans Moravec and Alberto Elfes [46] proposed a two dimensional representation of the environment. This representation divided the area into a grid of squares with sonars being used to determine the probability of occupancy of each square. The more often a sonar reading reported that there was an object at a given square, the higher the probability that it was occupied, and the more often a square failed to make the sonar beam bounce off it, the higher the probability that it is not occupied. Elfes followed up on this work in later papers, developing better mathematical models for integration of new data [42] and attempted to infer a large number of properties about the environment from the grid map [20] with varying degrees of success. Elfes work, which has strongly influenced this thesis, is discussed later in the chapter.

Kurt Konolige took the work of Moravec and Elfes, as well as other researchers who had developed area-based maps, and identified one of the largest problems, that of redundant information. That is, using information that tells us nothing new about the environment, and allowing it to affect the world model. He developed the pose bucket method to discard useless information, as well as creating a new mathematical

model, the *multiple target* model, for integrating new information into the map. Another of Konolige's contributions to the field of map building is his attempt to deal with the problem of noisy sonars using past information to calculate what he called the *probability of specularity* of each sonar reading. All of these are elaborated upon further later in the chapter.

2.5 Metric Area Based Maps

This thesis is primarily concerned with the construction of area based metric map, for their usefulness, their popularity with other researchers and their relative ease of construction over topological approaches which often stray into the area of image recognition. Area based maps come under many different headings, but they all have one thing in common – they are all divided into distinct regions with each region recording a property about itself (usually occupancy). When designing an area-based map, three questions need to be answered:

1. What shape and how large should the regions be?
2. What numbers should be stored for each region?
3. How should the numbers be updated?

By far the most logical and efficient shape to use in an area based metric map is the square, and it is used by more or less everyone in the map building field. However the question remains as to how big the squares should be, and what's the minimum size achievable for real time operation and with the accuracy of the sensors?

As to how small the squares can be, this depends on both the speed of the computer used and the accuracy of the sensors. A laser sensor is much more accurate than a sonar emitter, and can therefore provide information about a smaller area. Performance can become a problem when the squares are too small, as there are many more values to update and use. For example, when a sonar reading gives information about a area of one square metre, there much more work must be done to update the cells representing that are when there are 10,000 values per square metre (each square 1cm to a side) than when there are 100 values (each square 10cm to a side).

1	1	1	1	0	1	1	1
1	1	1	1	0	0	1	1
1	1	1	1	0	0	1	1
1	1	1	1	0	0	1	1
0	0	1	1	0	0	1	1
0	0	1	1	1	1	1	1
0	0	1	1	0	0	1	1
0	0	1	1	0	0	1	1

Fig 2.4 (a) Area based representation of an area.

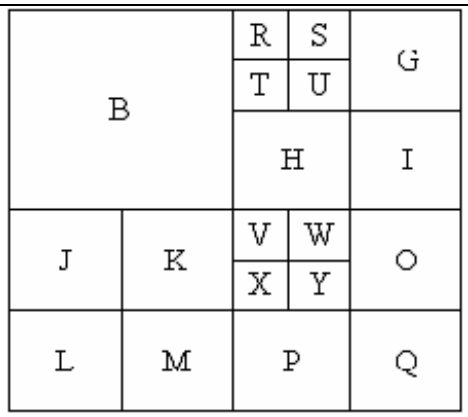


Fig 2.4 (b) Same area as 2.4(a), divided into areas of equal value.

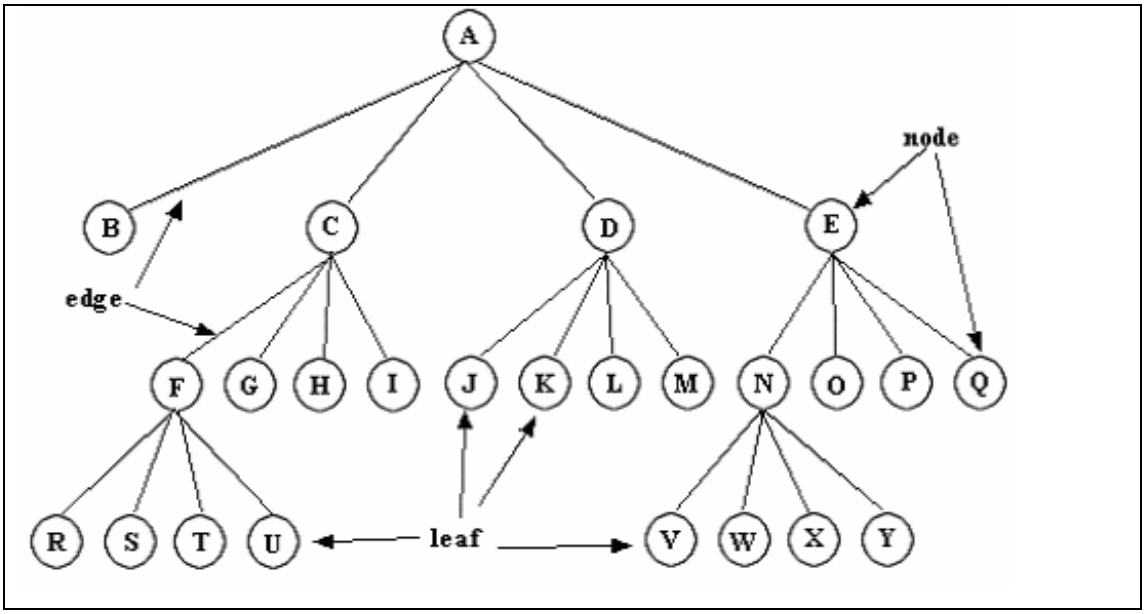


Fig 2.4 (c) Quad tree representation of (a) and (b).

Node A is the universal node, which represents the complete map. Its four child nodes are B, C, D and E, each of which gives more detail about a particular quadrant of its parent node's area. Each of these are broken down further into more children nodes if all the values contained in that quadrant are not uniform. Referring back to Fig 2.4 (a), because all cells in the top left quadrant have the same value, they can be represented by a single node, B, rather than sixteen different values, as in Fig 2.4 (a).

When deciding how large the squares should be, there are two possible approaches – grids with all squares the same size (Fig 2.4 (a)), and quad-trees, a more efficient method for sparse environments (Fig 2.4 (c)) with different sized regions.

A regular grid can be very inefficient when storing many similar numbers, for example with large open spaces. This is due to the fact that it stores a number for every possible region regardless of whether or not it is identical to all those around it.

However, they are very quick to update, since an update simply requires a Cartesian coordinate to identify the square to be changed, and a number being updated doesn't mean that the table's structure has to be altered.

Quad trees are very efficient when it comes to storing large amounts of identical numbers. This is because they only create the storage space necessary to store a number if it is different from the other numbers in its quadrant. For example, in Fig 2.4 (a), all of the numbers in the top left quadrant are the same, so they only have to be referenced by a single node, *B*, in Fig 2.4 (c). There is more variance in the numbers in the top right quadrant, so it consequently requires more nodes as can be seen with the numerous children of node *C*. Quad trees are slower to use than a static grid, as one cannot simply assign a value to a Cartesian position – the new value must first be compared to the current leaf's value, and if they are different four children are created and their values assigned to them. This all adds to the processing load. Which method to use depends on which is more at a premium, processing power or memory.

In the modules used in the experiments carried out in this research, even-sized grids were used to store the maps since the amount of memory required, less than 10 megabytes, falls well within the memory capacity of a standard desktop computer. Quad trees are used to store pose buckets (see section 2.11.7) due to the prohibitively large amount of information they are required to store.

The type of numbers held in each region depends on the application, but a common method is to record the occupancy status (unknown, empty, or occupied) and a probability value. Sonar sensor readings are taken from multiple positions using an array of emitters whose position in relation to each other is known. These are then interpreted to give distance information and projected onto a 2-D grid-based map. Due to the fact that sonars are inherently unreliable, what with all the noise from specular reflection and sound-absorbing substances, this grid is probabilistically plotted, as we can only specify our degree of certainty of occupancy of any particular cell. Moravec and Elfes [46] proposed using numbers in the range [-1, +1], with numbers less than zero indicating a level of certainty that a given cell was empty, greater than zero indicating occupation, and numbers equal to zero indicating the

absence of information on that cell. Each cell of this grid is therefore assigned a value, in the range [-1, +1], which is called its occupancy value.

Updating was originally done using a simple addition rule to merge new data with old. More recently, this has been superseded by a more mathematically rigorous Bayesian update rule. This has led to the majority of other researchers using numbers in the range [0, 1], with 0 representing an empty area, 1 an occupied cell, and 0.5 representing a lack of knowledge about the area. The advantage of this is that when the number 0.5 is put into Bayes equation, meaning that we know nothing new about the environment, no change is made to the cell value.

These metric maps have many advantages to map builders. They are relatively easy to construct, facilitate path planning, and are useful in map matching for self-localisation. Places and obstacles are also represented in relation to the robot's assumed position, which means that similar looking places, such as identical offices, can be recognised as being dissimilar, unlike topological approaches.

A key disadvantage of the metric approach is its computational cost. A grid of n*m evenly sized cells requires the same amount of space, regardless of the complexity of the environment. Quad trees solve this problem, but are slower to operate. Path planning is also inefficient when compared with topological approaches, with every possible path needing to be tested to see if it is the best. Metric methods can suffer greatly from errors in odometry such as drift and slippage, and these must be constantly compensated for with self-localisation techniques.

2.6 Topological Maps

Topological approaches differ from the metric approach insofar as they do not have a global coordinate system. The theory is that it is not strictly necessary to know every property of an environment. For navigation, it is only required that the places and objects necessary for the completion of the goal be known – each being represented as a node in a graph – as well as how to navigate from one node to another. To this end, the topological approach stores the map as a graph, with the nodes representing

recognisable locations or landmarks, with the edges representing clear paths from one node to another, usually doors or corridors.

An obvious advantage to this approach is the huge savings in both the space it takes to store, and the time it takes to plan a path. If it is sufficient to model only the properties of the environment of use to the robot, rather than recording every possible feature, then the size of the graph is directly linked to the complexity of the map, and is often many orders of magnitude smaller than a metric map. As for path planning, a simple algorithm such as Dijkstra's algorithm can be used to compute the shortest path between any two nodes.

Unfortunately, topological maps can be difficult to build. They have also been known to exhibit the perceptual aliasing problem, i.e. to confuse similar looking places, as they do not store the geometric relation between these places and other nearby landmarks, just the existence of paths between them. For this reason, they can be unreliable when it comes to self-localisation.

2.7 Sensors Used in Robotics

In robotics, a sensor is any piece of equipment which retrieves data on either properties of the environment or the robot's relationship with the environment. There are a wide variety of sensors available for use in mobile robotics. These include sonars, stereo vision, laser range finders, infrared sensors, wheel encoders and touch sensors, among others. Each sensor has its advantages and drawbacks.

Sonar sensors measure the distance, or range, to the nearest obstacle by emitting a high frequency pulse of sound. This is called active sonar, and is different to the passive sonar used in submarines. Where passive sonar simply listens for sounds, active sonar creates a sound and waits for the sound wave to hit something. Ideally, if the sound wave hits an obstacle, it will reflect back to the sensor. The sonar sensor measures the time between emitting the sound and the reflection returning, divides this by the speed of sound and estimates the range to the object as being that number divided by two (the distance to the obstacle and back).

Sonar sensors have a number of advantages:

- Speed of processing – sonars return a range reading very quickly, requiring little or no processing to determine the position of and distance to an obstacle
- Cheap – in comparison to other sensors such as laser range finders and stereo vision, sonars are relatively inexpensive, leading to their widespread popularity with researchers in robotics.
- Sensing a volume – the sonar beam expands from the sensor in a cone-shaped fashion, covering a wide area. This means it is less likely to miss a small object in its field of view, unlike, say, a laser.

Unfortunately sonars have a number of drawbacks. These include:

- Inaccurate and noisy – an object in the sonars beam does not always cause an echo due to it's orientation, or a beam can bounce off multiple objects before returning to the sensor, giving a reading that is too large. Much of the research in the field of map building involves compensating for this noise.
- Difficult to determine which part of the beam the object is in. Due to the width of the beam, we only know that there is an obstacle *somewhere* in the beam. Additional readings from other positions are needed to identify the obstacles exact position.
- Physically noisy – sonars, based as they are upon emitting a sound wave, make a clicking noise audible to the human ear, which can be intrusive.

Despite the problems associated with sonars, they are the primary sensing mechanism used in this thesis and by the researchers whose work is presented later in this chapter. Their computational economy, usefulness in obstacle avoidance and low cost means that they are present in the vast majority of mobile robots, and therefore worthy of further study.

Laser sensors are similar to sonars insofar as they measure distances using time of flight (TOF) methods. They emit many laser beams which are much narrower than sonar beams, usually with around a 0.5 degree spread, in comparison with a $25^{o} – 30^{o}$ spread for sonars making them much more accurate. However, they have some drawbacks. Laser range finders are considerably more expensive than sonar sensors, and they detect objects in a plane as opposed to the sonars volumetric sensing. This

means that if a object is just above or below the height the laser is at it will not be detected.

Stereo vision is a promising sensing method that uses two cameras placed in different positions capturing images that are then analysed to detect objects. There are many different methods to carry out this analysis, the details of which are beyond the scope of this work. One of the most common methods involves taking simultaneous pictures of the environment and applying an edge detection algorithm to determine where the edge of an object is. The edges in both images are matched with each other and the distance between corresponding images is used to determine the distance to the object. Stereo vision systems have the advantage of being able to detect obstacles that sonar and laser may miss, for example a hole in the ground. There is also work being carried out at the moment to use the colour of objects to identify and separate them [47]. Unfortunately, vision sensors have a number of drawbacks. Firstly, they require considerable processing, since each image contains far more information than a simple range reading. This means that for quick obstacle avoidance, stereo vision is not nearly as useful as sonar or lasers. Secondly they can miss many objects in a picture. A wall, for example, only has edges at either end of it and at doors. A stereo vision system using edge detection will look at a wall, detect both ends of it, and may fail to extrapolate the wall in between the two end points. This is in contrast to sonar and laser range finders that reflect their beams off solid objects. Thirdly, vision systems can function badly in poorly lit areas.

Wheel encoders (also called odometry) count the number of revolutions of the wheels and tells the robot how far it has gone and in what direction. Wheel encoders differ from the three sensors described above in that they do not tell the robot anything new about the environment. Instead wheel encoders provide the robot with information about itself. These very simple sensors are extremely useful for robotic navigation, but unfortunately they suffer from inaccuracies. Phenomena such as wheel slippage, differences in tire pressure and angular drift cannot be measured by the wheel encoders. This means that odometry should be used as a guide to the general distance and angle the robot has travelled, but the robot's estimation of its position must be refined using some localisation method if it is to be sure of its position.

Touch sensors are activated when the robot hits an obstacle and are generally a simple on-off switch. Unlike all the sensors listed above, touch sensors are more or less infallible, if they are pressed then robot has definitely hit something. Unfortunately, if this happens then it means that all the other sensors have failed, so touch sensors can be seen as the robot's last line of defence.

2.8 Map Building Approaches Evaluated Experimentally

Three approaches to map building were implemented and evaluated for this thesis. They were based upon theories put forward in the following publications:

- Moravec and Elfes seminal 1985 paper *High Resolution Maps From Wide Angle Sonar* [46]. This uses a two dimensional gaussian sonar model and a simple occupancy update procedure.
- Matthies and Elfes 1988 paper *Integration of Sonar and Stereo Range Data Using a Grid Based Representation* [42]. This uses a two dimensional gaussian sonar model with a Bayesian probability of occupancy update procedure.
- Konolige's 1997 paper *Improved Occupancy Grids for Map Building* [33]. This uses a sonar model based upon the normal distribution and what the author calls the Multiple-Target Model. It also introduces a method for enforcing the Independence Assumption for sonar readings, called pose buckets.

These three methods are discussed below.

2.8.1 Moravec and Elfes - High Resolution Maps From Wide Angle Sonar

2.8.1.1 Occupancy Grids

Major interest in metric maps was generated with the publishing of Moravec and Elfes' 1985 paper, *High Resolution Maps from Wide Angle Sonar* [46]. Their approach involved taking range measurements from a fixed array of sonar sensors arranged in a circular fashion whose position and angle in relation to each other is known. This range information is then translated onto a two-dimensional map by making the assumption that one of the points along the curved end of the arc is occupied by an obstacle (see Fig 2.6), and incrementing the probability that all points along the curve are occupied. The model they developed for this map was a tessellated spatial random field called an *Occupancy Grid*. Conversely, if it can be assumed that a certain range reading means that an obstacle is in the beam's path at that distance, all points in the body of the beam can be assumed to be free of obstacles, and their probabilities of occupancy decremented appropriately. While a single sonar reading yields very little information, each one changes the map slightly, and with a reading every 100ms, a strong confidence grid is quickly generated.

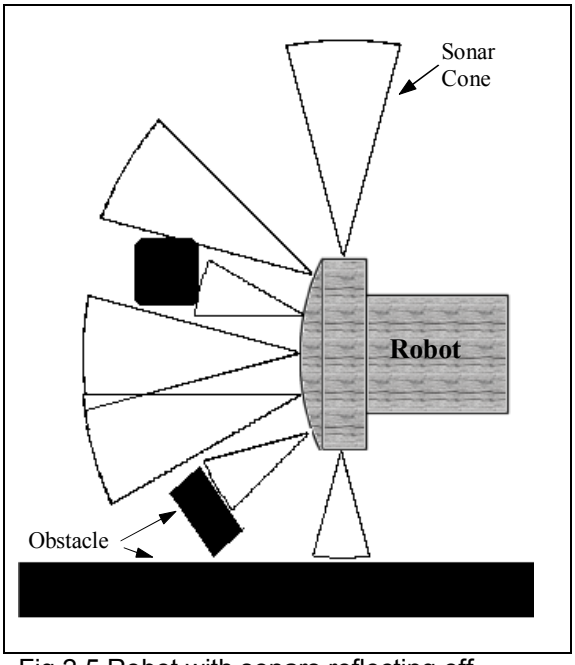


Fig 2.5 Robot with sonars reflecting off obstacles.

Because of the wide beam aperture, in this case 30^{o}, a range reading only gives indirect information about the occupancy of a cell, i.e. we assume that there is an obstacle *somewhere* in the area covered by the range minus the expected sonar error, $R - \varepsilon$, and the range plus the sonar error, $R + \varepsilon$. Also we assume that there is no obstacle in the body of the beam that is closer than the range minus the sonar error. However, our certainty of this fact decreases as the distance from the sonar increases, and the nearer the cell is to the edge of the beam, as the

probability of obstacle detection at the extremes of the beam is less than at the centre axis and near to the sensor. It is necessary to model this when translating a range reading onto a two-dimensional map (see Fig 2.6).

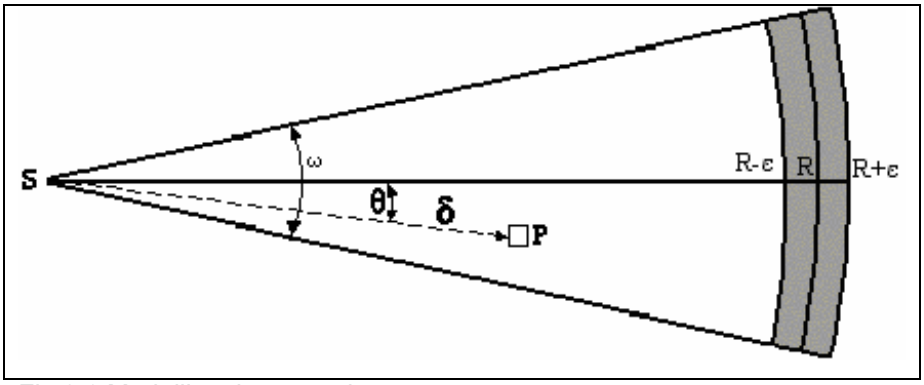


Fig 2.6 Modelling the sonar beam.

Fig 2.6 illustrates Moravec and Elfes' 1985 model of the sonar beam. In this model,

- *S* is the sonar sensor
- *P* is the cell being updated
- ε is the mean sonar deviation error
- ω is the beam aperture, the angle at which the beam spreads from the sensor *S*
- δ is the distance from *P* to *S*
- θ is the angle between the main axis of the beam and the line *SP*.
- *R* is the range reading – the distance the sonar beam travelled before it bounced off an object

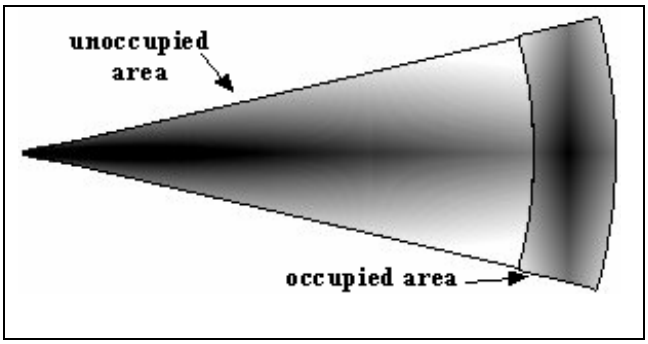


Fig 2.7 A sonar beam probability distribution. The darker the shading the higher the probability value for that point, either $P_E(X,Y)$ for the unoccupied area(on the left) or $P_O(X,Y)$ for the occupied area (on the right). The occupied area is enlarged for clarity.

A system developed using the methods laid down by Moravec and Elfes (see Chapter 4 for the implementation of this paper used in experimentation called *ME85*) uses three maps to store the two-dimensional representation of the world. One stores the probability of each cell being empty, another stores the probability of each cell being occupied, and the final map

stores the combination of the other two. Here follows an explanation of how the Moravec and Elfes method judges the probability of occupancy and of emptiness from a sonar reading. Fig 2.7 above is an example of the result of the following calculations.

The probability of any cell being empty, $P_E(X,Y) = E_r(\delta) * E_a(\theta)$.
$E_r(\delta)$ is the probability of the cell being empty based on the distance of the cell from the sensor. $E_a(\theta)$ is the probability of the cell being empty given the angle between the central line of the beam and the line from the sensor to the cell.

- $E_r(\delta) = \begin{cases} 1 - \left((\delta - R_{min})/(R - \varepsilon - R_{min})\right)^2 & \text{for } R_{min} \le \delta \le R - \varepsilon \\ 0 & \textit{otherwise} \end{cases}$

In plain English, this means that as the cell being updated moves farther away from the sonar, $E_r(\delta)$ becomes smaller, and therefore the probability of it being empty becomes smaller. If the distance of the cell from the sensor, δ, is greater than the range R or is less that the shortest possible range reading R_{min}, then it is not in the part of the beam thought to be empty, so the probability of it being empty is set to zero.

- $E_a(\theta) = 1 - (2\theta/\omega)^2$ for $(-\omega/2) \le \theta \le (\omega/2)$

This means that the closer the cell is to the central beam, the greater the chance that the sonar would have detected an obstacle had there been one, and therefore the higher the probability that the cell is empty. It also means that cells at a greater angle from the sensor than half of the beam width ω are not considered as the beam contains no information about them.

The probability of a cell being occupied, $P_O(X,Y) = O_r(\delta) * O_a(\theta)$.
$O_r(\delta)$ is the probability of the cell being occupied given the distance of the cell from the distance measured by the range reading. $O_a(\theta)$ is the probability of the cell being occupied given the difference in the angle between the central axis of the beam and the line from the sensor to the cell.

- $O_r(\delta) = 1 - \left((\delta - R)/\varepsilon\right)^2$ for $(R - \varepsilon) \le \delta \le (R + \varepsilon)$
- $O_r(\delta) = 0$ if the cell is not in the occupied region (shaded region in Fig 2.6).

This means that the greater the distance the cell is from the actual range reading, the lower the probability that it is the cell that caused the sonar beam to be reflected. Therefore the higher probabilities are grouped at the distance the sonar reported as a range reading, with the lowest values being at the distances R - ε and R + ε.

- $O_a(\theta) = 1 - (2\theta/\omega)^2$ for $(-\omega/2) \le \theta \le (\omega/2)$

This means, as with $E_a(\theta)$, the greater angle between the the central axis of the sonar and the line from the sonar to the cell, the lower the probability that it caused the sonar beam to be reflected. Therefore the highest values will be grouped at the central axis of the beam, with the values decreasing as they approach the sides of the beam.

At this point, after reading a single range measurement, we now have a small amount of information regarding the occupancy of the cells in the sonar beam. As mentioned earlier however, a single reading is not very useful in and of itself. It must be integrated with previous readings to be of any significant use. Moravec and Elfes propose a number of steps to achieve this update. The steps are not symmetrical on the Empty and Occupied maps. This can be explained by the fact that the empty and occupied areas represent two different situations. Since we are assuming that a single cell caused the sonar reflection, the occupied area represents just one occupied cell. We assume that there is definitely an occupied cell in that space, with a probability of 1, but that value must be spread over all the cells in the occupied area of the arc, therefore we normalise the values to sum to one. The empty area, on the other hand, represents a 'solid volume whose totality is probably empty' [46]. This means that we are assuming that there is *no* obstacle in the empty area of the arc, that each of the cells are empty. That is, we claim to know the state of all empty cells, while we only know the state of *one* of the occupied cells. There are some inconsistencies with their approach, which are discussed in section 2.9.3.

To update the Empty map, for every sonar reading we *Enhance* the prior value of the cell in the Empty map $Emp(X,Y)$ with the new value $P_E(X,Y)$ calculated using the formula:

- $Emp(X,Y) = Emp(X,Y) + P_E(X,Y) - Emp(X,Y) * P_E(X,Y)$

For example, if our prior belief that the cell was empty was 0.9, or 90%, and our new belief that it is empty, $P_E(X,Y)$, is 0.4. then our new belief that it is empty is (0.9 +

0.4) – (0.9 * 0.4) = 0.94. It might seem strange to some that integrating a lower certainty with a higher one should yield an even higher value. However it does make sense – the previous value claimed that the cell was empty with a degree of certainty of 90%, and this new sonar reading is confirming that belief not disputing it, so the combined value of both these estimates should increase our confidence in the emptiness of the cell, not decrease it.

The next step is to update the Occupied map. First we *Cancel* the occupied probabilities in response to the supposed emptiness of the cell.

- $P_O(X,Y) = P_O(X,Y) * (1 - Emp(X,Y))$

For example, if the latest reading claims it is 0.8 sure that a cell is occupied, but our previous readings claim that there is a 0.5 chance that the cell is unoccupied, then we recalculate our estimation of occupancy to 0.8 * (1 – 0.5) = 0.4. This means that if a previous reading disagrees with our current one, then we are less sure that this reading is correct, and so reduce it's value by an amount proportional to the strength of the evidence against it.

Secondly, all the occupied cells are *Normalised* to sum to one.

- $P_O(X,Y) = P_O(X,Y) / \Sigma P_O(X,Y)$

This is because it is assumed that only one single grid cell caused the reflection, but it is impossible to be certain which one, so it is necessary to distribute that probability over all the possible cells in the arc so that they all add up to one. Thirdly, for each sonar reading , the *Enhance* step from earlier is repeated. Here $Occ(X,Y)$ on the right of the equation represents the prior belief that a cell *was* occupied, and $Occ(X,Y)$ on the left of the equation represents the newly calculated belief that the cell *is* occupied:

- $Occ(X,Y) = Occ(X,Y) + P_O(X,Y) - Occ(X,Y) * P_O(X,Y)$

For example, if the prior belief in the occupancy of the cell was 0.7, and the new reading claims that it is 0.3, or 30% sure that the cell is occupied, then the new belief that the cell is occupied is (0.7 + 0.3) – (0.7 * 0.3) = 0.79.

Finally, it is necessary to combine both the Empty map and the Occupied map into a single map. This is done with a *Threshold*ing step, where the largest value of the two maps for each cell is chosen for inclusion in the main map.

- Map(X,Y) = Occ(X,Y) if Occ(X,Y) ≥ Emp(X,Y)

otherwise

- Map(X,Y) = Emp(X,Y) * -1

If the belief that the cell is occupied, Occ(X,Y), is greater than the belief that it is empty, Emp(X,Y), then we make the value of our main map equal to Occ(X,Y). Otherwise we make our main maps value, Map(X,Y), equal to Emp(X,Y). This means that, while both the Empty and Occupied maps contain values from 0 to 1, where 0 represents the certainty that it is not Empty and not Occupied respectively, the main map contains values from –1 (definitely unoccupied) to +1 (definitely occupied), with zero representing a lack of, or conflicting, information about the cell.

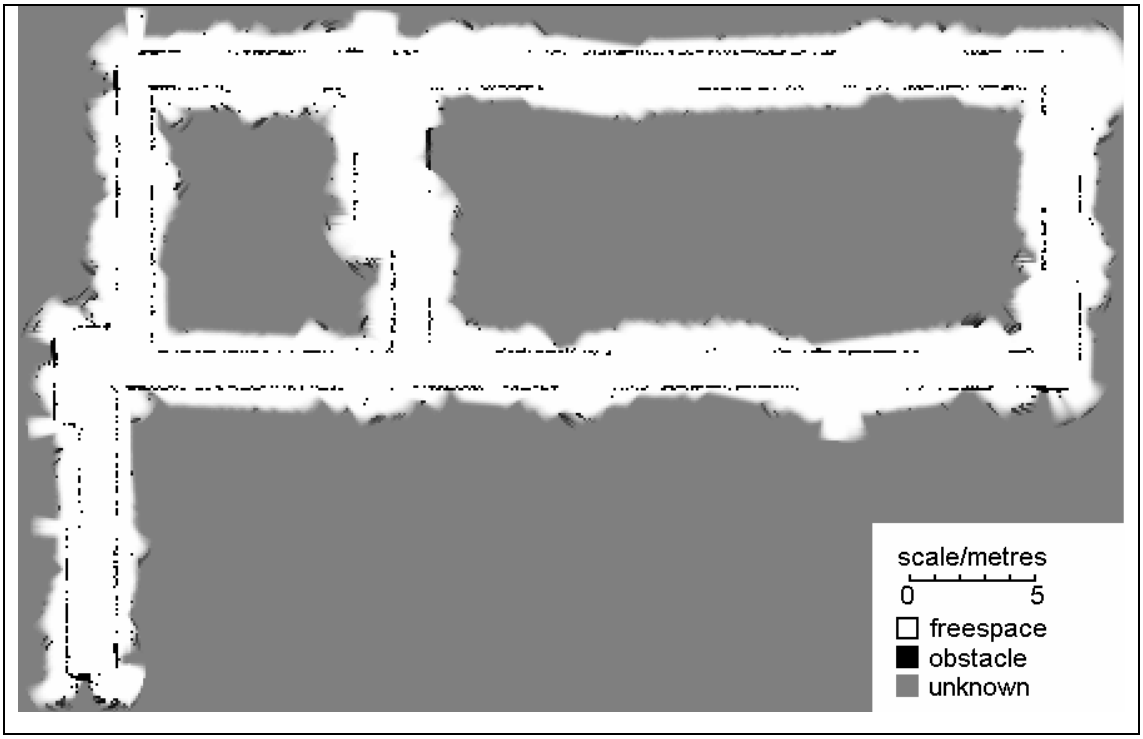


Fig 2.8 Map Generated using Moravec and Elfes' method in a simulated run around the CSIS building's first floor. White areas represent empty space, black areas represent obstacles, while grey areas are unknown. Note that very few walls have been identified due to the lack of compensation for incorrect readings.

2.8.1.2 Issues with Moravec and Elfes' 1985 model

As with most pioneering efforts, there are a number of problems with Moravec and Elfes' original mapping algorithms. The two main weaknesses lie in the assumptions made about the properties of the sonar reflection. These were:

1. If an object is in the sonar beam, it will cause a diffuse reflection back to the sonar emitter, regardless of it's orientation.
2. If one object causes a reflection, there is no other object in the sonar beam.

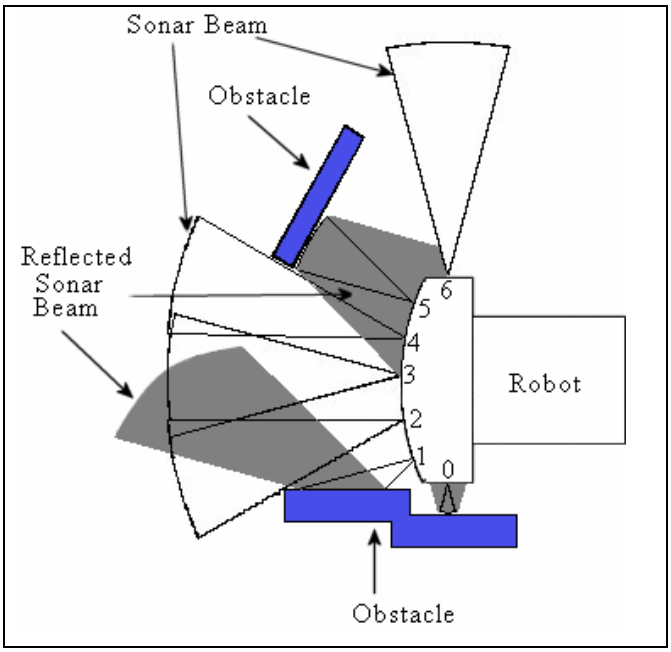


Fig 2.9 Specular reflection, with the signal emitted by sonars 0 and 5 returning a correct reading, and the beam from sonar 1 being reflected away at an angle and returning no reading to the emitter.

The first assumption ignores the large problem of *specular reflection* (see Fig 2.9). There are two types of reflection, diffuse and specular. Diffuse reflection occurs when the surface roughness is larger than the wavelength of the beam, which causes the energy to be refracted in all directions. In this case, the emitter receives a correct reading, within certain error bounds. Specular reflection occurs when the surface is smooth, in which case the energy is not scattered. Rather, it is reflected at a similar angle to which it struck the object. If the object is at an angle sufficiently far from perpendicular to the emitter, typically around 25^{o} to 30^{o}, the energy will reflect away from the emitter. This results in either no return signal being received by the sensor, in which case the space occupied by the obstacle is marked as empty, or giving a spurious reading after bouncing off multiple surfaces, in which case an area that may not be occupied is labelled as such.

The second problem with this approach is the *single-target assumption*. There are two facets to this. The first is that Moravec and Elfes method assumes that if a sonar beam bounces off an object, only one cell along the edge of the beam is occupied. The second is that the reading received by the emitter is merely the *first* signal returned with any following signal being ignored. This means that Moravec and Elfes' mathematical model does not consider the fact that there might be other objects in the sonar beam that might be reflecting it, as with the upper obstacle in Fig 2.10 which is large enough to occupy more than a single cell.

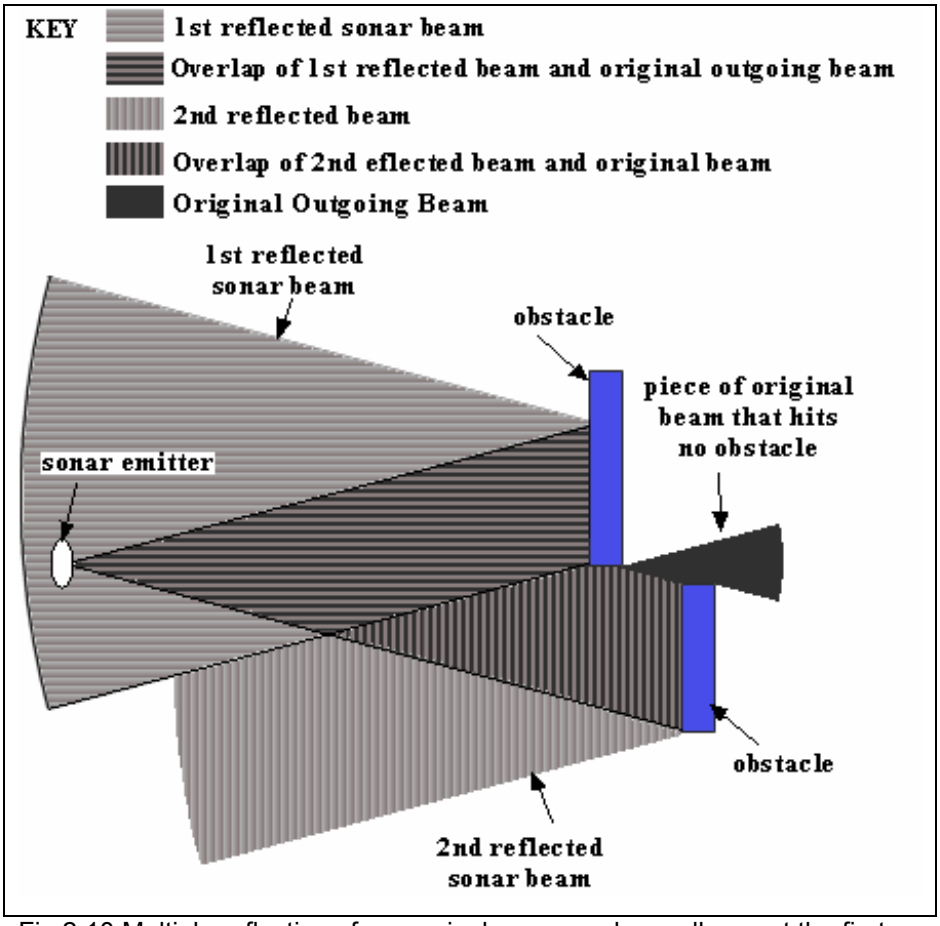


Fig 2.10 Multiple reflections from a single sonar pulse – all except the first are ignored.

Another object can be further away than the nearest obstacle, as in the lower obstacle in Fig 2.10, in which case the reading arrives back at the sensor later and is automatically ignored. There can also be another object at the same distance as the reflecting object, and reflecting at the same time. Either way, the model used here normalises all values in the occupied area of the beam to one. A proposed solution to this problem, the MURIEL method by Kurt Konolige, is discussed later in section 2.8.3.

While there are many shortcomings to the original paper by Moravec and Elfes, it must be said that it constitutes an excellent platform to build upon, and as such is the starting point for many researchers in the field. For this reason, as the first iteration of experimentation, a working version of the system proposed by Moravec and Elfes [46] was developed, without the map-matching (localisation) capabilities. Results were taken of the operation of this system, then modifications were made to the model

to counter some of the deficiencies in the original theory. These are explained in depth in Chapter 4.

2.8.2 Matthies & Elfes – Integration of Sonar and Stereo Range Data Using a Grid Based Representation

In 1988, Alberto Elfes followed up his earlier Occupancy Grid work with a paper on the integration of stereo vision data and sonar data. The key contribution of this paper was the change of the update function.

2.8.2.1 A Framework For Combining Sensor Data From Multiple Senor Types

The new framework is an enhanced version of the previous occupancy grids, but with the added capability to fuse the readings from many different types of sensors. The system, which can be seen in Fig 2.11, begins each cycle by taking in readings from the sensors. The *Spatial Interpretation Model* converts the sensor reading into a statement about the occupancy of certain parts of the grid, and the *Sensor Uncertainty Model* applies the conditional probability density function (*cpdf)* $p(sensor\ reading\ R | state\ of\ environment)$ to the reading in order to model our confidence in the sensor reading. This *cpdf* is represented in the later formulas as $P(e \mid s_i)$ which in English means: *what is the probability of receiving this piece of evidence* e*, given that the cell is in state* s_i*?*. While the paper strangely never fully elaborates on the exact function used, from the diagrams provided it can be assumed that the authors are using the function from Elfes' earlier 1985 paper with Hans Moravec [46] for calculating P_O(X,Y) and P_E(X,Y), the probability of a cell being occupied or empty respectively. These are discussed in the previous section. This generates what is called the *Sensor View*, which is our interpretation of what the sensor is seeing.

Unfortunately it can never be determined with absolute certainty the position the robot was in when the sensor reading was taken in relation to where it was when previous readings were taken, and this uncertainty must be incorporated into the model. The *Sensor Position Uncertainty Model* performs this function. The robot has, through

some localisation mechanism, a measure that denotes its certainty that it is in a particular position. This is incorporated into the Sensor View to create the Robot view – our belief of what the robot is seeing.

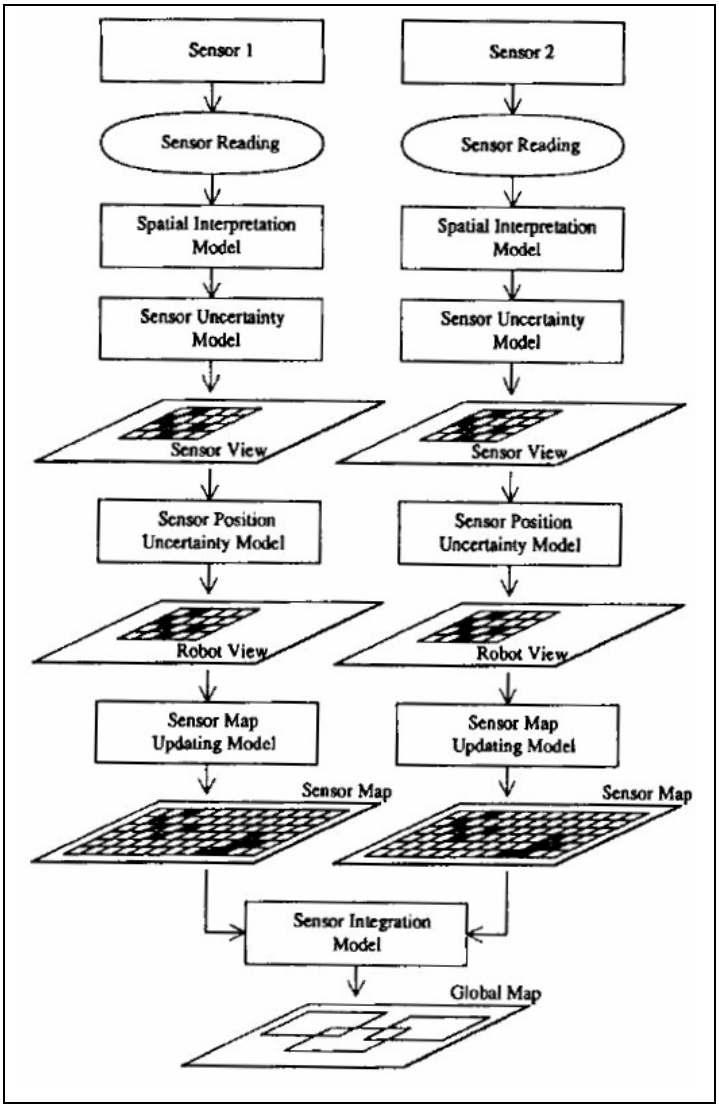


Fig 2.11 A Framework for Occupancy Grid-Based Sensor Integration.

The final two stages focus on integrating the various sensor views. The first stage, the *Sensor Map Updating Model*, combines all robot views formed from the same type of sensor into a single view. For example, with an array of sonar, a separate view would have been created for each emitter, and these would all be integrated to form a single *sonar-view*. The final step, the *Sensor Integration Model*, integrates all the views from the different types of sensors to form a Global Map. It does this using the Bayesian update approach explained below.

2.8.2.2 Using Bayesian Probability Theory To Update Map Values

Despite the usefulness of the framework Elfes designed, the most significant advance was the development of a Bayesian Model for updating the Occupancy Grid. With this new model, the old method of updating the occupancy grid described in the 1985 paper is abandoned in favour of a more accurate, mathematically robust model. For

this paper it is necessary to calculate is the state of a particular grid cell given a piece of new evidence. We begin with Bayes basic update theorem:

$$P(s_i \mid e) = \frac{P(e \mid s_i)P(s_i)}{\sum_j P(e|s_j)P(s_j)}$$

where s_i is the state being estimated, and e is the new evidence, such as a sonar reading. $P(s_i)$ is the prior probability of the cell having a certain value, either occupied of empty. What is required to calculate is the probability of the state of a cell being occupied, s(C)=OCC, or the state of a cell being empty s(C)=EMP, given the new range reading, i.e. P(s(C)=OCC | R) and P(s(C)=EMP | R), which will from now on be referred to as P(OCC | R) and P(EMP|R) respectively. When updating the prior probability with a new value, Bayes theorem can be expressed as:

$$P(OCC \mid R) = \frac{P(R \mid OCC)P(OCC)}{P(R \mid OCC)P(OCC) + P(R \mid EMP)P(EMP)}$$

where P(OCC) and P(EMP) are the prior probabilities. But since the probability of a grid cell being occupied is equal to one minus the probability of it being empty, it can be said that

P(EMP) = 1 – P(OCC) and P(R | EMP) = 1 – P(R | OCC)

and vice versa. The above formula then simplifies down to:

$$P(OCC \mid R) = \frac{P(R \mid OCC)P(OCC)}{P(R \mid OCC)P(OCC) + (1 - P(R \mid OCC))(1 - P(OCC))}$$

with the need to calculate the empty probability factored out. P(EMP | R) can be calculated in the same way. To carry out this computation, it is only necessary to compute the sonar model, $P(R|OCC)$, which is a simple Gaussian noise dispersion model like the one used in Elfes 1985 paper [46] to calculate $P_E(X,Y)$ and $P_O(X,Y)$ as shown in Fig 2.12. All other terms are lookups in a table.

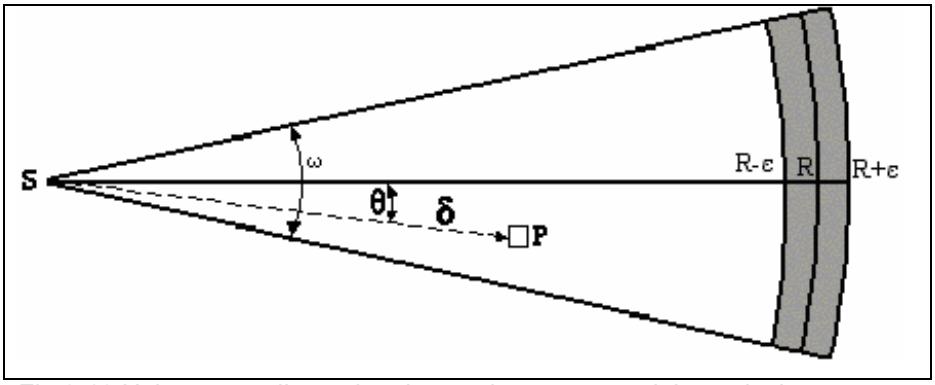


Fig 2.12 Using a two dimensional gaussian sonar model to calculate *P*(*R*|*OCC*) – see section 2.9 for discussion of this model.

2.8.2.3 Advantages To Using Bayesian Map Updates

There are three advantages to using this update rule.

1. It is both associative and commutative, so that data in a multi-sensory system can be incorporated into the model in any order.
2. If a grid cell has a value of UNKNOWN, P(OCC) = 0.5, integrating new evidence E = P(R | OCC) gives E as a result.
3. Conflicting measurements cancel each other out. If evidence of equal strength is received for both OCC and EMP, a value of UNKNOWN is produced.
4. Forms the basis for more complex, mathematically robust formulae, such as that developed by Konolige in [33] and presented in section 2.8.3.

2.8.2.4 Disadvantages To Using Bayesian Map Updates

There are two key disadvantages to using the Bayesian update formula. The first is that a single update can change the occupancy value of a cell drastically. For example, given a prior probability of 0.5, and a new update of 0.1, the posterior probability of occupancy is 0.1.

$$P(OCC \mid R) = \frac{0.1 * 0.5}{0.1 * 0.5 + (1 - 0.1)(1 - 0.5)} = \frac{0.05}{0.05 + 0.45} = 0.1$$

A single incorrect specular reading could therefore alter the map considerably. An approach that used smaller increments is desirable, where a map is built using many sonar readings, rather than just one or two readings being used to determine the occupancy of a cell.

The second disadvantage to using the Bayesian update rule is that once $P(OCC)$ converges to either one or zero it cannot be changed. For example, take the scenario where the previous occupancy value of a cell is zero:

$$P(OCC \mid R) = \frac{P(R \mid OCC) * 0}{P(R \mid OCC) * 0 + (1 - P(R \mid OCC))(1 - 0)} = \frac{0}{1 - P(R \mid OCC)} = 0$$

No matter what value is given for $P(R \mid OCC)$, once the prior probability is zero, it will remain zero. The same holds true for when the prior probability is one. This issue is related to the first disadvantage mentioned above. If a small number of specular readings can alter the occupancy values to a large degree, and once a cell has converged to zero or one it cannot be changed, then it will be impossible to compensate for noisy readings with a sufficient number of accurate readings.

2.8.2.5 Implementation Details

Matthies and Elfes' paper is implemented in the *ME88* module, which is described fully in chapter 4, with experimentation results, both simulated and real-world, being presented in chapter 6. A modified version of the paper, called *ME88mod* is also described in chapter four.

2.8.3 Konolige - Improved Occupancy Grids for Map Building

Occupancy grids, as previously discussed, are a probabilistic method for fusing multiple sensor readings into surface maps of the environment. One of the main challenges is how to apply occupancy grids to real-time sensor interpretation. Here follows a brief discussion on the significance of refined sensor models and independence assumptions are crucial issues with respect to occupancy grid interpretation.

The unreliability of sensor readings has caused many problems for those working with mobile robots. Occupancy grids are one of the most popular ways of compensating for this. Using this method, space is divided into a regular grid, with an estimate of the probability of each cell being occupied being calculated. Two common problems that sonar sensors suffer from are:

- *specular reflection*, where the energy from the device is reflected off several surfaces before returning to the device, or is reflected off a the nearest surface at a wide angle and never returns to the emitter.
- *redundant information*, where it is assumed that each new sonar reading gives new information, when often it is simply repeating what sensed previously.

The MURIEL method (Multiple Representation, Independent Evidence Log) for updating occupancy grids was developed by Kurt Konolige [33] in an attempt solve these problems. This method splits the sensor model into two parts. Firstly, it adjusts the model mixture dynamically for each new reading, resulting in a better estimation of occupancy. The model mixture is the degree to which we believe that the sonars are giving incorrect data. Rather than simply estimate that, for example, the sonars are wrong 50% of the time, the MURIEL method incorporates algorithms for dynamically estimating the fitness of a sonar reading. Secondly, it merges the readings from multiple positions, or poses, to gain more independent information about a particular cell. This means that if, for example, a robot is sitting in a single position and takes 100 readings, only the first reading is counted since all following readings tell us nothing new about the environment since they are merely repeating what the original reading claimed, or at least something very close to it. The MURIEL method can recall, for each cell in the map, what position the robot was in when it changed the value of that cell. If the current reading is from a position the robot has already used to change that cell, the reading is discarded.

2.8.3.1 Previous Work on Fusing Multiple Sensor Readings

There are two broad categories for fusing multiple sensor readings for map-making: target tracking and occupancy grid models. Target tracking involves modelling one or more features, and estimating their position at each new sensor reading. This work uses Kalman filters, which is a computational algorithm that processes measurements to deduce an optimum estimate of the past, present, or future state of a linear system by using a time sequence of measurements of the system behaviour, plus a statistical model that characterizes the system and measurement errors, plus initial condition information. Target tracking is appropriate when there are few landmarks and their interaction with the sensors is well known. A key issue in this field is the data-

association problem. That is, how to identify the target that a given sensor reading is associated with. One possible solution is to use a Bayesian tree approach to processing multiple hypotheses about data associations.

Target tracking methods are inappropriate in situations where it is important to determine the complete surface geometry of the environment. In these cases, occupancy grids provide a better solution. Similar to target detection, the primary problem of occupancy grids is data association, i.e. does a sensor reading give information about surfaces in a particular area? The geometric uncertainty of the beam width and multiple reflections of the beams energy are the main causes of error. Elfes [20] compensated for these errors by using a Bayesian update with gaussian noise with a very large variance.

Although Elfes' work succeeded in improving occupancy grid models, some problems remain unsolved. First, modelling multiple reflections as gaussian distributed is not realistic, and second, the use of a gaussian distributing implies an *averaging* model, in which every reading is assumed to be corrupted by the same noise. Finally, enumerating and updating probabilities for all possible environmental situations is computationally prohibitive, simplifying independence assumptions are made to reduce the computational complexity.

The MURIEL method addresses the problems above by introducing a multiple target detection model, which assumes a random (unbiased) distribution of surfaces in the environment. To solve the data association problem, the method asks under what conditions the model gives the assumed random distribution of surfaces. This way, sensor readings are only treated as independent if they come from different poses.

2.8.3.2 Probabilistic Sensor Models

A sensor model describes how a sensor interacts with the environment. An ideal sensor would give perfect information about the properties it reports on, but this is seldom if ever the case. The uncertainty of sensor readings can be expressed using probabilistic methods, specifically Bayes rule:

$$P(A \mid B) = P(B \mid A)\frac{P(A)}{P(B)}$$

where P(A) and P(B) are the prior probabilities of A and B. The sensor model is the quantity P(B | A), the probability of getting the measurement B given that the environment has property A. As it can be inconvenient to determine P (B), the equation can be rewritten as:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B \mid A)P(A) + P(B \mid \overline{A})P(\overline{A})}$$

Change of odds is often a much more intuitive quantity to deal with than the absolute probability, because it factors out the prior probabilities in a nice way. There odds of an event occurring is there probability of it occurring divided by the probability of it not occurring:

$$O(A) = \frac{P(A)}{P(\overline{A})}$$

The likelihood odds can be represented by

$$\log O(A \mid B) = \log \lambda(B \mid A) + \log O(A)$$

where $\lambda(B \mid A)$ is the *likelihood ratio of B given A*. This ratio can be calculated with the formula:

$$\lambda(r = D|C_i) = \frac{P(r = D|C_i)}{P(r = D|\overline{C}_i)}$$

where $P(r = D|C_i)$ is the probability of the range reading *r* being equal to distance *D* given that the cell C_i is occupied, and $P(r = D|\overline{C})$ is the probability of the range being a distance *D* given that the cell C_i is unoccupied. Both of these figures can be calculated theoretically using the following two points of information on the properties of sonars [33]:

1. The range error becomes larger as the range increases
2. The probability of an obstacle being detected becomes smaller as the range increases.

Earlier sensor models such as Elfes' 92 [20] model made the simplifying assumption that a single target caused the sonar reflection returned to the emitter, and that no other target can exist. Occupancy values are updated to reflect this, with points both in front of and *behind* the obstacle having their occupancy values lowered. However,

this is not always the case, and rather than a single target being present, we observe that the echo we receive is merely the *first* to come back, with later echoes being ignored by the emitter. This must be accounted for in the sonar model, as the more refined and accurate our understanding of the operation of the sensors the more accurate the world model derived from them will be.

2.8.3.3 The single target model

The full two-dimensional multi-target model is quite complex, so we'll begin with the basics, and build it up slowly. The single target mathematical model for obstacle detection in the 1-D case, not taking into account the angle of the sonar beam, is:

$$p_1\left(r = D\middle|C_i\right) = \frac{\alpha(r_i)}{\sqrt{2\pi}\delta(r_i)} e^{-(D-r_i)^2/2\delta(r_i)^2}$$

where $\alpha(r_i)$ is the attenuation of detection with distance, $\delta(r_i)$ is the range variance, which increases with distance, and r_i is the distance of the target from the sonar. To calculate the range variance, Konolige uses a value of $\delta(r) = .01 + 0.015r$, which means that there is a fixed error of 1cm, plus 1.5% of the range. This was calculated for Polaroid sonar sensors, and may have to re-optimised for other models. To calculate the detection attenuation, the formula $\alpha(r) = 0.6(1 - min(1,0.25r))$ was used.

The 0.6 means that objects are sometimes not detected, even at close range. This model attenuates linearly with distance, and has a limit of 4 metres for detection. If a shorter maximum distance is required, the formula can be altered to accomplish this.

The formula above for $p_1\left(r = D\middle|C_i\right)$ looks complicated at first glance, but it is merely a modified version of the formula for the standard normal distribution curve, the most common distribution used in statistics. A continuous random variable that is normally distributed, when plotted on a graph, forms a curve whose area between it an the x-axis is equal to one. The formula for the normal distribution is as follows:

$$f(x;\mu;\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/(2\sigma^2)} \quad -\infty < x < \infty$$

where μ is the mean of the curve (i.e. the value of X for which the curve has its largest Y value), and σ is the standard deviation from the mean. σ governs the distance from

the mean μ to the inflection points of the curve (i.e. the value at which the curve stops curving downwards and starts curving upwards), which in essence governs how tall and how wide the bell curve is. Konolige adapted this formula, replacing σ with $\delta(r_i)$, μ with r_i, x with *D* (the distance measured by the sonar) and multiplying the result by $\alpha(r_i)$.

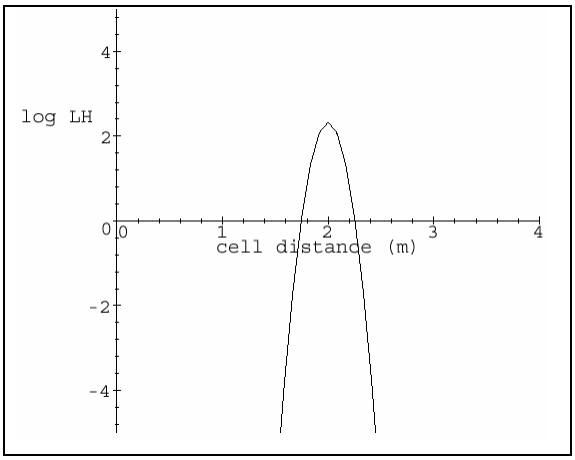


Fig 2.13 On-axis log likelihood ratio in the single target model for a range reading of 2 metres.

Unfortunately, as mentioned previously, and as can be seen in Fig 2.13, the single-target model treats points both in front of and behind the obstacle the same. This is clearly not accurate, as no information regarding the area behind the obstacle should be derived from a range reading due to the fact that it cannot affect the outcome of the reading. For this reason the multiple-target model was developed.

2.8.3.4 The multiple target model in one dimension

The multiple target model does not make the simplifying assumption that only one target can cause a reflection of the sonar beam. While only one reading per clock cycle is actually processed, the sensor could possibly receive other readings. It is assumed that these occupied surfaces have a random distribution, since there is no prior information about them. The only effect of this assumption is to add a small constant, *F*, to the probability density $p(r = D|C_i)$, where *F* is a estimation of the probability of detecting random other targets.

$$p_{1m}(r = D|C_i) = \frac{\alpha(r_i)}{\sqrt{2\pi}\delta(r_i)} e^{-(D-r_i)^2/2\delta(r_i)^2} + F$$

This causes the likelihood ratio to be changed to

$$\lambda_{1m}(r = D|C_i) = \frac{p_{1m}(r = D|C_i)}{F}$$

where once again, the target is distance r from the sensor, $\alpha(r_i)$ is the attenuation with distance, and $\delta(r_i)$ is the range variance. The result of placing the extra term in the formula is that the log likelihood ratio becomes 1 (no change) everywhere except near the range reading, with the freespace hypothesis ($\log \lambda < 0$) disappearing. This is because our assumption means that although there is an object at the current range reading, we are not ruling out there being an obstacle anywhere else. However, it is known that the sensor reading being processed is the *first* received by the emitter, and this must be reflected in the model. To do this, we conclude that if this is the first reading, no reading of a distance less than r has been received. Therefore multiply together the probability of getting a range reading at r, $p_{1m}\left(r = D|C_i\right)$ and the probability of getting no reading less than r, $p_{1m}\left(r \not< D|C_i\right)$. The new likelihood ratio, $\lambda_{1m}\left(r @ D|C_i\right)$, is then written as:

$$\lambda_{1m}\left(r @ D|C_i\right) = \frac{p_{1m}\left(r = D|C_i\right)p_{1m}\left(r \not< D|C_i\right)}{p_{1m}\left(r = D|\overline{C}_i\right)p_{1m}\left(r \not< D|\overline{C}_i\right)}$$

To calculate $p_{1m}\left(r \not< D|C_i\right)$, the probability density function is integrated up to distance D, and subtracted it from 1:

$$p_{1m}\left(r \not< D|Q\right) = 1 - \int_0^D p_{1m}\left(r = x|Q\right)dx$$

with Q = C_i or $\overline{C}_i$.

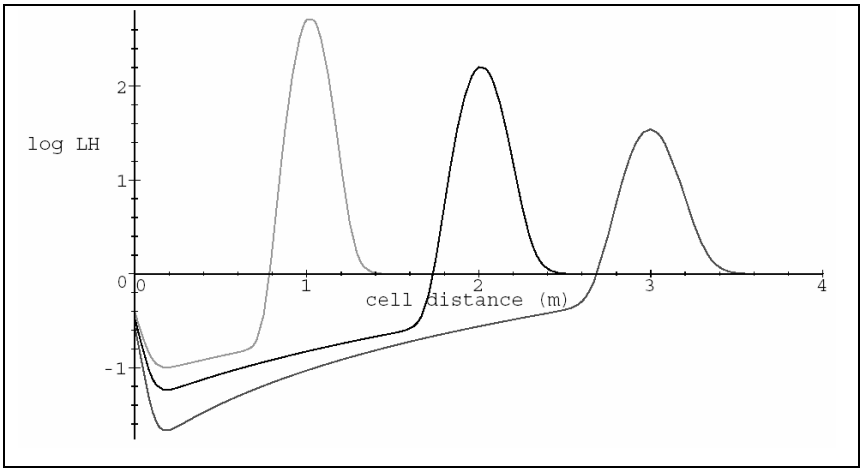


Fig 2.14 On axis log likelihood ratio for sonar range readings at 1, 2 and 3 metres using the multiple target model. The Y axis value is used to update the cell in the map.

An advantage to using the normal distribution becomes apparent when integrating the above function. Rather than having to laboriously sum all the values of the cells less than a given distance from the sonar in the sonar beam, the standard normal distribution has been tabulated, and can be used to calculate the integrated value for any normal distribution. Using this method, calculating $p_{1m}(r \not< D|C_i)$ is reduced to doing a lookup in a table of values, as well as a multiplication and subtraction to convert the value from the standard normal distribution to the normal distribution required using the formula:

$$Z = \frac{X - \mu}{\sigma}$$

where if X has a normal distribution with mean of μ and a standard deviation of σ, the Z is the equivalent standard normal variable.

2.8.3.5 The multiple target model in two dimensions

Up until now, we have been dealing with just the one dimensional case, not taking into account the width of the beam. To correct this, we simply add in a gaussian decay, parameterised by the angle, from the central beam of the sensor. The probability of obstacle detection in two dimensions then becomes:

$$p_{2m}(r = D|C_i) = \frac{\alpha(r_i)}{2\pi\delta(r_i)\sigma} e^{-\theta_i^2/2\sigma^2} e^{-(D-r_i)^2/2\delta(r_i)^2} + F$$

with θ being the angular deviation of the target from the central beam, and σ being half the beam width, e.g. $\sigma = 15^o$ for a beam width of 30^o. All the mathematics from the one dimensional model are also valid here, with $p_{2m}(r \not< D|Q)$ becoming

$$p_{2m}(r \not< D|Q) = 1 - \int_0^D \int_{-\pi}^{\pi} p_{2m}(r = x|Q) d\theta dx$$

with Q being either C or $\overline{C}$.

While $p_{2m}(r = D|C_i)$ seems quite complex, it is possible to simplify it somewhat by noting that it is essentially the amalgamation of two *normal distribution* formulae, the first modelling the probability of occupancy of a cell based on the distance of the cell from the sonar range measurement, and the second modelling the distance of the cell from the centre of the beam. The formula can be split into separate parts as follows:

$$p_{2m}(r = D|C_i) = \alpha(r_i)\left(\frac{1}{\sqrt{2\pi}\delta(r_i)}e^{-(D-r_i)^2/2\delta(r_i)^2}\right)\left(\frac{1}{\sqrt{2\pi}\sigma}e^{-(\theta_i-0)^2/2\sigma^2}\right)+F$$

When this is compared to the formula for the normal distribution

$$\frac{1}{\sqrt{2\pi}\sigma}e^{-(x-\mu)^2/(2\sigma^2)}$$

it can be seen that the first set of brackets is simply the normal distribution with parameters r_i for the mean, $\delta(r_i)$ for the standard deviation, and the sonar range measurement D instead of x. The second set of brackets is also the normal distribution with a mean of zero, half the width of the sonar beam, σ, as the standard deviation, and the angle between the cell, the sonar emitter and the central beam of the sonar wave θ as inserted in place of x.

Integrating this formula, as explained earlier, can be done by simply looking up the tabulated values for the standard normal distribution and converting them for the given μ and σ. The doubly integrated formula:

$$\int_0^D\int_{-\pi}^{\pi}\alpha(r_i)\left(\frac{1}{\sqrt{2\pi}\delta(r_i)}e^{-(D-r_i)^2/2\delta(r_i)^2}\right)\left(\frac{1}{\sqrt{2\pi}\sigma}e^{-(\theta_i-0)^2/2\sigma^2}\right)+F$$

can be separated out as follows:

$$\alpha(r_i)\left(\int_0^D\frac{1}{\sqrt{2\pi}\delta(r_i)}e^{-(D-r_i)^2/2\delta(r_i)^2}\right)\left(\int_{-\pi}^{\pi}\frac{1}{\sqrt{2\pi}\sigma}e^{-(\theta_i-0)^2/2\sigma^2}\right)+F$$

since the first integral will only affect variables to do with distance (r_i, D, $\delta(r_i)$), and the second integral only affects variables to do with angle from the centre of the beam i.e. θ. The function $\alpha(r_i)$ could be left in the first integral, but it would make no difference to the result, and for the table lookup to work the formula must take the form of a normal distribution. Therefore, $\alpha(r_i)$ is taken outside both integrals. The value F is placed outside both integrals also, and for the same reason as $\alpha(r_i)$ - it would make no difference to the overall result, but would cause either formula to not take the form of a normal distribution and it would then be impossible to use the tabulated lookup.

2.8.3.6 Specular and Diffuse Models

Specular reflection, as described in section 2.9.3, occurs when a smooth surface reflects the sensors energy coherently. If the surface is at an oblique angle to the sensor the emitted energy will either give no reading or an incorrect one. Whatever the case, it is erroneous to treat the reading as valid. It is therefore desirable to be able to detect and ignore incorrect readings. As in all our dealings with sonar sensors, it is only possible to estimate a probability that a reading is specular. We can say that $p(S)$ is the probability that a reading is specular, and that $p(\overline{S})$ is the probability of it not being specular, i.e. that it is diffuse. However, since specular and diffuse readings are mutually exclusive, $p(\overline{S}) = 1 - p(S)$.

So now it can be said that the information to be extracted from any reading is equal to the probability of detection, $p_{2m}(r = D|C_i)$, multiplied by the probability of a reading being diffuse, $1 - p(S)$, plus the amount of information given by a specular reading multiplied by the probability of the reading being specular. Since a specular reading provides no information about the environment, the information provided is

$$p_{2s}(r = D|C_i) = p_{2s}(r = D|\overline{C}_i) = F$$

Therefore, our new update formula for a sonar reading is:

$$p_c(r = D|C_i) = p_{2m}(r = D|C_i)(1 - P(S)) + p_{2s}(r = D|C_i)P(S)$$

The question must now be asked, what value does $P(S)$ take? While it is possible to give it a fixed value by tuning the sensor model for the environment, this does not take into account the local environment, with some areas being more likely to create specular reflections than others. One way of deciding if a sonar reading is specular is to use Drumheller's [19] *sonar penetration condition* which states: the freespace hypothesis of a sonar reading should not impinge on a high-confidence surface i.e. if we are sufficiently confident that a cell is occupied we should not believe any reading that claims that the cell is empty. If prior information regarding the environment exists, it can be used to derive a probability of specularity.

The MURIEL method uses the following algorithm to compute the probability of specularity, $P(S)$. The basic idea is that if sufficient readings are received stating that the cell is occupied, it can be assumed that readings claiming it to be empty are specular reflections. This is implemented by first summing up all the *log* λ surface readings at the cell. This figure is called *log* λ_s. Once this number passes a certain threshold, C_s, $P(S) = 1$. If *log* $\lambda_s = 0$, $P(S) = 0$. If *log* λ_s is in between zero and C_s, $P(S)$ is calculated by linear interpolation.

An advantage to this approach is that it is quick to calculate, and gives reasonably good results. A disadvantage to it is that it relies on prior information from multiple different angles to determine the specularity of a reading, and this may not always be available. For example, when exploring a corridor for the first time, it is usually only possible to get a few readings of a wall as it is passed by the robot, as it is not practical to rotate the robot every few seconds to compare sensor readings. This is especially apparent when the robot is exhibiting a wall-following behaviour, as is common. Multiple readings from diagonal sensors will accumulate incorrect specular readings as the robot approaches a particular wall segment, while relatively few correct readings will be received by side sensors, maybe even just a single correct reading, as the robot passes the wall segment. A possible solution to this is to set the threshold C_s, very low, so very few surface readings will negate many freespace readings. This could lead to many cells being marked occupied that are in fact empty, however. See chapters 4 and 6, for implementations and results of testing this theory respectively

Another problem is that all freespace readings that impinge on a single cell with a value of *log* $\lambda_s > C_s$ have the same probability of specularity. Obviously this is not accurate, with a freespace reading that covers many occupied cells being more likely to be incorrect than one that includes just one occupied cell. It may be possible to use a modified function of $P(S)$ that takes into account the percentage of occupied cells in relation to the total number of cells covered by the beam.

2.8.3.7 Independent Evidence – Ignoring redundant readings

As stated earlier, a simplifying assumption of conditional independence has been made that states that each cell in a map has no effect on another cell, and that each reading is independent of all other readings. While this is not entirely accurate, attempting to model a non-independent system is an almost impossible task due to the exponential number of variables it is necessary to keep track of. Using the idea of conditional independence, we build up a map by taking sensor readings from many different positions and angles. However, take the case when the robot is stationary and taking in multiple readings in a static environment. With each reading taken, more information is added to the map, reinforcing the set of previous range readings that have not changed. The problem with this is that no new information is really being added to the map after the first reading. Each successive reading is clearly not independent of the one that came before – it is exactly the same. If a mathematical model is used that assumes independence in respect to the readings, we will have to use some method of identify the redundant readings that contribute nothing new to the map, in order to ensure that the information is truly independent.

Konolige's MURIEL method uses a map which uses a dual representation – each cell represents the both the occupancy of the area and the pose of readings that have effected that cell. This is accomplished by using what he calls *pose buckets*, which essentially store a Boolean variable stating whether a reading from a given distance and angle has affected a particular cell. This Boolean is set to *true* when the first reading from a pose is received, and all following readings from that pose for this cell are discarded, as they merely duplicate information already in the model.

This method addresses the problem mentioned above, and ensures that all information included in the map is independent. However, there is a problem that Konolige neglected, and which needs to be addressed. It is that some useful information is possibly being discarded. This method is also unable to cope with dynamic environments. Take for example, if the robot detects a chair in a certain position, and builds up a sufficiently high confidence that it exists. If that chair is removed, the system will ignore all readings that report this fact. Clearly this is not good enough.

New information must be able to be dissimilar with earlier readings without being discarded.

One possible way of doing this is to bias the system towards believing the sensors when they claim that a cell previously thought to be occupied was in fact empty. This way, if a surface reading was originally received for a cell *C* from pose *P*, and later a freespace reading for *C* from *P* is returned, the likelihood of occupancy is reduced by an amount equal to the contribution made by the previous surface reading to increasing the likelihood of occupancy. The freespace likelihood is increased accordingly. This can be seen as analogous to a blind man feeling his way with a stick. If he taps an obstacle when in one position, walks somewhere else, then later returns to the same position and no longer feels the obstacle with his stick, it is reasonable to assume that although the obstacle *was* there, it is not any more. This is of course conditional on his ability to return to the same position and orientation, but since Konolige's paper doesn't deal with localisation, we can assume that a separate module is taking care of that function. See Chapter 4 for details of the implementation of this technique in the *K97* module, and Chapter 6 for the results of experiments and comparisons with other techniques, as well as with the original MURIEL method.

2.8.3.8 The MURIEL algorithm

The incremental method used by the MURIEL algorithm, is as follows:

1. Collect data. Check each reading against the pose bucket to see if it is duplicating any information. Two sets of pose buckets are kept for each cell, one for surface readings, and one for freespace readings. If the reading is not duplicating previous information, the pose bucket for that pose and that cell is marked *true*.
2. Update the likelihood ratios λ_s and λ_f. When a new surface reading comes in, calculate the new likelihood ratio using the method described earlier for calculating $p_{2m}\left(r = D \middle| C_i\right)$, and multiply it with the old one, λ_s. Similarly, when a new freespace reading comes in, compute the log likelihood of

freespace at the cell, $\log \lambda_f$, using the formula for $p_c(r = D|C_i)$ given earlier, and multiply it with the old ratio λ_f.

3. Calculate the probability of specularity, $P(S)$.
4. The odds of cell occupancy are calculated using a simple addition and multiplication, $\log \lambda_T = \log \lambda_s + \log(\lambda_f(1 - P(S)) + P(S))$.

This method of updating the log likelihoods by multiplication is used for efficiency, and is not as accurate as completely recalculating λ_s and λ_f at each clock cycle. As such it is an approximation, but the gain in performance is considerable, and an acceptable trade-off with accuracy.

2.8.3.9 Issues with MURIEL algorithm

One of the most significant problems with the MURIEL method is the way in which it calculates the probability of specularity of a sonar reading, $P(S)$. For a single sonar reading, different cells in the beam will have different values for $P(S)$ depending on how many correct readings have accumulated at each cell. This is unavoidable and is in itself is not a problem. However, if, for example, cell c_i claims the probability of specularity to be 1, and cell c_j claims $P(S) = 0$, c_i will be updated as if $P(S) = 1$ and c_j will be updated as if $P(S) = 0$. The deficiency lies in the fact that if one cell is very confident of the specularity of a sonar reading, then this should reflect on all the cells in the sonar beam, and not just that cell alone. For this reason, better results can be achieved by taking the maximum $P(S)$ of all cells in the sonar beam and applying it to all cells for that particular reading. This has been implemented in the *K97mod* system discussed in Chapter 4, and the original MURIEL method has been implemented in the *K97* system, also discussed in Chapter 4.

A second weakness of the MURIEL method is that, when estimating the probability of specularity of a sonar reading, all sonar readings are treated as completely independent of all other readings, even readings from other sonar sensors in a single scan. They are not independent however, and the readings from one sonar sensor can be use to validate or invalidate data from other sonar sensors, especially data taken from multiple sonar sensors at the same time. The method called Feature Prediction

was developed during this research to take advantage of that very fact, and is presented in chapter three.

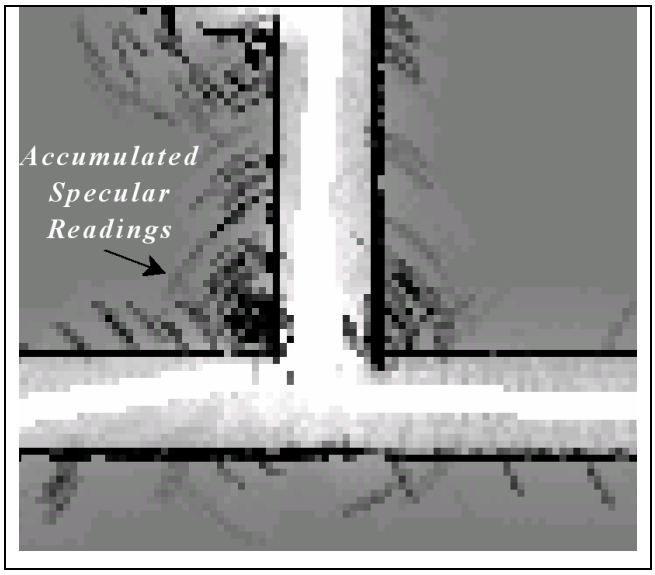


Fig 2.15 Pose buckets allow multiple specular readings of different lengths accumulate in highly specular areas, such as convex corners.

A third problem with the MURIEL method is the use of pose buckets to eliminate redundant, or duplicate, readings. The way in which Konolige advocates using pose buckets means that each cell can be updated from a certain number of angles, and from a certain number of positions. However, when the robot is stationary specular readings can fluctuate widely with readings of many different lengths, giving a result as in Fig 2.15. A possible solution to this is to use pose buckets to allow just one set of readings from a given *robot pose*, as opposed to allowing multiple different readings of different lengths from a single pose to update far too many cells.

A fourth issue with the MURIEL method is the use to which it puts the probability of specularity. When a sonar range reading is given a high probability of specularity, the theory states that the effect of that reading upon the map should be reduced. However the formulae developed for this purpose by Konolige only reduce the effect of the freespace segment of the beam.

$$log\ \lambda_T = log\ \lambda_s + log\left(\lambda_f\left(1 - P(S)\right) + P(S)\right)$$

In the above formula for combining the log of the occupied likelihood value of the map $log\ \lambda_s$ with the log of the freespace likelihood value $log\ \lambda_f$, the probability of specularity $P(S)$ is only combined with the freespace likelihood. The log likelihood of occupancy $log\ \lambda_s$ is incorporated directly, untouched. An improved version of this formula is presented in chapter four, where the *K97mod* map building system builds

on the basic MURIEL method to make the probability of specularity reduce both the freespace and occupied regions update values.

A final issue with the MURIEL method is its large storage requirements. For large complex maps, pose buckets can become extremely demanding of memory resources, as they generally require around 180 times as much storage as a single map. Some careful programming using quad trees can alleviate this problem for relatively simplistic maps, but in a complex map the pose buckets can require many hundreds of megabytes of storage. There is also the issue that the MURIEL method requires the use of three maps, rather than a single map as in other Bayesian based methods [42]. This is necessary because to calculate the probability of specularity using the dynamic mixture model $P(S)$, the cumulative surface readings must be stored separately from the freespace readings. Each of these must therefore be stored in separate maps, with the final generated map being an amalgamation of the two.

2.9 Alternative Approaches To Map Building

Many different approaches to map building have been developed and tested by researchers, each with their merits and their drawbacks. Theoretical evaluations of some of the more prominent mapping methods are undertaken in this section. The methods are as follows:

- Alberto Elfes 1992 paper *Dynamic Control of Robot Perception Using Multi-Property Inference Grids* [20]. This presented a methodology for the use of grids to store considerably more information about the environment in addition to the occupancy value of a cell. It illustrates some problems with the scalability of occupancy grids.
- James Crowley's 1985 paper *Navigation for an Intelligent Mobile Robot* [15], which used both a local grid representation of the world and a global map made up of line segments. This method of representing the environment using line segments in both a local and global model influenced the design of the Feature Prediction algorithm presented in chapter three.
- Sebastian Thrun's three papers *Learning Maps for Indoor Mobile Robot Navigation* [58], *Exploration and Model Building in Mobile Robot Domains* [57], and *Map Learning and High Speed Navigation in Rhino* [59]. These

papers introduced a method of automatically learning a sonar model using neural nets, and for detecting noisy sonar readings using what he called a confidence network. The confidence network is an method for assigning confidence measures to sonar readings, using a trained neural net rather than the Feature Prediction algorithm.

While this thesis is concerned for the most part with grid based maps, some discussion of topological maps is in order, in particular how they can be integrated with grid maps for path planning. This is dealt with in section 2.10.

The chapter finishes with a discussion of blurring maps, and its merits to map building.

2.9.1 Alberto Elfes – Dynamic Control of Robot Perception Using Multi-Property Inference Grids

2.9.1.1 Inference Grids – Taking Occupancy Grids to the Next Level

Elfes followed up his development of a framework based around the Occupancy Grid model by extending the capabilities of the Occupancy Grid. While the original model stored just one piece of information, the probability of occupancy of a set of cells, his new model, termed *Inference Grids* [20], could contain may different types of information about a mapped area.

An Inference Grid is a multi-property Markov Random Field defined over a discrete spatial lattice, or, put simply, a grid that can hold many different types of information at the same time. A Markov Random Field is any representation in which each individual piece of information, in this case each cell, contains within it all the information required to make a decision based on it, with no information needed regarding its past values or the values of the cells around it. They are called Inference Grids due to the way that the information stored within a cell can be *inferred* from it's occupancy state. Some properties that may be estimated include:

- *Reachability* – whether it is possible to travel to cell A from any other cell in the map, B.

- *Observability* – whether it is possible to see cell A from any other cell in the map, B.
- *Reflectance* – whether or not a cell represents part of a surface that is reflective, like a mirror.
- *Colour*
- *Traversability* – whether or not it is possible for the robot to cross a given area.

Which properties we choose to calculate depends on the task at hand and the type of sensors being used e.g. sonars, vision, infra-red etc. For example, sonars are unable to detect the colour of an object.

New methods were also developed to use the Inference Grids to guide the perceptual activities of the robot. The basic theory is that the robot should perform the action which provides the most information, depending on what information is required. The first step in this is to compute the *entropy* (lack of information) of a cell:

$$E(C) = -\sum_{s_i} P[s_i(C) \mid M] \log P[s_i(C) \mid M]$$

where $s_i(C)$ is the occupancy value of the cell, and M is the occupancy grid. When the entropy for each cell has been computed, the average entropy for an area, W, is attainable. We can then calculate the information provided by a sensing action α at time step k:

$$\Delta I(\alpha_k) = -(E_k(W) - E_{k-1}(W))$$

which is simply the reduction in entropy of an area that this sensing action causes. By identifying the area we are interested in, called the *Locus of Interest*, it is possible to identify the optimal sensing action to take by maximising the information gain for the 'interesting' cells. This is done by calculating the *utility* of each action α_i, $U(\alpha_i \mid \pi)$, given our current knowledge π.

$$U(\alpha_i \mid \pi) = \int_{\Delta I} U(\Delta I \mid \pi) P(\Delta I \mid \alpha i, \pi) d\Delta I$$

This is the integral of the utility of a piece of information multiplied by the probability of having such a change in information $P(\Delta I \mid \alpha_i, \pi)$ given the action α and the current state of the world π.

The maximal action is then chosen as the action with the most utility, which is the most useful.

$$\hat{\alpha} = \max_{\alpha i} U(\alpha_i \mid \pi)$$

This extension of the occupancy grid is a large step forward from the original idea of Occupancy grids proposed in [46]. This paper shows that the idea of an Occupancy Grid, or it's extended form of Inference Grids, is more than merely being a useful model to assist with path planning and obstacle avoidance to be built using sonar emitters. Once a task can be expressed in terms of an area of a map that we are interested in for the sake of exploring, finding, and/or manipulating, the Inference Grid can be used to not only guide what motions the robot should make, but also the best sensor(s) to use for that purpose. For example, if the task at hand is to follow a wall, the Inference Grid can be used in determining that sonar or laser range finders would be more suitable than vision input.

2.9.1.2 Issues with Inference Grids

Despite the advantages of Inference Grids, there are a number of drawbacks to using them. Firstly, the space requirements, and the complexity of managing and coordinating the data. Whereas Occupancy Grids only store three maps of an environment, Inference Grids store many different representations of the environment. Given the memory capacity of today's computers, this is not as much of a problem as it was when the paper was published, but the real problem becomes apparent when attempting to keep all the different representations of the environment current with each other.

Because one representation of the world is dependent on others, for example the *reachability* of a region must be updated whenever its occupancy is updated, the task of ensuring that the many different models agree with each other quickly becomes intractable. Whereas Elfes claims that there are many more uses to which Inference Grids can be put, as each extra layer is added to the system, the computational resources grows exponentially.

A second problem comes from the fact that sensors can be very noisy, causing errors in the model. Because one model of the environment depends on others, an error to one model can, in the worst case scenario, be propagated to all other models, reducing their usefulness. No attempt to address this problem is proposed in the paper.

Finally, no real experimental results are given to prove the validity of the approach, and no quantitive or comparative analysis is performed on maps generated using Inference Grids. Without this it is difficult to agree with the author that the effort and complexity of Inference Grids is either feasible or necessary in order to build accurate maps of the robot's environment.

2.9.2 James Crowley – Navigation for an Intelligent Mobile Robot

Around the same time that Moravec and Elfes released their High Resolution Maps [46] paper, James Crowley was proposing a different method of representing the state of the world, or rather, two different methods. His system kept two different types of models of the environment, a Cartesian based model akin to occupancy grids called the *sensor model*, and two line-segment based models called the *global model* and the *composite local model*.

The sensor model is a Cartesian abstraction of the range data, where some two-dimensional coordinates are marked as occupied, and others as empty. It is not a global system, and has a horizon only as distant as the furthest possible sonar range reading. Thus it represents only the robots local environment, what the robot can see at that moment in time. The information contained in this model is converted into a set of line segments, or line equations, using a recursive line fitting algorithm, and then added to the composite local model.

The recursive line fitting algorithm is similar to many others that have been used in the past to identify line segments in images. First points are grouped together where they have the same or similar value and where the distance between them does not fall outside a given tolerance. This tolerance is based on the maximum distance measurable by the sensors and the detail of the map. Next a line is drawn between the two endpoints (the two farthest away from each other) in the collection of points, using the line equation:

$$Ax + By + C = 0$$

A useful property of the line equation is that if we normalise A and B so that the sum of their squares is one and we then test the new equation with any point (x,y) we can calculate the perpendicular distance from that point to the original line. This method is applied to each of the points in the collection, and the point with the greatest perpendicular distance from the original line is identified. If this distance is less than a given tolerance then the original line is accepted. If it is greater than the tolerance however, then the original line is split in to two with (x,y) being the joining point of the two new lines, as in Fig 2.16. This process is recursively applied until the perpendicular distance of all points to their nearest line is within the tolerance.

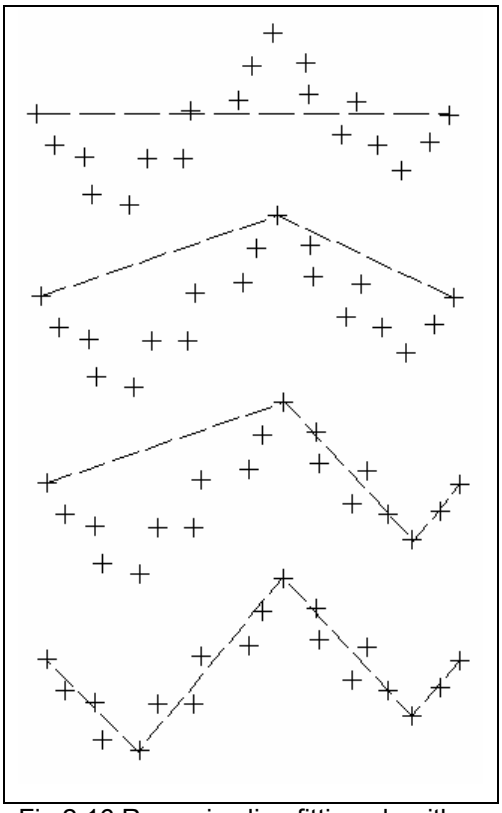


Fig 2.16 Recursive line fitting algorithm used in Crowley's work.

The composite local model is the heart of Crowley's system. As can be seen in Fig 2.17, all sensor information is filtered through it before being used to plan paths, update the global model, generate a network of recognisable places. It is generated from the local information taken from the various sensor models. We say *various* because, as the word composite implies, it is not limited to just a sonar model, but can incorporate any number of different models, such as touch sensors and infrared. Line segments from the sensor models are matched against line segments already in the composite model to find which segments they most closely correspond to. This is used to update errors in the robots position, and also to reinforce the objects that were previously detected. If a segment is added to the composite model, but is not reinforced for a set number of cycles, it is removed from the model as it is assumed to be a dynamic object, such as a person. Note that this is not a probability-based model, it more of a heuristic derived from experimentation.

The global model is a pre-learned set of line segments, similar to the composite model, and is generated in a special learning mode. It is used primarily for global path planning and for developing what Crowley calls a *network of places*, which is a set of open spaces connected by *legal highways*, or valid straight line paths. The global model is also used to update the composite local model, providing it with valid line segments to match its detected segments against.

One of the problems with the approach taken by Crowley is that much of the system he developed was based on experimentation, trial and error, and what "seemed to work". This means that while it may work on his robot and in their test environment, it would probably not port well to another robot and environment without going through the whole process of recalibration again, a non-trivial process. Also, since it is not based on a probabilistic architecture, i.e. the line segments do not have a probability associated with them to be raised or lowered, the update algorithms and segment removal strategies are not very mathematically robust.

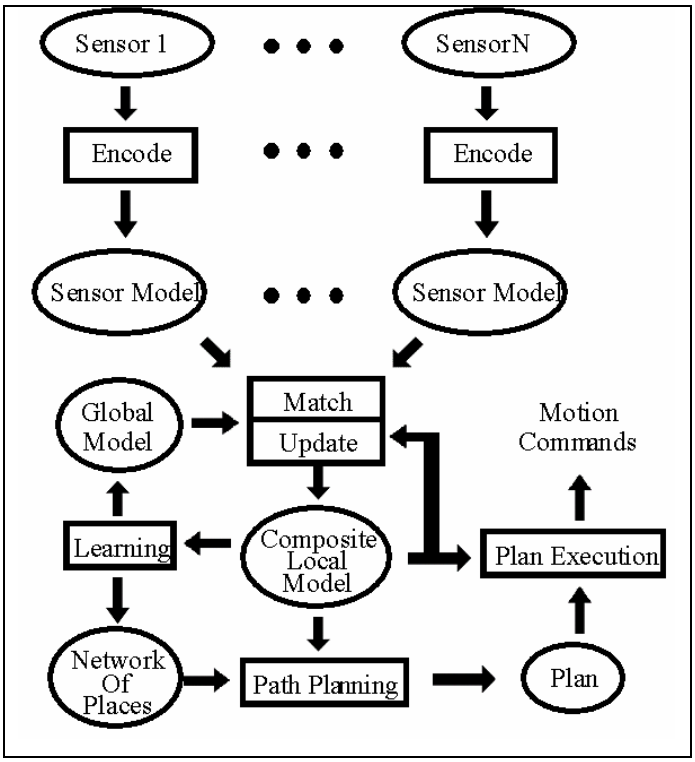


Fig 2.17 Crowley's framework for intelligent navigation.

A third problem with representing the world as a set of line equations is that, while it is possible in a two dimensional world, where a line equation is simple, in a three dimensional world the shape would be far more difficult to construct. While this is

possible, Crowley's whole update and line decay method would have to be completely rethought. As a result of these problems, Crowley's method does not scale well to three dimensions, unlike the Occupancy Grid method where layer upon layer can simply be stacked on each other to create a three dimensional model.

One advantage to representing the world as a set of line segments is that the orientation of the object is taken into account, and individual points are not assumed to be independent from each other. This is useful when doing Markov localisation. When performing Markov localisation, one of the steps is to estimate the correct sonar range reading from a particular position and orientation, or *pose*, in order to compare it with the actual sonar reading the robot has received. One simple, but naïve, method of doing this is to find an occupied grid inside the sonar cone which is closest to the emitter and assume that this obstacle would cause an echo to return to the sonar emitter. The problem with this is that it ignores an inherent property of sonars, that the sonar sensor must be near to perpendicular to the obstacle in order to detect it. If the angle from the perpendicular goes past a threshold, usually around 20^o – 30^o depending on the emitter, the sonar will not correctly detect the obstacle. This weakness must obviously be modelled as closely as possible if the localisation is to be in any way accurate. If the orientation of the obstacle is known, better decisions can be made as to whether the part of it in question, an occupied grid cell, would cause the sonar to reflect back to its source and give a range reading, or whether it would reflect in another direction, giving incorrect data. Crowley gives a simple recursive algorithm for fitting line segments to Cartesian points, but any one of a number of such algorithms are suitable for this purpose.

Representing the world as a set of line segments is also useful in understanding the behaviour of the sonars. The Feature Prediction algorithms for filtering out noisy sonar readings presented in chapter 3 make use this method of representing the world to decide when a sonar is giving correct information or not.

2.9.3 Sebastian Thrun - Learning Maps for Indoor Mobile Robot Navigation, Exploration and Model Building in Mobile Robot Domains, and Map Learning and High Speed Navigation in Rhino

2.9.3.1 A Probabilistic Model for Map Updating

An alternate method of updating a metric map is proposed by Sebastian Thrun [58] using formulas derived from Bayes update rule.

$$P(A \mid B) = P(B \mid A)\frac{P(A)}{P(B)}$$

This paper described a method of building a grid-based map by integrating multiple sensor readings over time using two neural nets. A neural net was trained, using backpropagation, to interpret the readings from the sonars and map them to occupancy values for grid cells. The network outputted a 1 if the cell was occupied and a 0 if it was not. For seemingly erroneous readings, an output of ≈0.5 is generated to indicate lack of certainty. This is then combined with the prior probability of occupancy using Bayes' update rule to give a new value. Using neural nets, as opposed to handcrafted methods, has two main advantages. The first is that, if the environment changes drastically, the network can be quickly retrained. The second is that multiple sensor readings are interpreted at the same time rather than treating each individual sonar reading as independent of the others, which can be useful in identifying erroneous readings. The Feature Prediction algorithm presented in chapter 3 does something similar to this.

For any single reading, we need to find $Prob(occ_{x,y}|s^{(t)})$, which is the probability that the cell (x,y) is occupied given the sensor reading s at time t. For a single snapshot of sonar range reading, Thrun trained a neural network through backpropagation to interpret these to give that very value, which is the range [0..1], with a value of 0.5 representing a lack of knowledge about the cell. This is in contrast to Moravec and Elfes grid, which contains values in the range [-1..1]. However, Elfes later developed a Bayesian model, and used a range of [0..1], since this has certain advantages when used with Bayes' formula. Once this value has been calculated it must be integrated with previous readings, for which he used the formula:

$$Prob\left(occ_{x,y}\middle|s^{(1)},\ldots,s^{(T)}\right)=1-\left(1+\frac{Prob\left(occ_{x,y}\right)}{1-Prob\left(occ_{x,y}\right)}\prod_{t=1}^{T}\frac{Prob\left(occ_{x,y}\middle|s^{(t)}\right)}{1-Prob\left(occ_{x,y}\middle|s^{(t)}\right)}\frac{1-Prob\left(occ_{x,y}\right)}{Prob\left(occ_{x,y}\right)}\right)^{-1}$$

where $Prob(occ_{x,y})$ is the prior probability of occupancy, which can be omitted if set equal to 0.5. This formula can be derived [58] from Bayes formula using the conditional probability assumption, which states that $Prob(s^{(t)}|occ_{x,y})$ is independent of $Prob(s^{(t-1)}|occ_{x,y})$ i.e. that the probability of getting a sensor reading at a particular time given the current occupant value of a grid cell is not dependent on any previous sensor readings. Bayes rule states that

$$\frac{Prob\left(occ_{x,y}\middle|s^{(1)},..,s^{(T)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(1)},..,s^{(T)}\right)}=\frac{Prob\left(s^{(T)}\middle|occ_{x,y,}s^{(1)},\ldots,s^{(T)}\right)}{Prob\left(s^{(T)}\middle|\neg occ_{x,y,}s^{(1)},\ldots,s^{(T)}\right)}*\frac{Prob\left(occ_{x,y}\middle|s^{(1)},..,s^{(T-1)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(1)},..,s^{(T-1)}\right)}$$

which can be simplified down to the following formula using the conditional independence assumption.

$$=\frac{Prob\left(s^{(T)}\middle|\,occ_{x,y}\right)}{Prob\left(s^{(T)}\middle|\,\neg occ_{x,y}\right)}*\frac{Prob\left(occ_{x,y}\middle|\,s^{(1)},..,s^{(T-1)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(1)},..,s^{(T-1)}\right)}$$

If we apply Bayes rule to the first term, this leaves us with

$$=\frac{Prob\left(occ_{x,y}\middle|s^{(T)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(T)}\right)}*\frac{Prob\left(\neg occ_{x,y}\right)}{Prob\left(occ_{x,y}\right)}*\frac{Prob\left(occ_{x,y}\middle|s^{(1)},..,s^{(T-1)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(1)},..,s^{(T-1)}\right)}$$

and induction over T gives

$$=\frac{Prob\left(occ_{x,y}\right)}{1\text{-}\,Prob\left(occ_{x,y}\right)}\prod_{t=1}^{T}\frac{Prob\left(occ_{x,y}\middle|s^{(t)}\right)}{1-Prob\left(occ_{x,y}\middle|s^{(t)}\right)}\frac{1-Prob\left(occ_{x,y}\right)}{Prob\left(occ_{x,y}\right)}$$

Finally, the update equation given originally is derived by solving the latter equation for $prob\left(occ_{x,y}\middle|s^{(1)},\ldots,s^{(T)}\right)$ using the fact that

$$prob\left(\neg occ_{x,y}\middle|s^{(1)},\ldots,s^{(T)}\right)=1-prob\left(occ_{x,y}\middle|s^{(1)},\ldots,s^{(T)}\right)$$

When a reading is received that indicates that a cell that is already believed to be occupied is occupied, then the occupancy value in increased. When the reading claims that an unoccupied cell is unoccupied, the value is strengthened, bringing it closer to 0. However, when a sonar reading disagrees with our current belief, then that belief is weakened, bringing the value closer to 0.5.

2.9.3.2 Using Neural Nets to Interpret Sensor Readings

The neural net mentioned above was a further development of an earlier system [57] designed by Thrun called COLUMBUS. This system used two artificial neural networks to generalise real-world experiences. One neural network interprets the sensors, and the other assesses the confidence of the readings. Both networks encode the characteristics of the sensors and of the typical environments the robot may find itself in, such as a cluttered office environment.

At every time step *i*, each neural net takes as inputs the sensor readings s_i, in this case 24 sonar readings called *sensations*, and the location of the robot x_i. The result is the expected reward r_i, be it negative or positive, received when moving to a given position *x*. This can be seen as being analogous to estimating the probability of occupancy of a cell. The adaptive model, *M*, generalises from a finite set of examples $\{\langle x_i, s_i, r_i \rangle | i = 1 \ldots n\}$, to new positions *x* in the domain:

$$r = M\left(\{\langle x_i, s_i, r_i \rangle | i = 1 \ldots n\}, x\right)$$

where *r* is the expected reward when moving to location *x*. Essentially, when each new set of sonar readings is received, the sensor interpretation network, *R*, generates a local map of the area surrounding the robot. The confidence estimation network, *C*, then estimates the validity of each of the sonar readings, and weakens the effect that sonars of low confidence value have on the local map. This local map is then integrated with the global map.

Both the sensor interpretation network, *R*, and the confidence estimation network, *C* were trained using back-propagation and supervised learning, and used one hidden layer with eight hidden units. The network was trained in a *known* environment, with all obstacles giving a negative reward if the robot should collide with them. This is clearly a limitation of the system, as there is always the danger that the network may not generalise well, or may not operate well in completely unknown environments. However, Thrun claims that this method of training was effective and successfully generalised to real-world office environments.

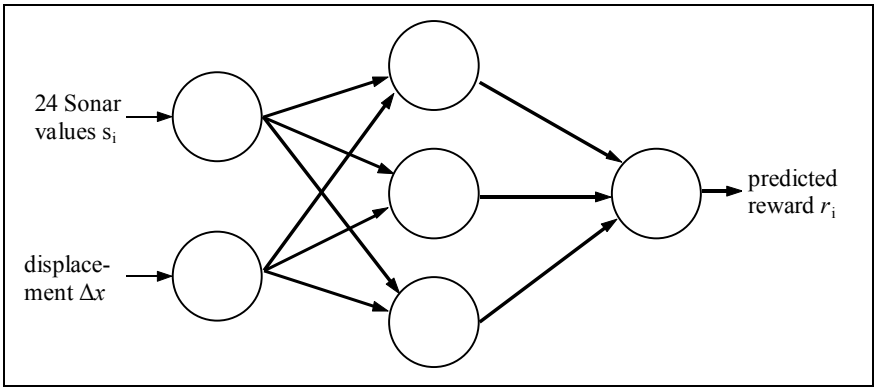


Fig 2.18 (a) Sensor Interpretation Network *R*.

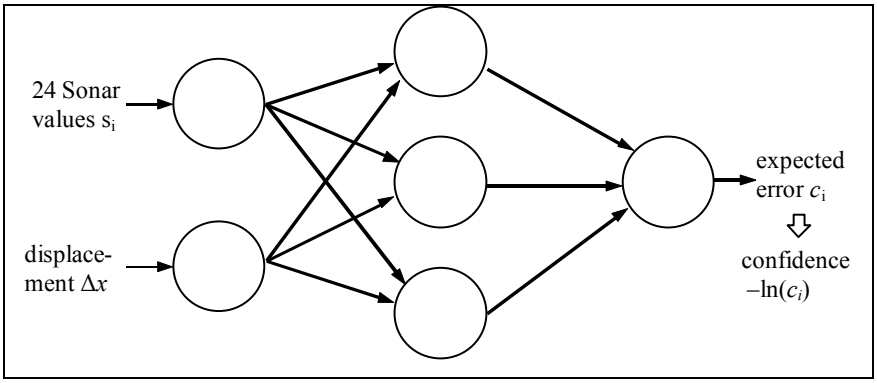


Fig 2.18 (b) Confidence Network *C*.

The *sensor interpretation* network seen in Fig 2.18(a) maps a single position and set of readings to a reward at a given cell, x_i. It takes as inputs the sensor readings and the displacement of x_i from the robot's current position, x, and uses the distance measurements from the sonars to calculate an expected reward, or occupancy, of a given cell, x_i.

The reward function above suggests that the neural net gives a single reward, r, for each position and set of sensor readings. This is not the case however. Rather, it gives n reward estimates, one for each data point (s_i, x_i). This essentially means that the whole map is updated at every time step. If this were all that occurred, it would also be the case that sonars who received no range reading would be propagated to infinity, and that points behind a wall would be updated as empty. However, two techniques are employed to prevent these obvious errors.

The first is the *confidence network*, *C*. This is the second neural network that works in parallel with the *sensor interpretation* network, *R*. It takes as its input the 24 sensor readings as well as the displacement of the cell in question from the robot's current

position. The use of Δx as an input removes the need for the network to model the global coordinate system, which significantly reduces the processing time required. *C* was trained to measure the expected error of the reward function *R*. Once *R* has been trained, an independent test set was used to train *C* to estimate the *expected* error between the reward estimated by *R* for position *x*, and the true state of occupancy of *x*. When the error is high, our confidence in the value given by *R* for the position *x* is low, and vice versa. The readings from *R,* r_i, are then weighted with their respective confidences, defined as $-\ln(c_i)$, and combined using the formula:

$$M(x) = \frac{\sum_i -ln\, c_i(x) * r_i(x)}{c_{M(x)}}$$

where $c_{M(x)} = \sum_i -ln\, c_i(x)$, and is called the *cumulative confidence* at *x*. This acts as a normaliser. As it stands, this formula would update the whole map at each time-step, which would have a negative impact on performance. Rather than do this, Thrun observed that the sonars on his robot had a maximum operating range of 10.5 feet, or about 3.5 metres. Therefore only the points on the map less than or equal to that distance from the robot were updated, which led to a noticeable improvement in efficiency.

A weakness of the confidence net *C* is that it has no memory of any previous confidence values generated for the sonars. For example, if sonar number 1, which returned a long range reading, was given a low confidence value due to sonar 2's short range reading, then it is fairly safe to assume that sonar 1's reading was specular noise. However, in the next time step, if sonar 2 receives a long range reading, due to a gap in a wall or noise for instance, sonar 1 will be given a high confidence value. The example in Step 6 of Chapter 3, Section 3.3 shows one occasion where this happens. If sonar 1 was giving noisy readings in the previous time step, it is likely that it is also giving noisy readings in this time step. Unfortunately the confidence network does not take this into account, and only uses the current sonar readings to calculate confidence values. The Feature Prediction method presented in Chapter 3 solves this problem by estimating the position and orientation of obstacles in the environment, and decaying them over time, rather than discarding them as the confidence network does.

2.9.4 Biologically Inspired Models

There are parallels that can be drawn between biological models and robots' internal models. Much research has gone into mimicking hippocampal cells in rats. These are brain cells that have been observed to fire only when in a certain patch of an environment. This is analogous to recognising a location. It uses three different 'cues' to recognise a location, which are arranged into a network. It has been shown that certain animals can localise themselves in an environment using three recognisable locations, which can be thought of as a form of triangulation. For example, a rat would see a familiar tree, a rock and a sewer, and know where it is and how to get home, or to the nearest food source. However, the issue is not as clear as it may seem, as it has been shown that the neurons fire even in the dark, suggesting that odometry also has a part to play [54], and possibly smell and touch also. This forming of a cognitive map from a number of small pieces eliminates the need for large-scale localisation, as the robot only needs to know its position within the current piece of the map.

There are currently two principal hippocampal theories. The most influential theory, which establishes the hippocampus' involvement in spatial learning and navigation, is the *cognitive map theory* by O'Keefe and Nadel [50]. This theory identifies the hippocampus in the brain as the region providing a neural representation of the rat's location within its environment. An alternative theory is the *auto-associative memory theory* formalised by Marr [39]. This theory holds that the hippocampus is a temporary memory device capable of storing and retrieving sequences of patterns.

2.10 Constructing Topological Maps from Metric Maps

Metric and topological maps each have their strengths and weaknesses. Metric maps are relatively easy to generate, topological maps are not. Metric path planning is slow, topological planning is fast. Sebastian Thrun [59] et al advise combining the two in order to maximise the positive attributes of each. These include the grid-based maps ease of construction and usefulness in self-localisation, and the speed at which it is possible to plan a path using a topological map allied with its compactness.
This involves building a metric map, then converting it to a topological map. Thrun [58] proposes a 5-step method of generating a topological map from a metric map.

Firstly it must be decided which cells are occupied and which not. This is done with *thresholding*, where all cells whose occupancy values are below a certain value are said to be empty, and all those above it are said to be occupied.

Secondly a Voronoi diagram is generated. This is the set of unoccupied points that have at least two equidistant basis points. A basis point is the closest occupied cell to the unoccupied cell, see Fig 2.19. In a simple corridor-like environment, this would lead to a line being drawn down the centre of the corridor. A method for generating Voronoi diagrams is presented in Chapter 5.

The next two steps are to identify critical points and critical lines. A critical point must be part of the Voronoi graph, and there exists a distance $\varepsilon > 0$ for which the distance from all points to their basis points in an ε-neighbourhood of $<x,y>$ is not smaller.

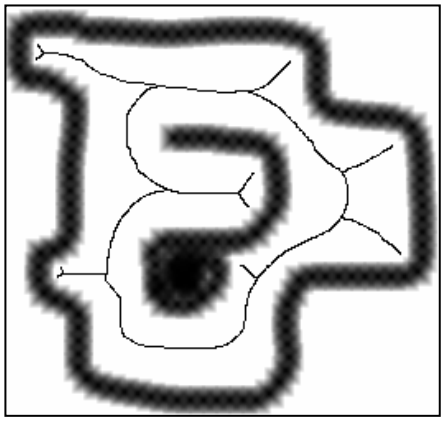


Fig 2.19 Metric Graph with Voronoi Diagram

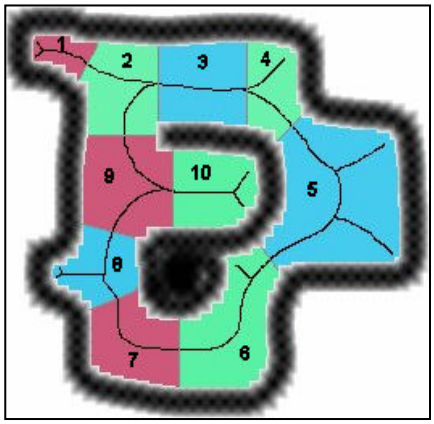


Fig 2.20 Metric graph partitioned by Critical Points and Lines

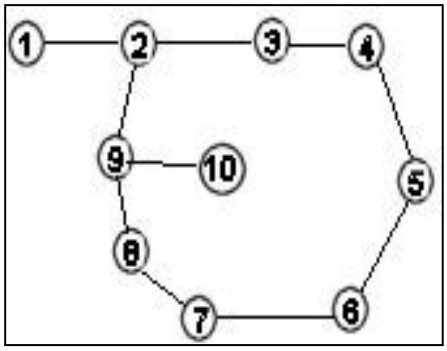


Fig 2.21 Graph generated of map area

Simply put, a critical point is a point in the set of cells in the Voronoi graph that is closer to an occupied point than all other cells in the graph that are within a given distance ε from it. How small or large ε is depends on how detailed the topological map is required to be. The smaller ε is, the more nodes there will be

in the graph and vice versa. Once the critical points are identified the critical lines are generated by simply joining the critical points with their basis points (see Fig 2.21).

The critical lines divide the map into separate areas, each of which is then mapped to a node in a topological graph. Each critical line is represented as an edge in the graph, acting as a transition from one node, or area, to another (see Fig 2.20).

2.11 Constructing Topological Maps Using Potential Fields

The potential field approach to mapping and path planning [30] uses the analogy of obstacles having electric charges, with the strength of the charge decreasing as the distance from the object increases. At any given point, the electric charge, or potential, gives an indication as to the distance to the nearest obstacle, as well as its shape. It can be visualised as a three dimensional map, with obstacles represented as flat-topped plateaus whose slopes descending at an angle from the plateau top to the free space surrounding the object, represented as valleys. Where the potential fields of two obstacles intersect, the maximum potential is chosen, rather than the more obvious method of adding the two values. This prevents small local maxima being created where there are no obstacles.

The lowest points in the map, that is the valley floors, are referred to as Minimum Potential Valleys (MPVs). These represent the Voronoi diagram, and can be followed using a path-planning algorithm. However, since the number of points in a MPV is infinite, it is necessary to limit the computation by obtaining a piecewise linear approximation of it. The MPV of a map is therefore represented as a graph whose nodes are certain points along the valley, and whose edges are straight-line segments connecting these nodes.

This graph can be generated using the following algorithm[30]. The algorithm maintains three queues, the ancestor queue *A*, the father queue *F*, and the son queue *S*. Initially *A* and *S* are empty, and *F* contains the start and goal nodes. The objective is to find the MPV branches exiting the start and goal nodes by recursively drawing spheres around the nodes on the *F* queue, and identifying the valleys within that sphere. The first step is to draw the largest free sphere (the largest sphere that

doesn't intersect an obstacle) around each node in *F*. Every grid reference within the sphere is a possible son of the node at the centre.

All nodes above a certain threshold are deleted as they represent either a collision, or the unacceptable probability of one. The grid point with the lowest potential is then chosen and appended to the *S* queue as a child. An edge is created between this child node and its parent so that an infinite loop does not occur as a result of making the parent a child of its own child. Another largest free sphere is drawn around this child node, and all grid points within it are deleted. Next the process is repeated, with the selection of the point of lowest potential from the remaining grid points, appending it to the *S* queue etc. This process continues until there are no more points in the original parent sphere.

At this point the distance between each node in the child queue, *S*, and each node in *F* that is not its father is computed, and an edge is created between them if the distance is less than the distance from the son node to the nearest obstacle. The entire *F* queue is then moved to the ancestor queue, *A*, the nodes in *S* are moved to *F*, and the whole process begins again. The algorithm will terminate so long as the map area is bounded.

2.12 Blurring

A discrete metric map purports to offer independent information about the occupancy of particular grid cells, for example cell 1 has an occupancy probability of 0.8, while cell 2 has a value of 0.1. Unfortunately, the unreliability of the sensors means that this assumption of independence is false, as one cell can in fact affect the value of another through incorrect range measurements, specular reflection etc. A good example of this is a concave meeting of walls at a fairly acute angle. While the walls near the corner might be detected, the robot will often not detect the corner itself due to the properties of the sonar beam. One way of compensating for this is to assume that if all the cells around a location are occupied, there is a higher probability that the sensors are wrong to mark the cell as unoccupied. This can be achieved by blurring the map.

Blurring a map makes each cell represent not only the occupancy value of that particular space, but also represent the space surrounding it. This leads to the first question: at what distance do cells stop affecting the value of the current cell? In other words, what size *kernel*, or mask, do we want? An example of a 5x5 kernel is given in Fig 2.21.The larger the kernel, the more cells affect the cell at the centre of the kernel. There are two types of blurring commonly used in map building, box blurring and gaussian blurring. These are discussed below.

2.12.1 Box Blurring

Box blurring is a method for averaging the value of a particular (X,Y) grid position with the values of the cells around it. The size of the kernel and weight applied to the map values depends upon the problem it is being applied to.

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Fig 2.22 Box Kernel.

In terms of image processing, a signal, *A*, can be applied to another signal, *O*, to give an output signal, *R*. The map being blurred can be seen as the original signal, *O*, while the applied signal, *A*, can be modelled as a mask or kernel, as in Fig 2.22. The final map generated by applying the mask is analogous to *R*.

The box blurring method for a 5x5 kernel is carried out as follows:

For any grid cell (X,Y) (the shaded region in Fig 2.22), sum all the values of each grid cell in the square with a top left-hand-side of (X-2, Y+2), and bottom right-hand-side of (X+2, Y-2). Divide the total by 25, and assign that value to (X,Y). Repeat this process either for the whole map, or for the chosen region of the map.

$$d_{x,y} = \frac{c}{(2k+1)^2} \sum_{i=-k}^{i \le k} \sum_{j=-k}^{j \le k} s_{x+i,y+i}$$

Here [21], $d_{x,y}$ is the new map value, (2k+1) is the kernel size e.g. if the box is 5x5, k = 2. c is the constant, in this case 1, and s is the source map.

2.12.2 Optimising Box Blurring

Unfortunately, there is a double sum per grid position, which is very processor intensive for a large map. This can be overcome by employing a simple method [21]

that involves looking at the blurring as two separate blurrings – a horizontal blur and a vertical blur. First do one, then the other, with the result being exactly the same as the above method. The trick lies in the fact that when the kernel is moved one place to the right, the sum of all cells except for the rightmost column has already been computed. Taking this into account, the algorithm applies one line of the kernel at a time, moving from left to right. Each time the kernel moves one place to the right, the cell one place to the left of the kernel is subtracted from the total, and the rightmost cell is added to the total. A similar technique is applied to the vertical blurring. This algorithm operates in constant time, $O(N)$, with each cell only being accessed twice, unlike the brute force method mentioned above, which runs in $O(N^2)$ time.

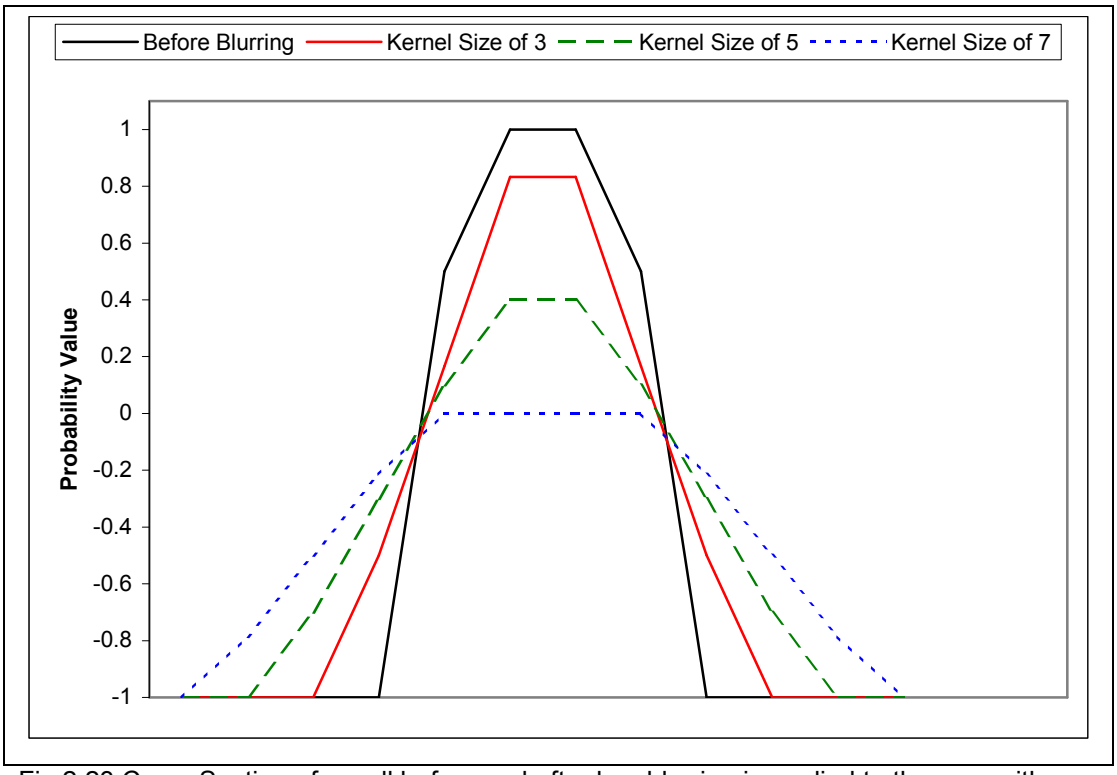


Fig 2.23 Cross Section of a wall before and after box blurring is applied to the map with three different masks, or kernels, size: 3*3, 5*5, and 7*7. Note the lower probability values as the kernel size increases.

When Box Blurring is applied to a very sparse map with few obstacles, there is little change, since most cells have the same value and are therefore not greatly changed. However, when there are small obstacles surrounded by empty areas, or thin walls with empty spaces to either side of them (a very common occurrence) all that is accomplished is the lowering of the occupied certainties, and, with a sufficiently large kernel, obliterating the walls completely. In Fig 2.23, the result of applying box

blurring to a map using various kernel sizes can be seen. As the kernel becomes larger, the confidence in the existence of the wall decreases, while the empty areas near the wall have their values raised.

2.12.3 Gaussian Blurring

The problem with box blurring a map is that the thin walls are surrounded by large expanses of lower-probability grid cells. What is needed is that less priority be given to cells farther away in the kernel from the source cell, and higher priority be given to the cells closer to it. This can be achieved with Gaussian blurring [21].

7	28	42	28	7
28	113	170	113	28
42	170	255	170	42
28	113	170	113	28
7	28	42	28	7

Fig 2.24 Gaussian Kernel.

Gaussian blurring weights the cells, with values dependent on their distance from the centre of the kernel. As can be seen in Fig 2.24, the weight given to each cell decreases the further it is from the centre of the kernel. While this method does give a better result that box blurring, it is more computationally expensive, with a costly double sum and a multiplication, as can be seen in the following formula:

$$d_{x,y} = \frac{1}{(2k+1)^2} \sum_{i=-k}^{i\le k} \sum_{j=-k}^{j\le k} \ker_{i,j} * s_{x+i,y+i}$$

where $d_{x,y}$ is the new cell value, $s_{x+i,y+i}$ is the source cell value, and $ker_{i,j}$ is the weight.

Using the ratios shown in Fig 2.24, the gaussian mask was generated and applied to the map. As with the box blurring function, it can scale from a 3x3 kernel up to any size, but using too large a kernel effectively renders the map unusable. The result of applying masks of varying sizes to a map containing a single wall are as follows:

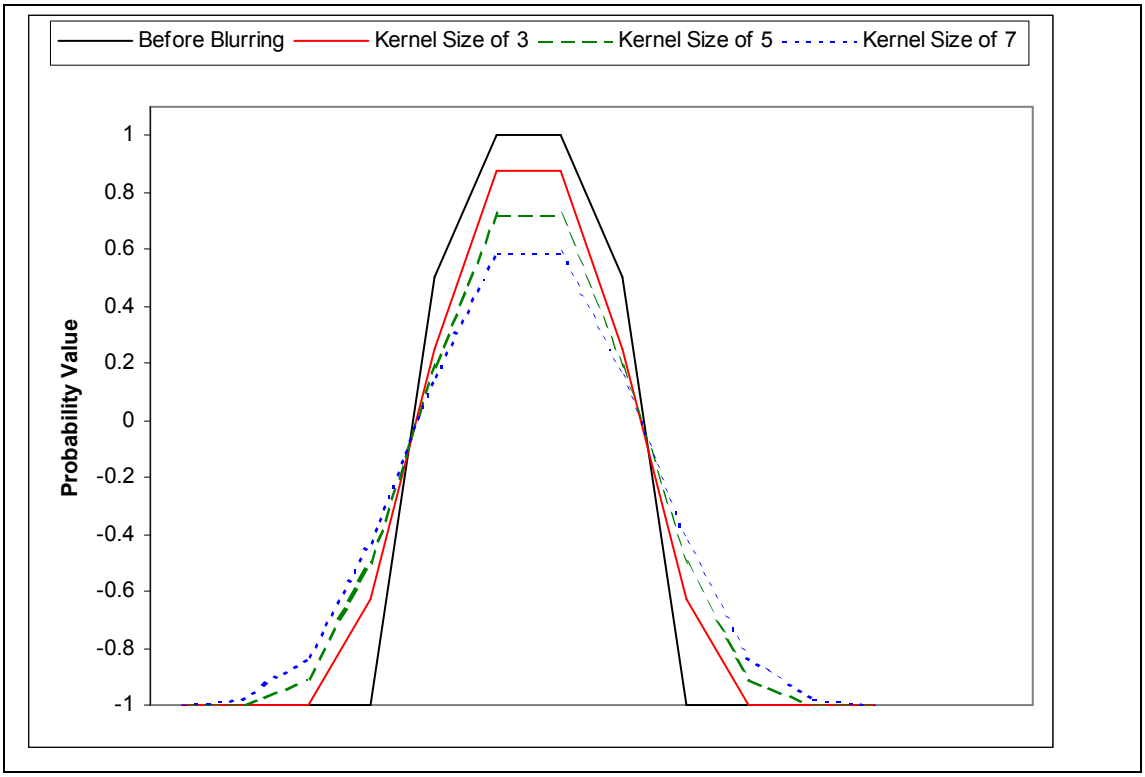


Fig 2.25 Cross Section of a wall before and after gaussian blurring is applied to the map with three different masks, or kernels, of size: 3*3, 5*5, and 7*7. Note the lower probability values as the kernel size increases is not as severe as with box blurring.

As can be seen in Fig 2.25, gaussian blurring scales much more gracefully to larger kernel sizes than does box blurring, with the maximum probability values decreasing only slightly as the kernel size becomes larger. This means that a gaussian mask can be used to blur the map, and thereby smoothing over gaps in walls, without reducing our confidence in the surfaces that we are sure about by too great a measure.

Fig 2.26 shows a map before blurring, after box blurring with a kernel size of 9, and after gaussian blurring with a kernel size of 9. The gaussian blurred map is clearly a closer representation of the map than the box blurred map, while at the same time smoothing over any small imperfections in the map.

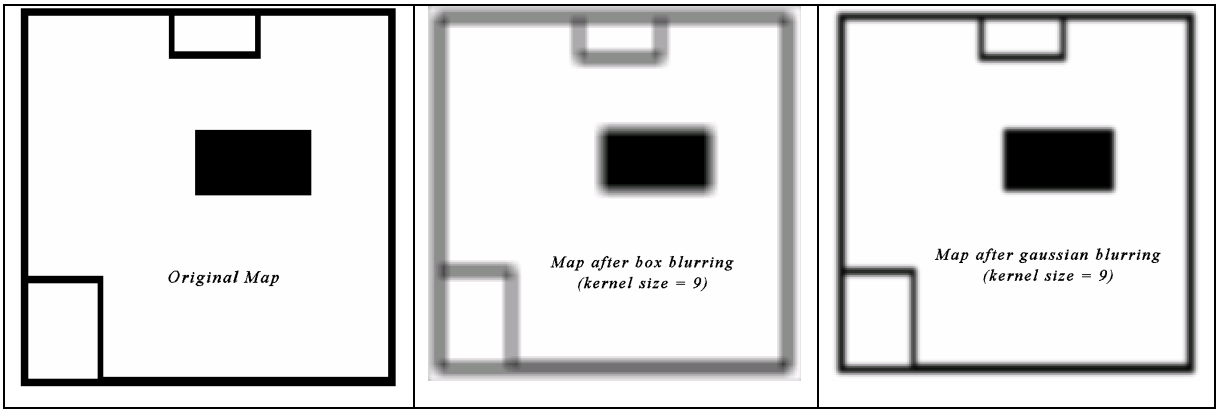


Fig 2.26 A map without any blurring (left), after box blurring with a kernel size of 9 (middle), and after gaussian blurring with a kernel size of 9 (right). Note that when using box blurring the walls have almost disappeared due to the same weight being given to the cells around the cell being blurred. Gaussian blurring fares better due to a higher weight being applied to cells close to the cell being blurred that to cells farther away.

Chapter 3: Feature Prediction for Filtering Noisy Sonar Readings

3.1 Feature Prediction – An Introduction

A method of identifying specular readings based on predicting the state of the world has been developed for this thesis to extend the work of previous researchers in map building using mobile robots. Three models of the environment are maintained in order to build a map, a sonar map *S*, a local map *L*, and a global map *G*. The sonar map *S* contains features estimated from just the current set of sonar readings. These features take the form of straight line segments. The local map *L* maintains a set of features estimated from previous scans, but only within the area immediately surrounding the robot. The global map is a grid based metric map that represents the environment as a set of occupancy cells [33,46].

A paper entitled "Linear Feature Prediction for Confidence Estimation of Sonar Readings in Map Building"[52], detailing the feature prediction algorithm, has been accepted for publication at the Ninth International Symposium on Artificial Life and Robotics (AROB) 2004 conference in Oita Japan.

The sonar and local maps are predictive models, i.e. they estimate the state of the world by using previous sonar readings to predict what the robot *should* sense in the future. These predictions are used to differentiate between those sonar readings that are based on meaningful range measurements and those which are based on noisy reflections. This is expanded upon in section 3.2. By their very nature, predictions can often be inaccurate, especially given the high degree of noise in sonar sensors, and for this reason the features are not incorporated directly into the global map *G*. Rather, they are used to guide traditional map update procedures that create a global map based on information inferred directly from sonar readings using a sonar model, enabling the map building algorithm to compute which sonar readings to use and which readings to weaken or discard completely.

There are seven steps involved in calculating a confidence measure for each sonar based on all past and current readings and using it to update a global map.

1. Create a feature hypothesis for every sonar that detected an obstacle, and place it in the sonar model *S*. These features take the form of straight line segments.
2. Compare all features in *S* with each other in case they represent the same real-world object. If they do, merge the features while at the same time refining the features' position and orientation.
3. Identify any features in *S* that were created from noisy readings, and discard them.
4. Add all remaining features in *S* to the local map *L*. If a feature from *S* is sufficiently similar to a feature in *L*, the two features are merged and the resultant line segment is placed in *L*. If a feature in *S* doesn't match any existing line segment in *L* then it is added to *L* unchanged, as a new feature.
5. Segments in *L* too far from the robot are removed from the model.
6. Generate the confidence values in the sonar readings, and decay any feature that the sonar should detect but doesn't.
7. Use the sonars confidence value to reduce the effect the range reading has on the global map *G*. The lower the confidence value, the less the map should be changed as a result of that reading.

Each of these steps is expanded greatly upon in section 3.3.

3.2 The Need for Feature Prediction

Researchers in the field of map building using sonars have often encountered the problem of noisy readings that can result from specular reflections [33,42,46,58]. Specular reflections occur when the sonar beam strikes a smooth surface, which causes the beam to reflect at an angle related to the angle at which it struck the object (see Fig 3.1 (b) and 3.1 (c)). A smooth surface is one whose bumps are smaller than the wavelength of the sonar beam. Some researchers [33] use the word 'specular' in a slightly different way, to refer only to sonar beams that strike a smooth surface and do not return correctly to the sonar emitter. Here, on the other hand, the term is used in its more general sense to refer to any beam that is transmitted off a surface at the same angle at which it struck that surface. While it is technically possible to receive a

correct range reading from a specular reflection however (Fig 3.1(c)), the majority of specular readings result in noisy readings (Fig 3.1 (b)).

Another type of reflection is diffuse reflection, in which a sonar beam strikes an object whose surface contains irregularities larger than the wavelength of the sonar beam, causing the beam to be scattered in all directions. In this case a correct range reading is almost always received.

In the case of specular reflections however, if the angle of the sensor to the orientation of the object is sufficiently far from perpendicular, then the reflected beam does not come directly back to the sonar, as can be seen in Fig 3.1 (b). Non-noisy readings can be received from specular reflections also, as long as the angle of incidence is close to 90^{o}, as in Fig 3.1 (c). The beam will also return to the sonar emitter sometimes from a specular reflection at an acute angle, but this reading is based on the beam reflecting off multiple objects and therefore does not represent a correct range reading to the nearest obstacle. It is universally agreed that such a reading contains no information and should not be integrated into the map. The problem then becomes that of identifying which of the three types of reflective behaviour displayed in Fig 3.1 the sonar is exhibiting. If the noisy readings caused by specular reflection can be identified and discarded the

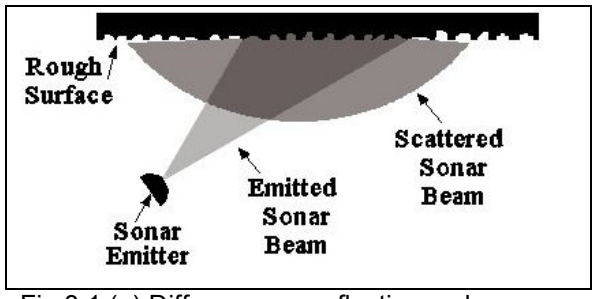


Fig 3.1 (a) Diffuse sonar reflection - when a sonar beam hits a rough surface it is scattered in all directions and returns directly to the emitter.

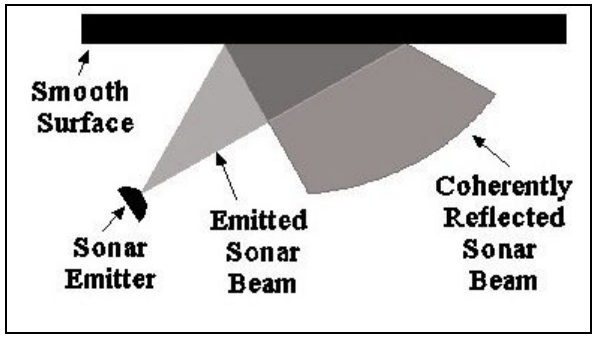


Fig 3.1 (b) Specular sonar reflection giving a noisy reading - when a sonar beam strikes a smooth surface at an angle far from 90^{o} it does not return directly to the emitter.

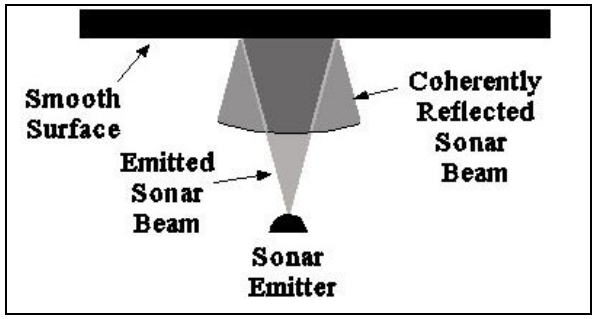


Fig 3.1 (c) Specular reflection giving a correct reading - when a sonar beam strikes a smooth surface at close to 90^{o} the beam will return to the emitter.

resulting global grid map will be far more accurate.

3.2.1 Previous Attempts at Specularity Estimation

Konolige [33] recognised this problem and developed his MURIEL method in an attempt to solve it. As discussed in chapter 2, Konolige uses what he called a *Dynamic Mixture Model* to sum all independent sonar readings that report an obstacle at a cell (surface readings), and once the sum reaches some value, usually around 2 or 3, all readings that claim the cell to be unoccupied are assumed to be specular. This method favours occupied cells over unoccupied cells since if two readings disagree on the occupancy of a cell, the one that claims it to be occupied is believed. This follows from Drumheller's [19] *sonar penetration condition* which states: the freespace hypothesis of a sonar reading should not impinge on a high-confidence surface.

The independence of sonar readings is assured by using pose buckets, which ensure that a cell can only be updated once as being occupied, and once as being unoccupied from any given position (see section 2.8.3). The basic idea behind this is similar to the *cancelling* step proposed by Moravec and Elfes [46], except that it favours the occupied cells instead of the unoccupied cells. The cancelling step in [46] states that whenever a reading claims a cell to be occupied, multiply its probability of occupancy, $P_O(X,Y)$, by one minus its probability of being empty, $P_E(X,Y)$.

$$P_O(X,Y) = P_O(X,Y) * (1 - P_E(X,Y))$$

This favours the unoccupied cells since once $P_E(X,Y)$ reaches the value of 1 all new sonar readings claiming the cell to be occupied are multiplied by 1 – 1 = 0 giving a probability of occupancy of zero.

$$P_O(X,Y) = P_O(X,Y) * (1 - P_E(X,Y)) = P_O(X,Y) * (1-1) = 0$$

Konolige's dynamic model mixture essentially reversed this and instead cancelled out the empty readings based on the probability of occupancy at the cell, although his mathematics differed considerably, using probability of specularity $P(S)$. To calculate $P(S)$ for each cell in the freespace part of the beam, he took the occupancy likelihood value of that cell and, given a value at which the cell is said to be absolutely certain of it being occupied, used a linear interpolation function to map this value to $P(S)$. For example, if the occupancy likelihood value of the cell is 0.8, and an upper limit of 3 is set for surface readings above which the cell is said to be definitely occupied, then

$P(S) = 0.8 / 3 = 0.266666$. The freespace likelihood of each of the cells in the sonar beam was then multiplied by $(1 - P(S))$, meaning that the higher the value of $P(S)$, the weaker the update applied to the global map.

Although the MURIEL method's estimation of the readings' specularity undoubtedly creates better maps than it would without this estimation, its greatest drawback is that it needs sufficient surface readings at or around that cell before it can filter out noisy readings. When modelling an obstacle in the global map with correct and/or incorrect range data, there are two possible scenarios:

1. A correct reading is received first, and an incorrect reading second, if at all.
2. An incorrect reading is received first, and a correct reading second, if at all.

The MURIEL method's dynamic mixture model can compensate for incorrect specular readings in the first case, but it is far less effective in the second case.

If specular readings are received before any correct readings, as is often the case for example when travelling straight down a corridor, then the incorrect readings will be incorporated into the map. Whether or not later correct readings rectify the errors caused by the earlier specular readings depends on the pose of the robot and how many correct readings are received. Another problem is that some parts of an environment, such as concave corners, rarely if ever return accurate readings due to multiple reflections and therefore all specular readings that claim a corner to be unoccupied have no correct readings to disagree with them.

This chicken and egg problem of requiring surface readings at a cell before incorrect readings can be discarded means that it is necessary to make predictions about the state of a cell *before* we receive any correct readings at that cell, or even without ever getting any correct readings at all. These predictions can be made based on sonar readings received that don't directly affect the cell in question, and also on our knowledge of how a sonar beam behaves in a noisy environment. This is possible because the environment contains structural regularities and symmetries such as walls that can be approximated using straight line segments.

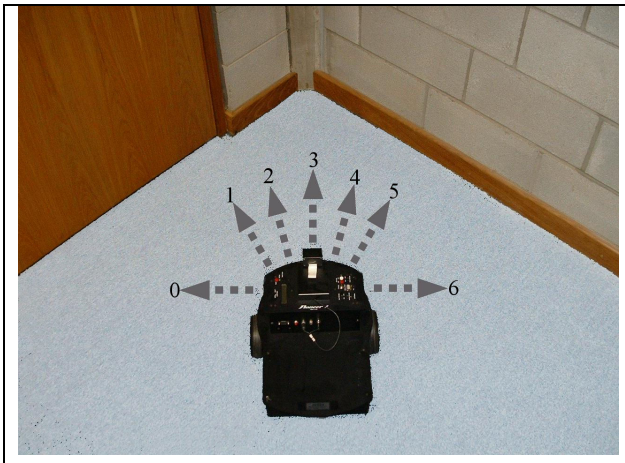


Fig 3.2 (a) Robot facing into a corner often receives incorrect readings from sonars not sufficiently close to 90^{o} from the walls.

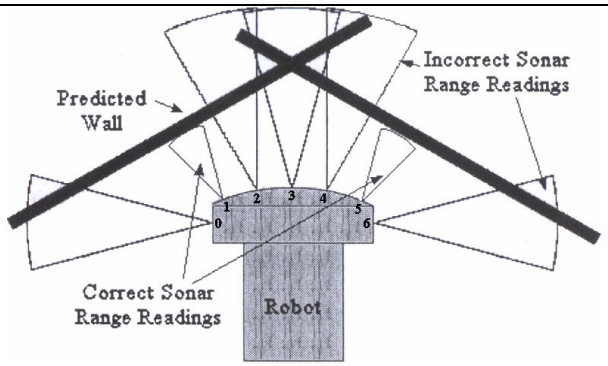


Fig 3.2 (b) Robots sonar readings facing into the corner in 3.2 (a) and the resulting predicted features. Note that only sonars 1 and 5 return a range reading, all others return no reading at all. This is admittedly a trivialised example, purely for displaying the concept.

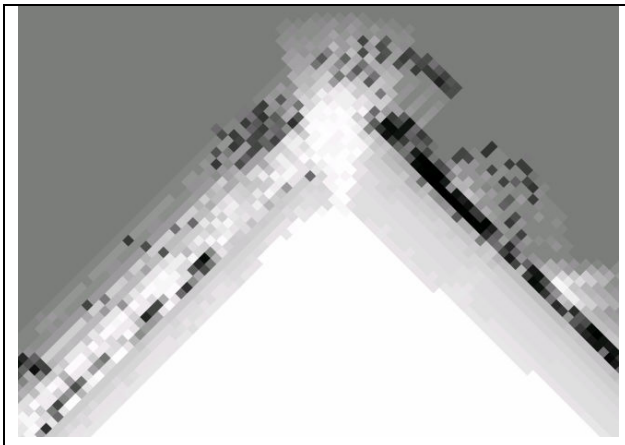


Fig 3.2 (c) Interpretation of sonar readings from Fig 3.2 (a) without feature prediction using a Bayesian update procedure and the sonar model from Moravec and Elfes 1985 paper and pose buckets. Black areas represent occupied cells, white unoccupied, and varying levels of uncertainty are represented by lighter and darker levels of grey.

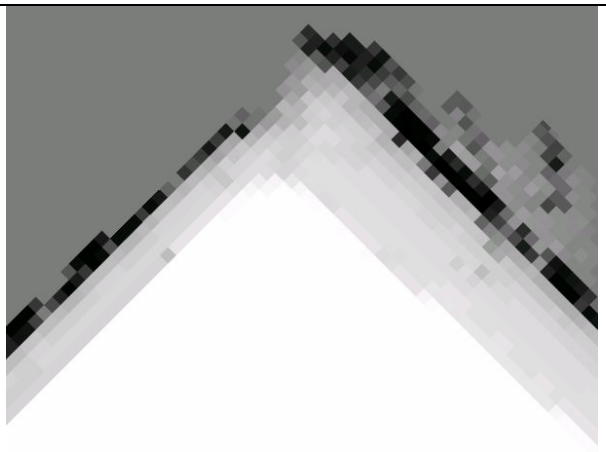


Fig 3.2 (d) Interpretation of sonar readings from Fig 3.2 (a) with feature prediction, based on the algorithm in section 1. It uses a Bayesian update procedure, the gaussian sonar model from Moravec and Elfes 1985 paper and pose buckets. The colour key is as in Fig 3.2 (c).

3.2.2 Example of the Feature Prediction in Operation

As a brief example of the feature prediction algorithm in operation, take a robot facing into a corner as in Fig 3.2 (a). The central sonars receive incorrect readings due to the

beams reflecting off multiple surfaces before coming back to the sensor, if they come back at all. Since all their readings are longer than the horizon of 2.5 metres, they are judged to have not detected any obstacles. Sonars 1 and 5 receive correct range readings as they are close to perpendicular to the walls. Sonars 0 and 6 receive incorrect readings due to their angle to the walls, and once again are judged to not have detected any obstacles due to their range reading being larger than the horizon. Experiments into the operation of sonars show that when a sonar strikes a smooth surface it will not return a correct reading unless the orientation of the obstacle is close to the range [90 – sonar beam width, 90 + sonar beam width] to the angle the sonar is facing. The first step therefore is, as in Fig 3.2 (b) to assume that there is a feature at the centre of the only two sonars to return a valid range reading, sonars 1 and 5, at an angle of (max Possible Angle + min Possible Angle)/2, which in this case happens to be 90^{o} from the angle of the sonar (Fig 3.2 (b)). The algorithm for extracting features from sonar scans is presented in section 3.3.

The next step is to model all of the sonar readings and give them all a confidence value of 0.5 since as yet there is no reason to doubt their validity – the reason for choosing this value is explained in section 3.3. If any of the sonar beams cross where a feature is predicted to be without detecting it, the confidence value for that sonar is decreased. The mapping system can then take these confidence values and use them to reduce the degree to which each individual sonar reading affects the map and/or establish a cut-off point below which the reading is simply discarded.

As can be seen from Fig 3.2 (c), when all sonar readings are treated equally, i.e. with no estimation as to their specularity being made either by feature prediction or using past information as in the MURIEL method, then incorrect specular readings ‘wash out’ much of the wall and fail to detect the corner at all. In Fig 3.2 (d) the feature prediction algorithm identifies specular readings and discards them if they fall beneath a threshold, which the majority of them do. The left wall of the corner in Fig 3.2 (d) is not marked as occupied since no correct readings were received from those cells due to the position of the robot and the specular reflections caused by corners as discussed previously. Importantly however, the readings that claimed the corner to be unoccupied, as in Fig 3.2(c), have been removed, leaving the corner a medium grey, or a value of 0.5.

3.3 Algorithm for Generating Features and Calculating Sonar Confidence Measures

In the following algorithm there are two predictive models of the environment, the sonar view *S* and the local model *L*. The sonar view consists of features, or line segments, that can be estimated from the most recent set of sonar readings. The local model contains a history of all the line segments estimated from past sonar scans. The current sonar range readings are used to predict features in the environment, which take the form of line segments which then constitute the sonar model *S*. The segments in *S* are then analysed to identify any lines that may have been caused by incorrect sonar readings, which are then removed. The features in *S* are added to the feature list in the local model, *L*, either as new features that have not been detected before, or by being merged with existing line segments that they are sufficiently close to in orientation and position. Finally, the current set of sonar readings are compared to the line segments in *L*. Any sonar that crosses any of the line segments without detecting it has it's confidence value changed. This confidence value is then used to alter the degree to which that sonar reading affects the global map being built, *G*.

The algorithm for calculating confidence measures in the accuracy of the sonar readings is as follows.

STEP	ALGORITHM
1	**Create a line segment for every sonar that claims to have detected an obstacle and place it in the sonar view *S***
	FOR EACH sonar range reading < maximum range Create line segment feature s_i at centre of beam, at 90^o to direction of beam, and place in *S*
2	**Compare each line segment in *S*, s_i, with all the other line segments in *S*, s_j, to determine if they represent the same feature in the environment.**
	FOR EACH s_i in *S* FOR EACH s_j in *S* NOT EQUAL TO s_i IF similar(s_i , s_j) THEN Create new feature s_k between s_i and s_j and place it in *S* Remove s_i and s_j from *S*

3	**Remove any feature in *S* caused by incorrect readings. Confidence values derived from the sonar view are denominated by c_x^s, where the *s* signifies that this confidence value was derived from the sonar view, and *x* is the sonar sensor that it applies to. An ^{L} superscript signifies that the confidence value was derived from the local view *L*.**
	FOR EACH son_i in *SONARS* Reset confidence of son_i, c_i^s to 0.5

	FOR EACH son_j in *SONARS*
	FOR EACH s_i in *S*
	IF inconsistent(s_i , son_j) THEN
	IF son_j should detect s_i THEN
	Reduce confidence c_j^s of son_j varying the angle
	ELSE IF son_j should not detect s_i
	Reduce c_j^s of son_j varying the distance
	FOR EACH l_i in *L*
	IF inconsistent(l_i , son_j) THEN
	IF son_j should detect l_i THEN
	Reduce c_j^L of son_j varying the angle
	ELSE IF son_j should not detect l_i
	Reduce c_j^L of son_j varying the distance

	FOR EACH s_i in *S* IF s_i detected by only one sonar, son_j AND $c_j^s < T_s$ Remove s_i from *S*
	FOR EACH son_i in *SONARS* Reset c_i^s to 0.5
4	**Compare all remaining features in *S* with all lines in *L*, refining the hypothesis of the angle and position of the features in *L***

	FOR EACH s_i in S FOR EACH l_j in L IF similar(s_i , l_j) THEN Create new feature between s_i and l_j and place it in L Remove s_i from S and remove l_j from L
	FOR EACH s_i in S Place s_i in L as a new segment Remove s_i from S
5	**Features in *L* too far from the robot are removed**
	FOR EACH l_i in L IF dist(current_robot_position, last_robot_position_detected(l_i)) > T_d THEN Remove l_i from L
6	**Generate final sonar confidence values. For any segment in *L* that should have been detected but was not, split in two and shorten it. Confidence values derived from the local model *L* are denoted with a superscript *L*, with the number sonar they apply to represented as a subscript, as in c_x^L.**

	FOR EACH son_i in *SONARS*
	FOR EACH l_j in L
	IF inconsistent(s_i , son_j) THEN
	IF son_i should sense l_j if it is there THEN
	Reduce confidence in son_i, c_i^L , varying the angle
	Divide l_j at point of intersection with son_j, creating l_k and l_l
	Remove l_j from L and insert l_k and l_l into L
	Reduce length of l_k and l_l
	IF length(l_k) < T_L
	Remove l_k from L
	IF length(l_l) < T_L
	Remove l_l from L
	ELSE IF son_i should not sense l_j if it is there THEN
	Reduce c_i^L varying the distance

7	**Use sonar confidence values to alter how the sonar information is integrated into the global map. The implementation of this step is specific to the method being used to generate the global map, and is described in detail later.**

Each step is described in detail below.

3.3.1 Step 1: Create a feature for every sonar that detected an obstacle and place it in the sonar model *S*

Set a maximum range that will be accepted for a sonar reading, beyond which the reading will be treated as if it didn't return a reading. The larger this range the further the robot will be able to 'see' but the less reliable the readings will be due to the increasing possibility of multiple specular reflections and the weakening of the wave's power. A reasonable trade-off between myopia and accuracy is 2.5 metres for Polaroid transducers.

For each sonar that returned a reading of less than the maximum range assume the presence of a feature at the *centre* of the outer arc of the beam. At this stage all that is known about the orientation of the feature is that for the beam to detect it, the difference in angle between the object and the central axis of the beam must be in the range [90 – sonar beam width, 90 + sonar beam width] from the angle of the beam. For example, if the sonar sensor is oriented at 60°, a feature perpendicular to it is oriented at 150°. Given a sonar beam aperture of 25,

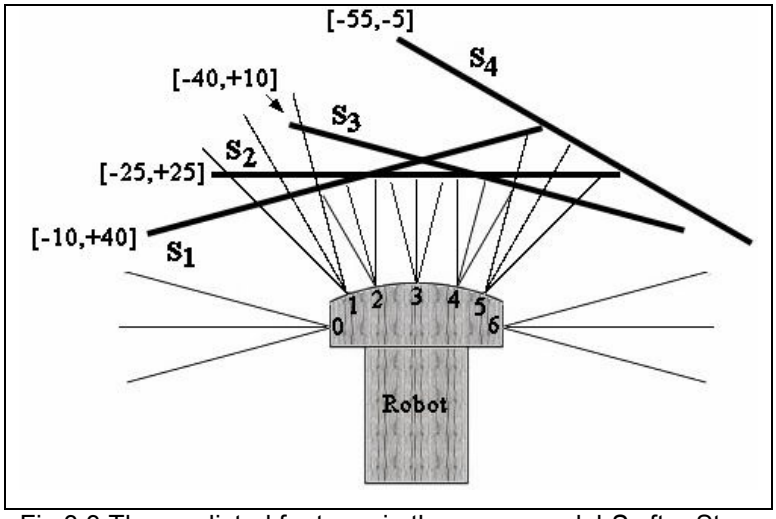


Fig 3.3 The predicted features in the sonar model *S* after Step 1. The robot is facing straight into a wall (as opposed to a corner as in previous examples) with sonars 2, 3, 4 and 5 returning a reading that is less than the maximum range. Sonars 0, 1 and 6 return no reading, and sonar 5's reading is incorrect. The min/max possible angles the line segments can be in is in the [] brackets beside each line.

the minimum possible angle the feature could be oriented at is $150^o - 25^o = 125^o$, and the maximum orientation it could have is 175^o.

The min/max values are stored for use later when comparing the line segment with others to see if the could represent the same object. The aperture of a sonar beam depends on the particular emitter, but a typical value is 25 degrees [27], which holds for the Pioneer 1 robots used in the experiments reported in Chapter 6. Later, when the exact orientation of the feature is required to make predictions regarding a sonar reading's confidence, the value taken for the feature's orientation is the value half way between it's minimum and maximum possible values. The advantage of using min/max values for the orientation of the line segment is that when it is later merged with other line segments its range of possible orientations will be continually refined, and finally a very strong hypothesis of the lines' orientation is calculated.

As for the length of the feature, only the robot's local area is being estimated, so it is best if the feature does not extend far outside this area. Therefore the length of the line segment is set to twice the maximum range of the sonars with the centre being at the centre of the edge of the sonar beam that detected it. This gives a local map of around 5m * 5m.

As an example, in Fig 3.3 with the robot facing a wall, four sonars, 2 through 5, return a reading that is less than the maximum range. A line segment is placed at the end of the central line of each sonar at 90^o to it.

3.3.2 Step 2: Compare each line segment $s_i \in S$, with all the other line segments in S, s_j, to see if they could represent the same feature in the environment.

This method of feature prediction makes the assumption that the robotic platform contains sonar emitters arranged in a ring in sequential order. For example, in Fig 3.4 the sonars on the robot are arranged from sonar 0 to sonar 6, left to right. Three conditions must be met for any two features in the most recent snapshot to match:

1. The two features must have been detected by adjacent sonars. As an illustration, take for example the robot configuration in Fig 3.4. If sonar 1 detects an obstacle, sonar 2 does not and sonar 3 does, sonars 1 and 3 cannot

be detecting the same obstacle, otherwise the sonar between them would have detected it too.

2. The [MIN, MAX] range of possible angles for both segments must overlap. For example, two segments with the possible angles of [0 , 50] and [70, 120] could not be the same object, whereas lines with the ranges of [0 , 50] and [30, 80] could possibly be the same object since they could both be at an angle between 30 and 50 degrees.
3. The line segments must be sufficiently close in 2D space to the hypothetical segment they would create if amalgamated.

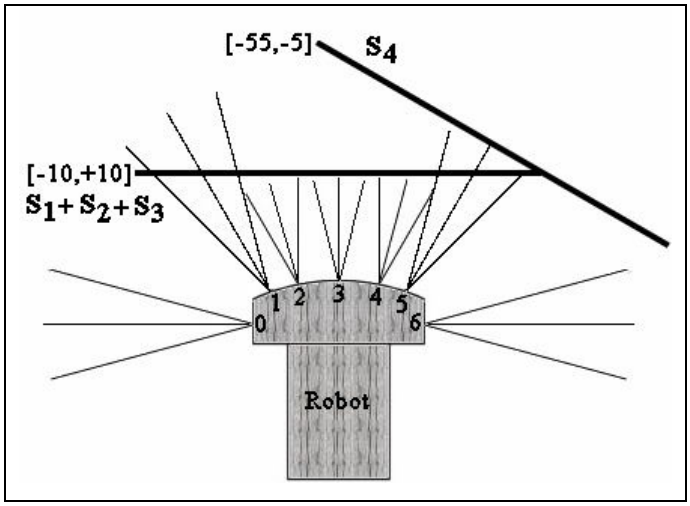


Fig 3.4 Predicted features after Step 2 is applied to the sonar model from Fig 3.3. The three line segments S_1 S_2 and S_3 have been merged to create one segment.

The first two conditions are simple to evaluate, but the third requires some explanation. To determine if a line segment is close to another, we use a line matching algorithm similar to that used in [15], albeit slightly modified. Draw a box around the first line segment, as in Fig 3.5, and if the other line segment passes through this box or is completely contained within it then is close enough to the first line. The size of the box is a parameter of the algorithm, but setting each corner to be 200mm from the nearest end point of the first line in both the X and Y direction has been found to yield good results. A larger value will cause more features to be matched, but if too large a value (i.e. a value much greater than the diameter of the robot) is used, in tight spaces features on opposite sides of the robot may be mistakenly judged to match each other.

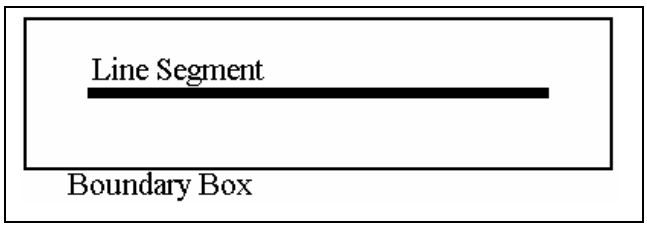


Fig 3.5 The boundary box drawn around a line segment through which another line segment must pass to be judged to match it.

If the two segments match then both are removed from *S*, and the new segment created from them has a new [MIN,MAX] range of possible angles it can be at which is equal to the overlap of both parent segments' angle ranges. For example, if the two segments had angle ranges of [0 , 50] and [30 , 80], then the child segment will have an angle range of [30 , 50]. This can be explained by the fact that the first parent had an angle somewhere between 0 and 50, but we didn't know exactly where. It could have been 10^o, it could be 45^o, with just one sonar reading there is no way of knowing. By matching the first parent with the second, which has an angle range of [30, 80], we know that the first line can no longer be at an angle of <30 or it would be outside the angle range of the second line, so it's new minimum possible angle is 30. It also can't be at an angle of >50 or it would be outside the angle range of the first line. The child segment is added to *S*.

To finish the amalgamating of the two segments, translate the child segment so that the perpendicular distance between it and the midpoints of each of its parent features is equal.

Fig 3.4 shows the sonar model *S* from Fig 3.3 after Step 2 has been applied to it. Line segments s_1, s_2 and s_3 have been merged because they were sufficiently close to each other and their range of possible angles overlapped. The single line replacing them has a possible angle range of [-10,+10], which is much better defined than any of its three parent lines, each of who could have been at any of fifty angles (give a sonar beam aperture of 25^o), rather the possible twenty angles the new line can be at. Line segment s_4 was not found to match any of the other lines so it is unchanged at this point.

3.3.3 Step 3: Remove any line segments in *S* caused by incorrect readings

As it is desirable for the predictive model of the world to be as accurate as possible, it is necessary to identify features created from sonars readings that are the result of specular reflections and to discard them. While many specular reflections that return a reading, usually due to multiple reflections, are long enough that they are beyond the maximum range horizon, some are not. Sonars sometimes exhibit this behaviour in corners where the sonar beam hits first one side of the corner, then the other before coming back. Since both walls are very close together it appears as if there is an

object not far behind the actual corner. Readings such as this must be identified and removed from the model.

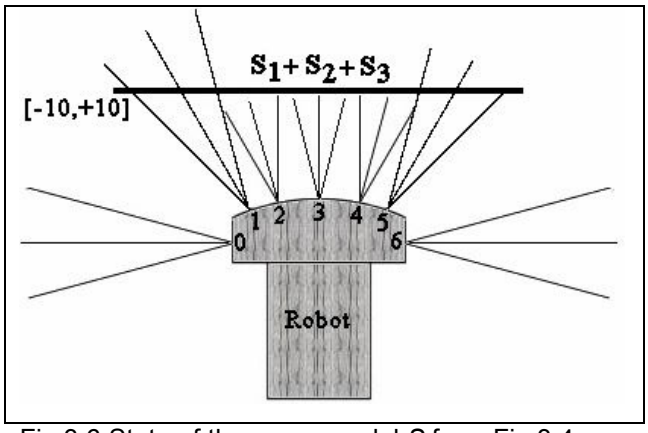


Fig 3.6 State of the sonar model *S* from Fig 3.4 after Step 3 has been applied. Sonar 5 is inconsistent with the line created by sonars 2, 3 and 4, and its confidence is low enough so that the line predicted by it has been removed.

To identify erroneous features the sonar is modelled by three lines – one which marks the left edge of the beam, one which marks the centre of the beam , and one to mark the right edge. Each line begins at the sonar and has a length equal to the range reading returned by that sonar (Fig 3.7). If any of the three lines intersects with one of the line segments in *S* then its confidence measure is changed. The confidence measure of each sonar is then used to remove erroneous line segments from *S* before they can be integrated into the local model *L*. Note that this confidence measure is not the final one used, and is later discarded before being recalculated. This ensures that confidence values based on a sonar being inconsistent with an erroneous feature are ignored.

The manner in which the confidence in the sonar is changed, and to what degree, depends on the angle of the sonar to the line segment it intersects as well as the difference in distance between the range reading of the sonar and the distance from the sonar to the line segment. The initial confidence in the sonars reading, c^s, is 0.5. The *s* superscript specifies that the confidence measure is

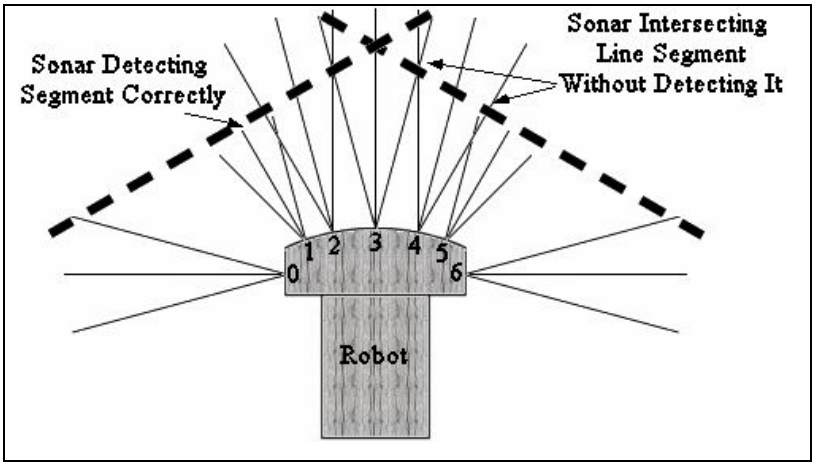


Fig 3.7 Modelling the sonar beams from Fig 3.2 (a) with the robot facing into a corner using three line segments. The dashed lines represent the hypothetical line segments predicted from the range readings from sonars 1 and 5.

based on the sonar view, and not the local model. The reason for this is that the final confidence measure is intended to be used in a Bayesian update formula to reduce the degree of the update being applied to a map. If the sonar intersects with no features and the initial confidence measure of 0.5 is unchanged, then it will have no effect on the map update procedure. If the value is reduced to less than 0.5 due to the sonar intersecting a feature without detecting it, then it will reduce the strength of the map update procedure, as it should. Whenever a new set of sonar readings are added, all the sonar confidences are reset to 0.5, a new sonar model *S* is generated, integrated with *L* and the confidence values are recalculated.

In order to identify erroneous segments confidence values must be assigned to the sonars. Two confidence values are computed for each sonar that is not consistent with a line segment. The first, c_a^s, is the confidence in the sonar with regards to the difference in angle between the sonar and the line segment. The second, c_d^s, is the confidence in the sonar in relation to the difference between the range reading and the distance from the sonar to the line segment.

How c_a^s and c_d^s are calculated depends on the difference in angle between the sonar and the feature it intersects with. There are two possible scenarios.

1. The angle between the sonar and the feature is such that the sonar *should* detect the feature if it were there. This is when the angle between the sonar and the feature is within 'sonar width' of perpendicular. In this case alter the confidence in the sonar *and* decay the feature.
2. The angle is such that the sonar should not detect the feature, given that the surface is smooth enough to cause a coherent transmission of the wave's energy. In this case alter the confidence in the sonar, but do not decay the feature.

3.3.3.1 Scenario 1: The Sonar Should Detect the Feature But Does Not

In the first scenario, where the sonar should detect the feature but doesn't, something is obviously wrong with the predicted feature – perhaps it is shorter than previously estimated. In this case the closer the difference in angle between the sonar and the feature is to perpendicular the higher the confidence in the sonar's reading, therefore

c_a^s should approach 0.5 as the sonar beam gets closer to perpendicular to the feature. The distance between the sonar and the feature is immaterial in this scenario since regardless of the distance, as long as the object is less than the maximum range from the sonar and at the correct angle the sonar should detect it. c_d^s should therefore be equal to 0.5. Theoretically, in the Bayesian formulas described later for using confidence values when updating the global map, values greater than 0.5 will cause the overall value to rise, and values less than 0.5 will cause it to fall, with values equal to 0.5 will have no effect whatsoever on the final outcome. Therefore the closer to 0.5 the value of either c_a^s or c_d^s the less of an effect either of them will have on the overall sonar confidence measure. However, c^s never goes above 0.5, which means that the confidence value generated by feature prediction is not used to strengthen the global map update. If c^s is left at its default value of 0.5, the global map update is not weakened. The lower the value of c^s the weaker the global map update.

In the case where the sonar *should* detect the predicted feature but doesn't:

$$c_d^s = 0.5$$

$$c_a^s = 0.5 - \left(\frac{(\theta - (90 - \omega))}{\omega} \right) * 0.25$$

where θ is the difference in angle between the sonar and the feature, and ω is the beam aperture. This formula for c_a^s maps its value in the range [0.25,0.5] which means that a feature that should be sensed by a sonar but isn't can, at worst, halve the confidence in the sonar (see Fig 3.8). c_a^s is restricted to a range of [0.25, 0.5] because there is an error with the estimated features, and so their effect on the confidence value of the inconsistent sonar is decreased.

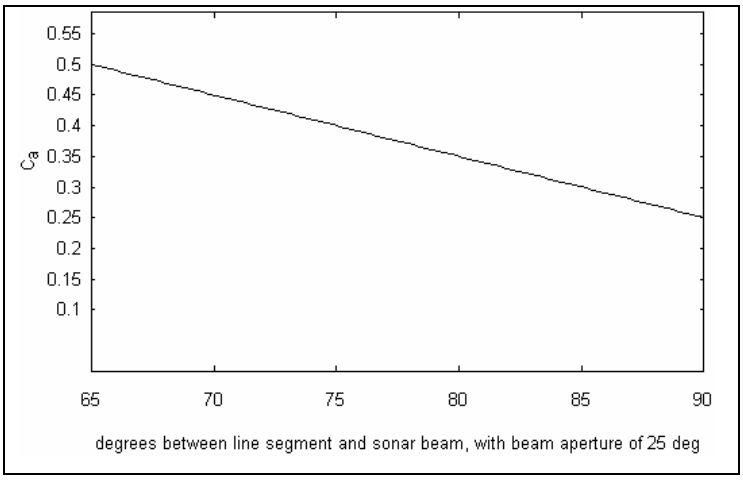


Fig 3.8 Calculating c_a^s when the sonar should sense an object but doesn't, using a sonar beam aperture of 25°.

3.3.3.2 Scenario 2: The Sonar Should Not Detect the Feature

In the second scenario, where the sonar beam intersects the predicted feature at an angle such that we would *not* expect it to return a correct reading, the method of calculating c_a^s and c_d^s changes. In this case it is the confidence with regard to angle that is invariant. This is due to the fact that if, for example the sonar is at an angle of 50^o to the feature it would not be expected to return a reading any more than it would at an angle of 10^o to the feature. That is, once it has passed the threshold ω degrees from perpendicular, we do not expect to receive a correct reading from it.

The confidence with relation to distance, c_d^s, is another matter. As can be seen in the sonar beam model in Fig 3.9 (a), when modelling the freespace part of the beam the closer the cell is to the sonar emitter the more strongly it is believed that the cell is unoccupied. With this in mind, note that the purpose of using a scaled confidence value rather than a simple 0 or 1 is so that cells in the sonar beam in the region of the feature and beyond it are unaffected while leaving the cells in front of the feature be affected by the reading, albeit by to a lesser degree. When the feature is far from the sonar emitter, as in Fig 3.9 (a), only a relatively high confidence value is needed to prevent the cells on the at the feature and on the far side of it from being incorrectly labelled as unoccupied while allowing most of the other cells in the beam to be correctly labelled as empty.

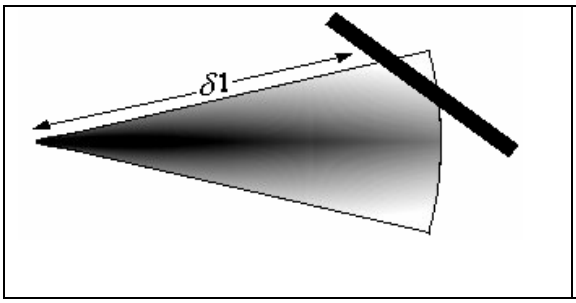


Fig 3.9 (a) Sonar beam with an undetected feature at distance δ1 from the emitter. The darker the area, the more strongly it is believed to be unoccupied.

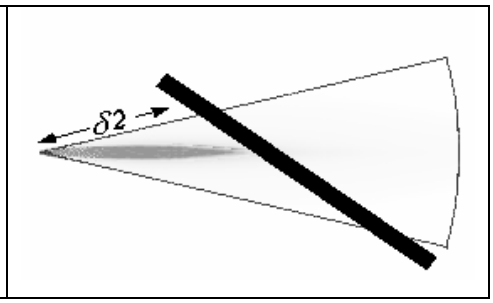


Fig 3.9 (b) Sonar beam with undetected obstacle δ2 from the emitter. The strength of the freespace update has been reduced.

However when the obstacle is nearer to the sensor, as in Fig 3.9 (b), a lower confidence value is required in order for all the cells at and beyond the feature to unaffected by the freespace reading. For this reason, c_d^s must be relative to the distance of the obstacle from the sonar, δ, and the length of the range reading R.

Moravec and Elfes' [46] gaussian sonar model for the sonar beam claims that the sonars power reduces at a rate of $(\delta/R)^2$ the further from the sonar the beam travels, where δ is the distance of the cell being considered from the sonar and R is the range reading returned by the sonar. The same ratio is used here with c_d^s being calculated as:

$$c_d^s = \frac{(\delta/R)^2}{2}$$

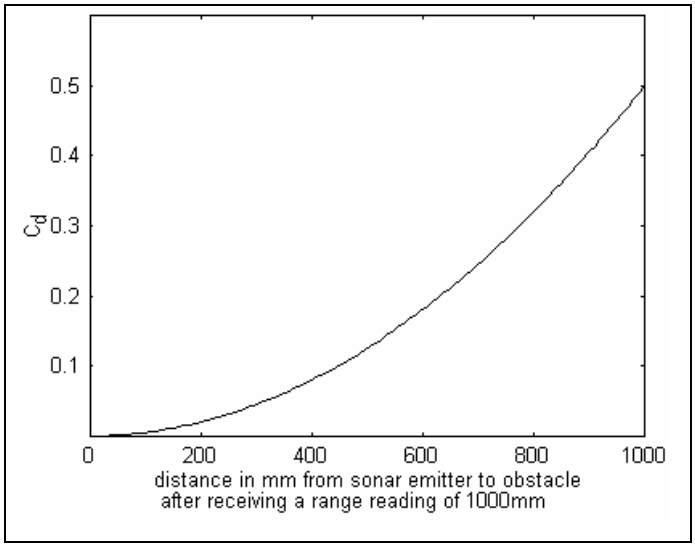


Fig 3.10 Calculating c_d^s with a range reading of 1000mm. The closer the obstacle comes to the range reading received by the sonar, the higher the confidence in the sonar with regards to distance, c_d^s.

This maps the value c_d^s into the range [0, 0.5], with its value approaching 0.5 the farther from the sensor the predicted feature is. As mentioned earlier, c_a^s remains invariant in this scenario, so:

$$c_a^s = 0.5$$

This means it will not affect the result one way or another, as it should not since as the angle changes it does not affect how the feature should be treated.

Once c_d^s and c_a^s have been computed, they are combined to give a single confidence value for the sonar, c^s. c^s is given a starting value of 0.5 because it is intended to be used in a Bayesian update procedure in the map building process using the formula:

$$P(OCC)' = \frac{P(OCC) * c^s}{P(OCC) * c^s + (1 - P(OCC)) * (1 - c^s)}$$

A value of $c^s = 0.5$ in this formula will make the probability of occupancy, $P(OCC)' = P(OCC)$, i.e. nothing happens to the occupancy value.

The two confidence values, c_d^s and c_a^s, are combined first with each other, and then with the current confidence value for the sonar, c^s, using an approach inspired by techniques for parallel combination of rules in expert systems [29]. A rule was developed for the combination of certainty factors in the expert system MYCIN which combined two certainty factors in the range [-1,+1] called *a* and *b*. The rule states:

$$CF(a,b) = \begin{cases} 1-(1-a)*(1-b) & \text{if a > 0 and b > 0} \\ -CF(-a,-b) & \text{if a < 0 and b < 0} \\ a+b & \text{otherwise} \end{cases}$$

The range of confidence values used in MYCIN, [-1,+1], differs from the range used here, [0, 0.5]. In order to use the rule above, the values in the range [0, 0.5] are analogous to the values [-1,0] as they can both be seen as negative confidence in an hypothesis. To convert a number in the range [0,0.5] to the range [-1,0], subtract 0.5 from it and divide the result by two. For example, 0.4 in the range [0,0.5] becomes

$$(0.4-0.5)*2 = -0.2$$

in the range [-1,0]. For the sake of simplicity, the function *R* is used to convert a number from the range [0,0.5] to the range [-1,0], and is defined as:

$$R(n) = (n-0.5)*2$$

The function R^{-1} is used to convert a number from the range [-1,0] to the range [0,0.5], and it is defined as:

$$R^{-1}(n) = \frac{n}{2} + 0.5$$

Therefore, rather than pass the actual confidence values c_d^s and c_a^s to the *CF* function, pass $R(c_d^s)$ and $R(c_a^s)$. The confidence value created by the *CF* function is in the range [-1,0], and must be converted back into the range [0,0.5] using R^{-1}. The values c_d^s and c_a^s will always be ≤ 0.5 so their converted values will be ≤ 0. This means that only the second and third parts of the rule are required, and the rule can be simplified as follows:

- If $R(c_d^s) < 0$ and $R(c_a^s) < 0$, i.e. $c_d^s < 0.5$ and $c_a^s < 0.5$ then the combined confidence value c_{da}^s is:

  $c_{da}^s = R^{-1}\left(CF\left(R(c_d^s), R(c_a^s)\right)\right)$ where

  $$\begin{aligned} &CF\left(R(c_d^s), R(c_a^s)\right) \\ &= -CF\left(-R(c_d^s), -R(c_a^s)\right) \\ &= -1 + \left(\left(1 - -R(c_d^s)\right) * \left(1 - -R(c_a^s)\right)\right) \\ &= R(c_d^s) + R(c_a^s) + R(c_d^s)R(c_a^s) \end{aligned}$$

  The *CF* function must then be converted back a number in the range [0,0.5] using R^{-1}:

  $$R^{-1}\left(R(c_d^s) + R(c_a^s) + R(c_d^s)R(c_a^s)\right) = \frac{R(c_d^s) + R(c_a^s) + R(c_d^s)R(c_a^s)}{2} + 0.5$$

  but since $R(c_d^s) = (c_d^s - 0.5) * 2$ and $R(c_a^s) = (c_a^s - 0.5) * 2$, the above formula simplifies down to

  $$\frac{2(2c_d^s c_a^s) - 1}{2} + 0.5 = -0.5 + 0.5 + 2c_d^s c_a^s$$, therefore

  $c_{da}^s = 2c_d^s c_a^s$ when $c_d^s < 0.5$ and $c_a^s < 0.5$

- If either $R(c_d^s) = 0$ or $R(c_a^s) = 0$, i.e. $c_d^s = 0.5$ or $c_a^s = 0.5$ then the combined confidence value, c_{da}^s is additive:

  $$\begin{aligned} c_{da}^s &= R^{-1}\left(CF\left(R(c_d^s), R(c_a^s)\right)\right) \\ &= R^{-1}\left(R(c_d^s) + R(c_a^s)\right) \\ &= \frac{(c_d^s - 0.5) * 2 + (c_a^s - 0.5) * 2}{2} + 0.5 \end{aligned}$$ therefore

  $c_{da}^s = c_d^s + c_a^s - 0.5$ when either c_d^s or c_a^s is equal to 0.5.

The CF function is applied twice in order to calculate the confidence in the sonar. The first time it is applied it integrates c_d^s with c_a^s to give their combined confidence value c_{da}^s, as described above.

The second time the *CF* function is applied, it merges c_{da}^{s} with the previous confidence in the sonar, c^{s}, to give the new confidence in the sonar, $c^{s}{}'$. All the maths from earlier carries over, so:

$$c^{s}{}' = 2 * c^{s}{}' * c_{da}^{s} \text{ if } c^{s} < 0.5\, C < 0.5 \text{ and } c_{da}^{s} < 0.5, \text{ otherwise}$$

$$c^{s}{}' = c^{s} + c_{da}^{s} - 0.5$$

The reason for this is that the sonar may intersect with more than one feature, for example in a corner, so this incremental update allows each sonar's confidence measure to be modified for each feature it is inconsistent with. A new confidence measure c_{da}^{s} is computed for each line segment that the sonar intersects with, and this is combined with the sonars current confidence value. As stated earlier, the initial confidence value of each sonar is 0.5.

Once all sonars have been assigned a confidence measure, all line segments in the sonar model *S* that were detected by just a single sonar, with that sonar having a confidence measure, c^{s}, of less than a value T_S, are removed. As before, thresholds are subjective, but a value of around 0.35 has been found to work quite well. The reason for stipulating that a feature must be detected by just a single sonar in order to remove it, is that it is highly unlikely that two sonars will both receive specular readings in the same scan that are similar enough to seem to form a feature at an angle which is valid for both sonars. Therefore if two sonars agree on the presence of a line segment, even though they pass through another predicted line segment it is better to err on the side of caution and believe them.

Once the above has been completed the procedure is repeated, but this time comparing the sonar beams to the line segments in the local model *L*. The confidence measures c^{L} are calculated and any segments in *S* that were caused by just one sonar with that sonar having a confidence of below T_S are removed.

At this point in the algorithm, any features in the sonar view *S* that are inconsistent with other features either in *S* or in the local model *L* have been removed. All sonar confidence values c^{L} must now be reset, as they are based on the local model *L*

before the sonar view has been integrated with it. They will be recalculated in step six, when the sonar view has been fully integrated with the local model

3.3.4 Step 4: Compare all remaining line segments in *S* with all lines in *L*, refining the hypothesis of the angle and position of the lines in *L*

This step is similar to step two, in which all line segments in *S* were compared with each other to determine whether or not they were sufficiently similar in orientation and distance from each other to represent the same feature in the environment. However, for this step, instead of matching s_i against s_j, each line in the sonar model S is compared with every line in the local model *L*. Whichever l_j offers the best match, i.e. has the smallest possible angle range, is merged with s_i. If a S_i does not match any line in *L*, it is added to *L* as a new segment.

The process of merging the two lines is very similar to the method used in step two, but now in addition to changing the range of possible angles for the merged line, it must also be translated to position between the two segments it is formed from. In step two it is placed exactly half way between the two parent features s_i and s_j, but that was because both segments had been formed from just a single reading and therefore nothing distinguishes one line as being better than another. Now there is the possibility that l_j has been detected from many different positions and orientations so its pose is very well defined – this is represented by the fact that its range of possible angles is very small. This feature must therefore be given a higher precedence than a newly created feature when deciding exactly where to place the merged feature. The answer is that it is placed between its two parent features at a perpendicular distance of

$$\frac{(size\ of\ s_i\ angle\ range)}{(size\ of\ l_j\ angle\ range)} * \frac{(average\ distance\ between\ s_i\ and\ l_j)}{2}$$

from the previous feature. Therefore if both lines have an equal angle range so that it is impossible to tell which is better, the merged line is placed half way between them. If the certainty of the previous feature is higher than the new one however, the merged feature is place closer to the previous wall. For example, if the angle range of s_i = [30,80], the angle range of l_j = [50,60], and the average distance between both lines is

10mm, then the combined line will have a range of possible angles of [50,60] (due to the overlap rule) and be placed $\frac{60-50}{80-30} * \frac{10}{2}$ = 1mm from l_i in the direction of s_i.

3.3.5 Step 5: Segments in *L* too far from the robot are removed

The map kept by this model is a local map, and only records features in the close vicinity of the robot. This minimises the problems caused by things such as angular drift and slippage, which can reduce the value of a global map, and require localisation techniques to combat them. Each time a segment in *L* is identified in *S*, its position is updated to keep it inside the local model. This means that the features do not lengthen and do not extend outside the local map. So, when a feature in *L* has not been matched in a sonar scan for a given distance T_d, it is simply deleted. Since the local model extends around the robot to a distance of about 2.5 metres (the maximum distance allowed for a sonar reading) on all sides, a value for T_d of 3 metres has been found to yield good results.

3.3.6 Step 6: Generate final sonar confidence values, and split and shorten any segment in *L* that should be in *S* but is not.

At this point in the algorithm, all of the line segments created by the latest sonar scan, the sonar view *S,* have been integrated into the local map *L*. All lines in *L* too distant from the robot have been removed, and all segments caused by specular reflection have been discarded. This step computes the final sonar confidence values, as well as decaying any features in the local model *L* that should have been detected in the most recent sonar scan but were not.

The sonar beam is once again modelled using three lines representing the left edge of the beam, the right edge and the centre as in step 3. If a sonar beam intersects with any of the segments in *L* without detecting it, at such an angle as the sonar would be expected to return a correct reading from them, then the line segment is split in two at the point of intersection of the sonar and the line, and both newly created segments are shortened as in Fig 3.11. This is used to remove old features that are the result of specular reflections that it was not possible to identify from other sonar readings. Any segment that falls below the threshold T_L = 30% of its initial length is removed from the model. At this length, either a number of sonar readings have been

inconsistent with the feature, making it very unlikely that it exists, or it is the truncated segment that has been removed from a feature whose initial length was overestimated. Fig 3.11 gives an example of this in practice as a robot goes past a doorway.

As explained earlier, the sonar should be expected to return a correct range reading from a smooth surface if the surface is within the *sonar width* from perpendicular to it. For example, given a beam aperture of 25 degrees, any smooth obstacle at an angle of between [90–25,90+25] = [65,115] degrees from the central beam of the sonar should be detected by it.

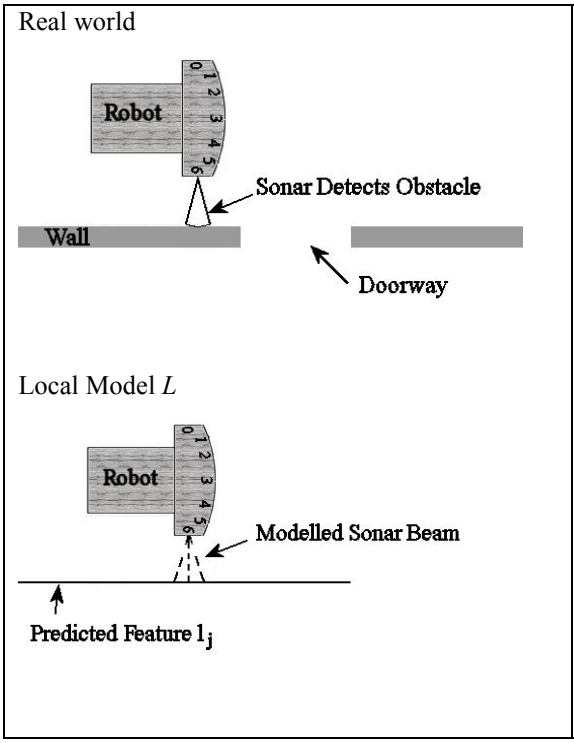


Fig 3.11 (a) The robot detects a wall to its right and models this with the feature l_j.

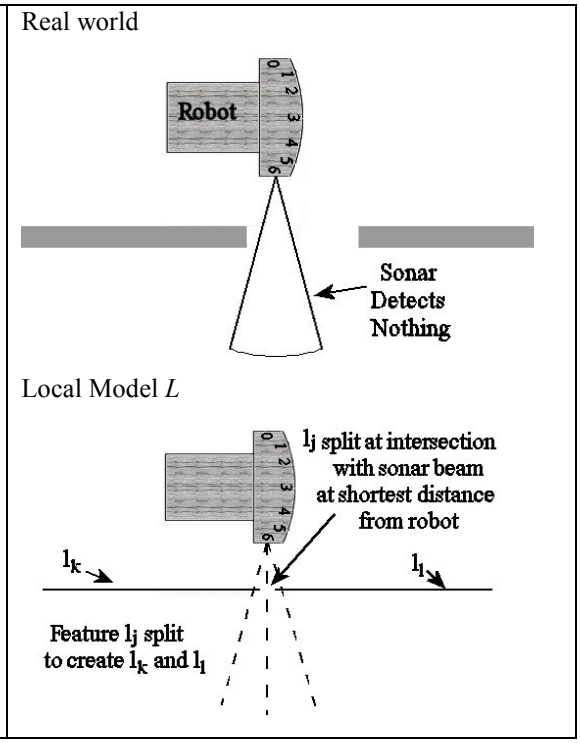


Fig 3.11 (b) Robot's sonar reading is inconsistent with l_j so l_j is split into l_k and l_l .

If the sonar intersects a line segment, and would not be expected to return an accurate reading due to the angle between the sonar beam and the obstacle being too great, then the confidence in the sonar is reduced using the same methods as explained in step 3.

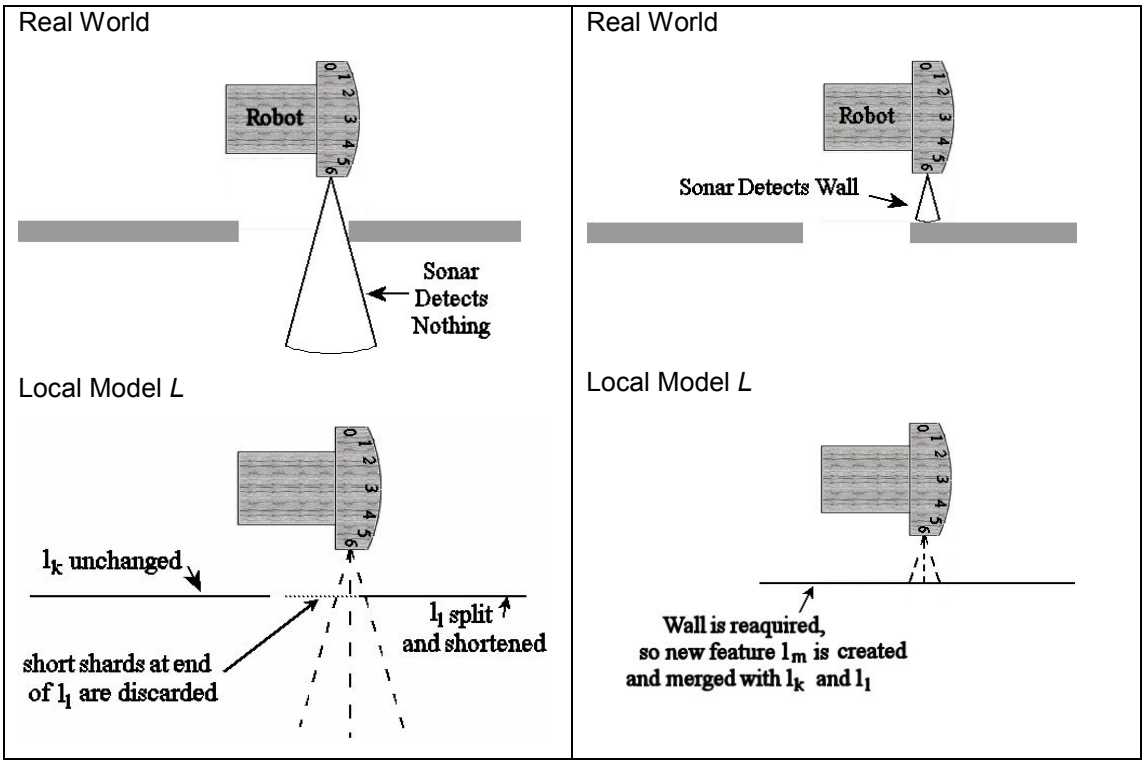


Fig 3.11 (c) Robot's sonar reading is inconsistent with feature l_l, so it is split into two. The short segment at the end discarded as it is below the length threshold T_L.

Fig 3.11 (d) The wall is reacquired, so a new feature l_m is created, and merged with l_k and l_l.

An advantage to splitting an inconsistent feature that should be detected into two rather than using another strategy for decaying it, for example shortening or removing it from L directly, becomes apparent in Fig 3.11 where the robot is passing a doorway. We observe that if a wall is calculated to exist, and it suddenly disappears, there is a strong possibility that it will reappear after a short distance. The line segment l_l can be seen as preventing the wall in front of the robot from being marked as unoccupied by incorrect specular readings. By allowing l_l to exist until it has been shortened to the threshold T_L where we can be sure that the lack of an obstacle in the sonar beam represents a continuous break in the wall rather than a short gap, any obstacle that carries on from a previously detected obstacle will not be marked as unoccupied. This method has been found to perform significantly better on both concave and convex corners also.

A possible disadvantage to continually splitting line segments is that if T_L is too small then many small segments will remain in L without being removed or detected. This

can adversely effect the accuracy of the computation of confidence measures in the sonars, since a sonar may be inconsistent with many small line segments, greatly reducing its confidence when perhaps the sonar reading is fine and those line segments should have been removed. However, keeping T_L above 20% removes this problem since lines greater than this size are much more likely to be detected, split and removed from L.

3.3.7 Step 7 - Using Sonar Confidence Values To Build Maps

The process described above generates a confidence value in the range [0, 0.5], with zero meaning we have no confidence in the sonar reading, and 0.5 meaning that no evidence has been found to cause us to disbelieve the reading. The question now is how confidence values are employed in updating the global map G.

The answer is that it depends on the sonar model being used in the map building process, the range of values being used in the map and how they are generated. The desired end result is that, given the value V that is being used to change the probability of occupancy of a cell, V should be integrated with the confidence measure C such that the degree to which it changes the occupancy value of the cell is lessened.

For example, when updating an occupancy grid using the formulas put forward by Moravec and Elfes in [46], two values are calculated for each sonar beam, P(OCC) and P(EMP), the probability of a cell being occupied and empty respectively. Both these values are in the range [0,1]. To use the sonar confidence value with P(EMP), simply apply the Bayesian update formula to it:

$$P(EMP)' = \frac{P(EMP) * C}{P(EMP) * C + (1 - P(EMP)) * (1 - C)}$$

and do the same for P(OCC). If the confidence value C is 0.5, its default value, then P(EMP) and P(OCC) will remain unchanged, and as C approaches zero, so do P(EMP) and P(OCC).

To use the confidence value with Konolige's map building algorithm MURIEL, it can be treated differently. The MURIEL method, as explained earlier in the chapter, uses a different technique to determine the possibility of a sonar reading being specular,

P(S). As mentioned, it suffers from the problem that it needs sufficient readings at a cell claiming it to be occupied (surface readings) before it can determine if a reading claiming it to be empty is specular or not. However once it has enough surface readings it works quite well. The MURIEL method can work hand in hand with the feature prediction method. Using feature prediction eliminates the greatest weakness of the MURIEL method, that of having to wait until it has enough surface readings before *P*(S) can be calculated properly. It is now possible to calculate the probability of specularity based both on past information (the MURIEL method) and expected future information (feature prediction). To do this, *P*(S) is calculated as usual by the MURIEL method and is then integrated with the confidence factor *C* using the following Bayesian formula:

$$P(S)' = \frac{P(S) * (1 - C)}{P(S) * (1 - C) + (1 - P(S)) * (1 - (1 - C))}$$

The value (1-*C*) is used rather than *C* because the end result is that, as *C* gets smaller *P*(*S*) should rise.

3.4 Conclusion

The feature prediction algorithm and sonar confidence calculation method described in this chapter have been implemented in the *SpecularEstimator* service described in Chapter 4, and is used with the *ME85mod*, *ME88mod* and *K97mod* map building services also described in Chapter 4. An experimental evaluation of the benefits of using Feature Prediction is presented in chapter six by determining the quality of the generated maps both with and without feature prediction being employed.

Chapter 4: Software Architecture for Robotic Experimentation

4.1 Introduction

A combination of two robotic control architectures were used in experimentation for this thesis. Low-level robotic control was performed by the Saphira architecture from ActivMedia, in conjunction with the Aria api. All higher level operations such as mapping, path planning and feature prediction, were performed using the software-services architecture designed and implemented during the course of this research.

This chapter is divided into three broad sections. The first section deals with the Saphira robotic control architecture. Due to the fact that the software modules written for this thesis are completely self contained, and can operate as the deliberative layer on top of any architecture, the Saphira architecture had little influence on the generation of the maps during experimentation. For this reason the Saphira architecture is not presented in depth here. A large body of documentation is available at http://robots.activmedia.com if further information is required.

The second section details the software architecture that performs all high level operations. This was written primarily in C++, using object orientation techniques. A standard interface to all services is presented, using the client/server paradigm. A single point of entry for data is enforced, which the various services can access in either a synchronous or asynchronous manner, for real time and non-real time applications respectively.

The third section explains in detail each component of the overall architecture. These include the objects responsible for the following areas:

- Map storage.
- Map generation.
- Client registration to a service.
- Thread management/simultaneous access to resources.
- Any-time feedback from server to client.

- Feature prediction algorithms.
- Pose bucket maintenance.
- Interfacing the Saphira architecture with the software architecture presented later.
- Path Planning.

There are six map building systems presented, *ME85*, *ME88, K97, ME85mod, ME88mod* and *K97mod*. The first three systems are based directly on the theories of Moravec and Elfes, Matthies and Elfes, and Konolige respectively, although they contain completely new code. The final three systems, each ending with the term *mod*, contain modifications on the original theories of the authors just mentioned. These modifications include:

- Feature prediction, to calculate the probability of specularity of sonar readings from predicted features.
- An enhanced version of Konolige's *dynamic mixture model* for calculating the specularity of sonar readings from previous information.
- In the case of *ME85mod*, the formulae used to incorporate sonar readings into the map have been altered to remove the bias favouring freespace readings over readings reporting an obstacle.

4.2 The Saphira Architecture for Autonomous Mobile Robots

The Saphira architecture is an integrated sensing and control system used to control autonomous mobile robots. It uses a client/server paradigm that abstracts Saphira from the particulars of any one robot, making it very portable, as long as the robot adheres to the server protocol. The server is responsible for low-level operations, such as monitoring the sensors and motor control, based on commands from the client.

While there are numerous robot simulators available for use with the Pioneer 1 robot, Saphira stood out as the best option for four reasons. Firstly it uses a robust and pragmatic programming interface through the ARIA C++ libraries which is completely object oriented, leading to easier and better design. Secondly, it was designed with ActivMedia (creators of the our robot the Pioneer 1) robots specifically

in mind. Thirdly, it is very well supported, with a large community of users and an active and helpful support team. Finally, it is very cost effective, with a single license entitling the buyer to both Saphira, ARIA, and all future upgrades.

In the Saphira architecture, the robot is considered to be the server, which the Saphira client connects to and communicates with, as in Fig 4.1. The client is responsible for low level operations, and sends direct motion commands to the robot server. The server sends an information packet to the client every 100ms containing position, velocity and sensor readings. While the server is always on-board the robot, the client can be on-board or off-board, on a host computer. The software architecture created for this thesis acts as a second server which the Saphira client communicates with. This server is responsible for the more high-level operations, such as mapping.

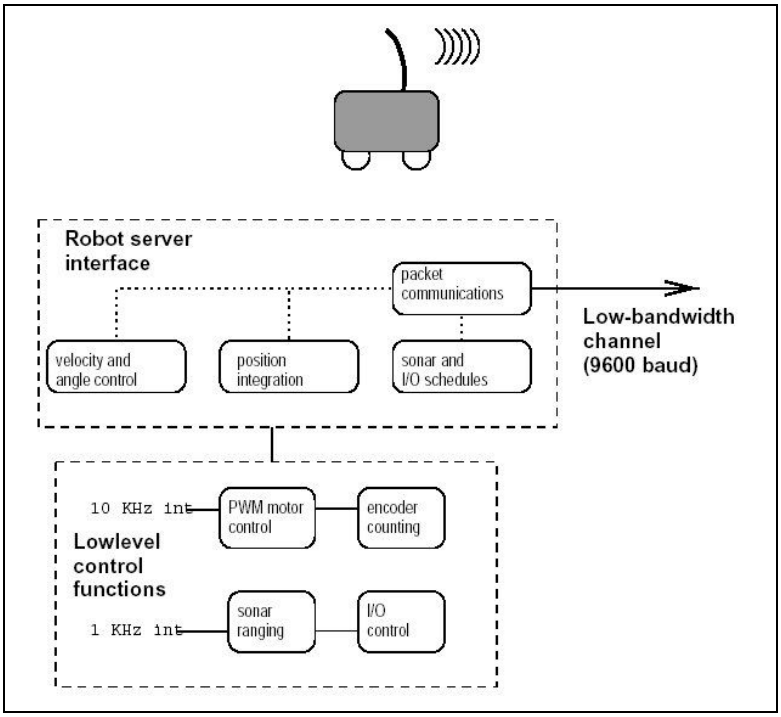


Fig 4.1 Saphira server architecture

The organisation of the Saphira architecture is partly vertical and partly horizontal. Operations can be carried out in both real-time and any-time. Operations that are required for real-time control, such as obstacle avoidance and wall following, can be completed within a single processing cycle of the robot. Non time critical operations such as map building and localisation, on the other hand, can operate asynchronously to the low level operations without impeding them in any way.

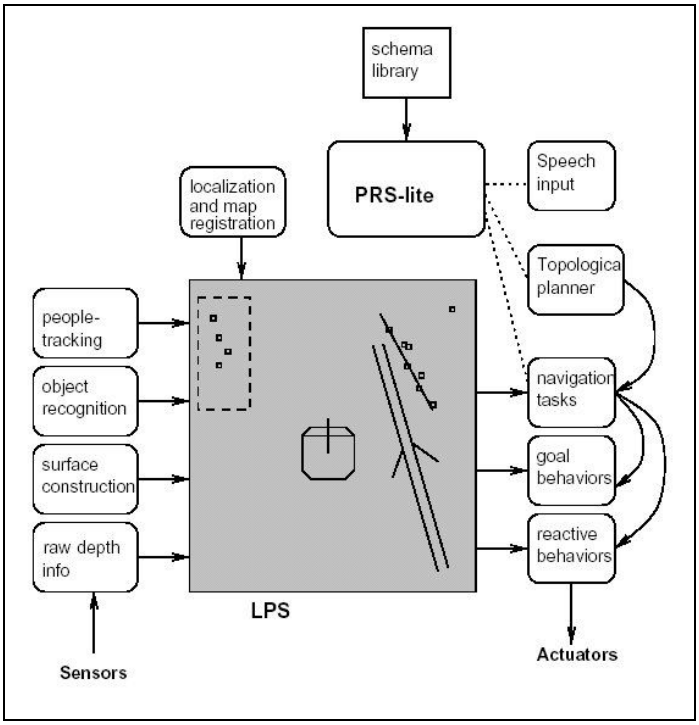


Fig 4.2 – Overall Saphira architecture for robotic control.

Saphira contains a central repository for all information is has gleaned from the environment, called the Local Perceptual Space (LPS) seen in Fig 4.2. All sensory data, such as sonar, laser and vision, is incorporated into this model. A form of mapping and localisation is possible using this world model. However, the LPS was not used, as it was determined that the LPS was not appropriate for the experimentation required for a number of reasons:

- The world is modelled as a set of features, such as corridors, doors and walls. These features are not probabilistically represented, and are added/removed based on an *ad hoc* approach. It was infeasible to create grid based maps with this framework, as it was based on a fundamentally different modelling paradigm.
- The LPS is currently still a research effort, and far from perfect. It can often be observed to make gross errors, with the resulting robotic performance being far from optimal.
- The LPS is tied firmly to the Saphira architecture. It is desirable for the modules developed for this thesis to be as flexible and portable as possible. Therefore the architectural decision was taken to write all code from scratch, enabling it to be used with any robot configuration, as the deliberative layer of any robot control architecture.

4.3 Software Systems Used in Experimentation

A set of software systems were designed and implemented for this thesis in C++ for the purpose of examining the difference in performance between different map building methodologies put forward by various researchers, as well as investigating the effect of modifications on the quality of maps generated.

4.3.1 Mapping algorithms represented as Service Providers to a Client

Six software map building modules were developed, each being a standalone service capable of running either in a distributed system, ideally with one service per machine, or all on a single machine. Distributing multiple services on a number of machines generally results in superior performance, as some services are very processor intensive, and running both multiple mapping services on a single desktop workstation (the development environment used due to cost considerations) can lead to delays in processing the sensor readings. Either multiple computers, connected using a distributed medium such as CORBA, or a more powerful machine, such as a SPARC station, are required to make the robot operate in real time. However, a distributed framework allows for greater extensibility than a single high-powered machine, as adding new services without impacting real time performance can be achieved by simply adding a new computer to the network.

4.3.2 Distributed Robotic Framework

A distributed CORBA framework for mobile robot control was developed in conjunction with others in the robotics group in the University of Limerick. This framework distributes the mapping services created for this thesis, as well as services designed by other members of the group, such as path planning, mapping systems based upon neural net techniques, and pursuit-and-evasion behaviours. Each of these services allows a client to register with them, takes in the sensor readings and places them in a queue to be processed in whatever way is appropriate to that service. This framework enabled the simultaneous testing of all six map building systems with all of them running in parallel off the same robot in real time, and ensured that each service received identical sensor readings. The maps generated by each system could then be compared directly with each other, as they all related to exactly the same area and the same test run. See Appendix A for diagrams of the distributed architecture.

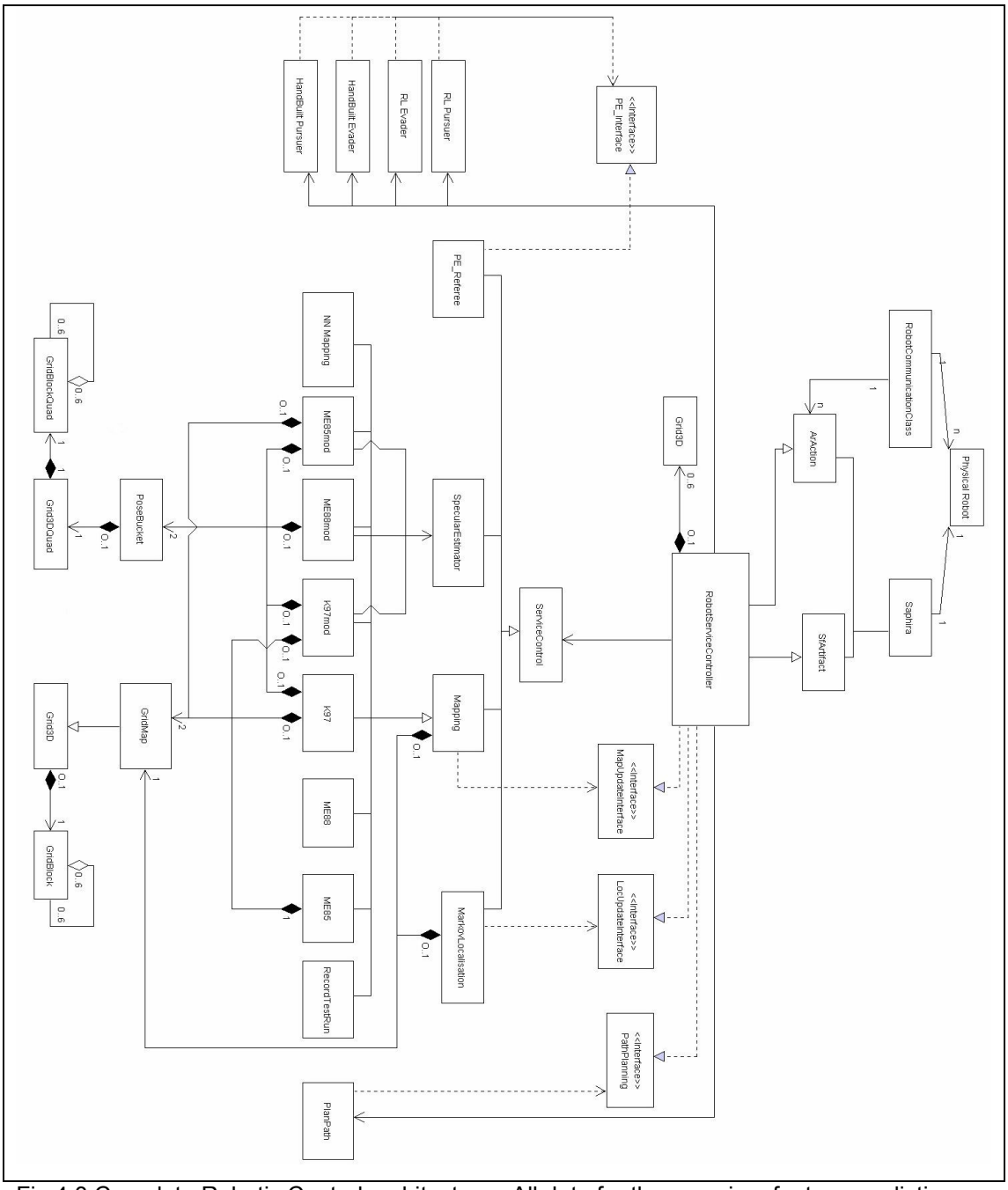


Fig 4.3 Complete Robotic Control architecture. All data for the mapping, feature prediction and localisation modules flows through the *ServiceControl* object. All modules can be co-located on a single machine, or there can be a machine boundary between the *RobotServiceController* and the *ServiceControl* object. This diagram includes research undertaken both for this thesis, and also by other researchers in the UL Robotics Group. A subset of the modules in this architecture were used in experimentation, see Appendices for additional diagrams.

4.3.3 Mapping Systems Summary

The naming convention used in the following systems is as follows:
Each system is based on the theory put forward in a paper published by a certain author in a given year. The name of the system begins with the first letter(s) of the surname of the author(s) of the paper. This is followed by the year of publication of the paper. E.g. *ME88* is the system based on the 1988 paper by Larry Matthies and Alberto Elfes. If the system features modifications/improvements on the original theory, its name is followed by *mod*.

The systems designed are as follows.

System Name	Description
ME85	Based on the map-building methods described by Moravec and Elfes in 1985 [46]. It contains the following features: • Two-dimensional gaussian sonar model. • *Ad hoc* mathematical map update method. • Not based on probability theory. • Bias towards favouring freespace readings over surface readings. See Chapter 2 for further information
ME85mod	Similar to *ME85*, but with the following modifications. • Feature prediction is used to identify specular readings • Improved Dynamic Mixture Model used to identify specular readings from previous information gathered. • No bias towards freespace readings.
ME88	Based on the system designed by Matthies and Elfes in 1988 [42]. It contains the following features: • Two-dimensional gaussian sonar model identical to *ME85*. • Bayesian mathematical update procedure. • Based on probability theory. See Chapter 2 for more information.

ME88mod	Similar to *ME88*, but with the following modifications: • Feature prediction is used to identify specular readings • Improved Dynamic Mixture Model used to identify specular readings from previous information gathered.
K97	Based on the system designed by Konolige in [33]. It contains the following features: • Sonar model based on the normal distribution. • Probabilistic mathematical map update methods based on Bayesian theory and log odds formulae. • Pose buckets used to discard duplicate readings. • Simplistic Dynamic Mixture Model used to identify/discard specular readings based on previously gathered information. • Conservative approach taken towards obstacle detection – biased towards detecting obstacles rather than freespace regions.
K97mod	Similar to K97, but with the following modifications: • Feature prediction is used to identify specular readings • Improved Dynamic Mixture Model used to identify specular readings from previous information gathered. • Less biased towards obstacle detection than *K97*.

Fig 4.4 Outline of the features of each of the six map building systems developed for this thesis.

4.4 Architectural Modules

The systems outline above are comprised of, and interact with, a number of different modules. The modules developed were:

1. *Grid3D*: The *Grid3D* class handles storage of the map, and is used by each map-building system being tested. Handles general grid/map storage organised in a 3D lattice. Performs memory management for a map, growing the map to whatever size required, while at the same time only consuming the necessary amount of resources.

2. *GridMap*: The *GridMap* class inherits from the *Grid3D* class, and add extra functionality to it, such as blurring, performing comparisons between maps, and converting the feature-based maps used by Saphira into grid-based maps.
3. *PoseBucket*: A C++ class that contains a G*rid3D* class. Rather than containing probability values in a Cartesian map like *GridMap*, it contains a list of Boolean values for each Cartesian point in the map stating whether or not a sensor reading has been received for that cell from a given angle at a given distance. The angle resolution and distances are user definable, but the values used in Konolige's work [33] are 60 different possible angles (Konolige actually uses 64 angles), and three possible distances. This gives 180 Boolean values for each Cartesian coordinate in the map.
4. *RobotServiceController*: A C++ class to interface with Saphira, take in the readings from the sensors, and pass them along to whichever robot services it is currently interfacing with.
5. *CommandClient*: A program used to interface with the *RobotServiceController* class in order to allow the user to interface with the dynamically linked library in Saphira. This enables the tester to tell the *RobotServiceController* to load map systems and to pass along commands to them, such as to blur the map, save the map etc.
6. *ServiceControl*: An abstract base class that enables clients, for example the *RobotServiceController* object, to register with a service, and enqueues the readings sent from the client for further processing by the services that inherit from it.
7. *Mapping*: A C++ class that contains all the functions and attributes common to all map building systems. These include geometrical calculations like determining if a point is within the sonar beam, as well as containing the primary map used to store the model being built. This class is also responsible for performing callbacks to the client of the service, notifying them of changes in the map.
8. *ME85*: A C++ class that updates a map using the methods developed down by Moravec and Elfes' 1985 paper [46].
9. *ME85mod*: A modified version of *ME85*, with additions made for this research to compensate for certain shortcomings in the original model, such as the grid

update procedure, and the assumption of perfection in the sensors. This is explained later.

10. *ME88*: A C++ class very similar to *ME85*. It accepts sonar readings from a client and processes them, but instead of using Moravec and Elfes' grid update procedure, it calculates the prior and posterior probabilities of each cell using Bayes theorem, as described by Matthies and Elfes in their 1988 paper. See Chapter 2 or [42] for more information.
11. *ME88mod*: A C++ class identical to *ME88*, but with one of the modifications mentioned in *ME85mod* added – modelling the imperfection of the sonar sensors using an improved dynamic mixture model**.** See Fig 4.4 for more detail.
12. *K97*: A C++ class that implements the system described by Konolige's 1997 paper [33]. It uses another module, *PoseBucket*, a modified version of *Grid3D*, to implement *Pose Buckets* (see Chapter 2).
13. *K97mod*: A C++ class very similar to *K97*, but with modifications added to solve certain problems, for example requiring correct non-noisy readings at a cell before noisy readings can be recognised, as well as a modified Dynamic Mixture Model.
14. *SpecularEstimator*: This child of the *ServiceControl* class implements the Feature Prediction algorithms described in chapter 3. It predicts the position and orientation of obstacles in the environment based on the sonar readings and uses these hypotheses to calculate a confidence measure of the accuracy of the sonars readings.

PlanPath: This object plans a path between two points in a given static map using a modified version of the A-Star algorithm.

4.5 Grid3D Class

The *Grid3D* class is a template class, used to store a collection of objects in a Cartesian grid. For the purposes of storing a map, the type of objects it stores are numbers of type *double*. This grid can be either in two dimensions or three dimensions, and extends to be as large as the user needs. The grid is comprised of a set of objects called *GridBlock*, which can be thought of as a miniature grid which, once initialised, is fixed in size. Each *GridBlock* contains links to six other blocks, to

the north, south, east, west, above and below it, as well as knowing its own position in the global map.

The *Grid3D* class manages these *GridBlock*s, arranging them in a linked list. The grid has a given size, and when the user attempts to set the value of a position outside the grid, it automatically grows itself to encompass the new point. The *Grid3D* does this by creating new *GridBlock*s and linking them onto the edge of the grid in the direction of the required point. For example, if the map were of size 1000 x 1000 with x in the range [-499, 500] and y in the range [-499, 500], using *GridBlocks* of size 100 x 100,

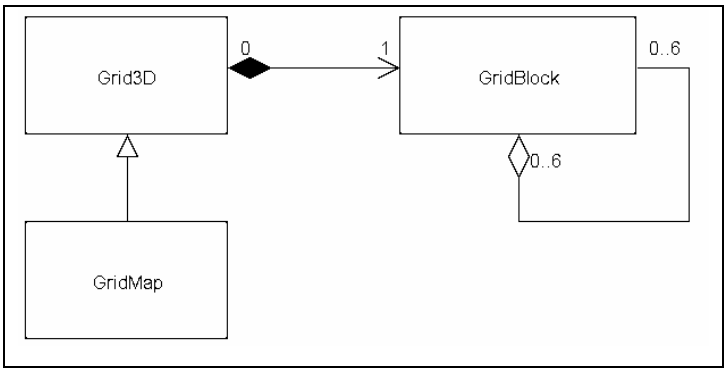


Fig 4.5 Grid3D package containing the gridmap class, mapblock class, and external object WorldFileLexer.

and the user tried to set the value of grid position (150, 750) which is north of the existing grid, two extra rows of *GridBlocks* would be added to the north, each row growing the grid by 100 squares in the northerly direction. This means that the map may not always be square shaped, but it will always be rectangular.

While Saphira does contain its own class for storing a grid, *SfGrid*, it is preferable for the implementation to be both simulator-independent, and also be efficient and extensible. The *Grid3D* class is portable to any simulator that supports C++ or Java, and can even be used independently of any simulator for many other purposes that may only be distantly related to robot simulation. This class also acts as a parent class to the *GridMap* class, which adds extra functionality to the grid (Fig 4.5).

```
template <class T>
class Grid3D
{   public:
        Grid3D();
        Grid3D(int blocksize, int radius, T Unknown, int blockheight = 1);
        virtual ~Grid3D();
        T getGridRef(long x, long y, long z = 0);
        bool updateGridRef( T value, long x, long y, long z = 0);
        void copy(Grid3D<T>* mapToCopy);
        bool save(char* filename);
        bool load(char* filename);
        long getDimensions(int direction);
        long getUpdatedDimensions(int direction);
        const T getUnknown();
        void reset();
    protected:
        void appendBlock(GridBlock<T>* original_block,
                         GridBlock<T>* new_block, int direction);
        int growMap(int times, int direction=0);
        GridBlock<T> * newBlock();
        GridBlock<T>* findBlock(long x, long y);
        GridBlock<T>* myMap;
        int errorVal;
        long blockSize;
        long blockHeight;
        long dimensions[6];
        long updatedDimensions[4];
        T unknown;
};
```

Fig 4.6 Grid3D Class.

4.6 GridMap Class

The *GridMap* class is a specialisation of the *Grid3D* class, i.e. it adds new functionality to the basic grid-managing abilities already in place. This is used to store a two dimensional grid of objects, for example floating point numbers in the case of storing a certainty grid, as well as perform certain manipulations on this data.

```
template <class T>
class GridMap : public Grid3D<T>
{
 public:
  GridMap();
  GridMap(int blocksize, int radius, T Unknown);
  GridMap(GridBlock<T>*mapToCopy,long* dimensions,long blocksz,T defaultVal);
  ~GridMap();

  bool updateGridRef(T value, long x, long y);
  T getGridRef(long x, long y);
  void copy(Grid3D<T>* mapToCopy,int reduceFactor = 1,int valueToSelect);
  void reduceDimension(int reduceFactor = 4, int valueToSelect);
  void boxBlur(int kernelSize=3, double boxVal=1);
  void gaussBlur(int kernelSize=3);
  bool importPointMap(char* fileName, T value=1, long squareSize=100);
  bool addLine(long x1, long y1, long x2, long y2,T value, long squareSize);
  double correlateMap(GridMap<T>* mapToCompare);
  double scoreMap(GridMap<T>* mapToCompare);
  bool growOccArea(long radius, T lowerBound, T upperBound,long squaresize);
};
```

Fig 4.7 GridMap class, which inherits from Grid3D.

These include adding lines to the map, comparing one map with another to find a

fitness match, shrinking the map, importing maps stored in a Cartesian point format, and blurring the map using both box-blurring and gaussian blurring techniques.

4.7 PoseBucket Class

The *PoseBucket* class contains a class of type Grid3Dquad, which is essentially the same as the class *Grid3D*, except that it stores a three dimensional grid in a quad tree representation. The *PoseBucket* class is used by map building systems based on Konolige's [33] papers to record whether or not a previous reading was received from a particular point at a particular angle, as well as by *ME85mod* and *ME88mod*.

As explained in section 2.11, Konolige uses pose buckets to prevent more than one range reading from the same position affecting a particular cell in the same way – this means that only one reading from a given pose can claim that a cell is occupied, and one reading can claim that it is empty. This is achieved by taking in a sonar reading, checking if a reading from this position and angle, or pose, has already been used to update the current cell as being either occupied or empty. If it has then the current reading is discarded. If a reading has not been received from this pose, a flag is set in the *PoseBucket* stating that no more readings from this pose can affect this cell in the same way. Any given square in the grid can be accessed using three coordinates, x, y, and height. The x and y represent the position of the robot, and the height coordinate represents the angle the robot was facing when this reading was taken.

```
class PoseBucket
{
 public:
  PoseBucket(int sqSize,int numberAngles,long divDist1=0,
             long divDist2 = 0, long divDist3 = 0);
    ~PoseBucket();
    bool getVal(long X, long Y, SosPose sonarPose);
    void setVal(long X, long Y, SosPose sonarPose);
    void reset();

  private:
    long squareSize;
    double angleDivisor;
    long divDist[3];
    Grid3DQuad *myMap;

    double getDist(double x1,double y1,double x2,double y2);
};
```

Fig 4.8 *PoseBucket* class

The PoseBucket class offers three operations to the user, *getVal, setVal*, and *reset*. To mark the cell as being previously updated or to check it's value the user gives the

coordinates of the cell in Cartesian coordinates and the pose of the sonar sensor. The distance and angle of the robot to the cell are calculated and the corresponding cell in the three dimensional map is updated or returned respectively. The *PoseBucket* object can be instantiated with up to three different divisions of distance and with as many divisions of angle as needed. Therefore with the settings of sixty divisions of angle and three divisions of distance, 180 Boolean values must be stored for every cell in the map being generated.

In the experiments performed for this thesis, the area surrounding each cell is divided into sixty different angles and by the areas ≤0.5m, ≤1.5m and >1.5m, similar to Fig 4.9. For any single cell, if two different readings that cause it to be altered have originated from the same range of angles and the same distance range, then all readings except the first will be discarded.

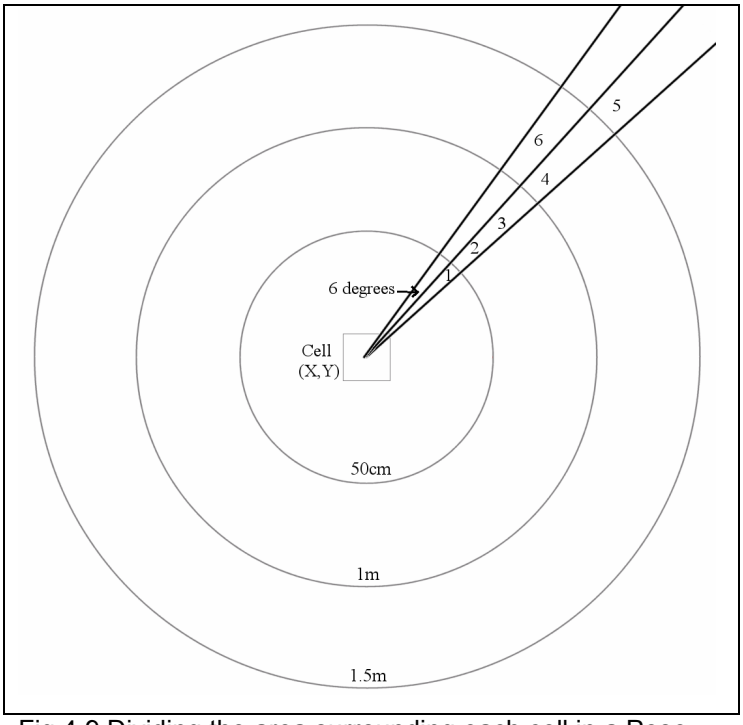


Fig 4.9 Dividing the area surrounding each cell in a Pose Bucket by degrees and distance from the cell.

In this diagram the numbers 1 through 6 represent different positions that readings have been received that affected the cell at position (x,y). Readings 2 and 3 duplicate each other since they both came from between 50cm and 1m from the robot, and from between 42 and 46 degrees from it. Readings 4 and 6 do not duplicate each other as they come from sufficiently different angles even though they are both in the same distance bracket.

One problem with using pose buckets is the prohibitive amount of memory required for any map of non-trivial size. For each cell in the actual map, the *PoseBucket* stores (number of angles * number of distances) values. This means that when using the

values presented above, there are 60 * 3 = 180 values stored for each cell in the map, as well as the occupancy value stored in the map itself. Using the standard method of storing a grid map, where every cell in the map has a corresponding pose bucket of 180 values, the amount of memory required can be in the hundreds of megabytes.

To combat this performance hit, the *PoseBucket* class is implemented as a quad tree, as described in section 2.5 (see Fig 2.4(c)). This means that the amount of memory stored in the map is proportional to its complexity, since memory is only requested for a cell if it is necessary to store information in it, otherwise a default value is returned. Because of the quad-tree approach taken with the *PoseBucket* class, it is possible to scale up to very large maps even on a standard desktop computer.

4.8 RobotServiceController Class

The *RobotServiceController* class fulfils the following roles:

- Receives the sonar readings from the robot.
- Draws the map on the Saphira window.
- Accepts commands from the user via socket-based communication through the Command Client
- Registers with the service(s) requested by the user, for example the *ME85, ME85mod, ME88, ME88mod, K97* and *K97mod* map building systems.
- Passes the sonar readings to all active services for them to use however they deem appropriate.
- Accepts callbacks from the running services which are used to update the local copy of the map, which is then used to draw the generated map on Saphira's window.

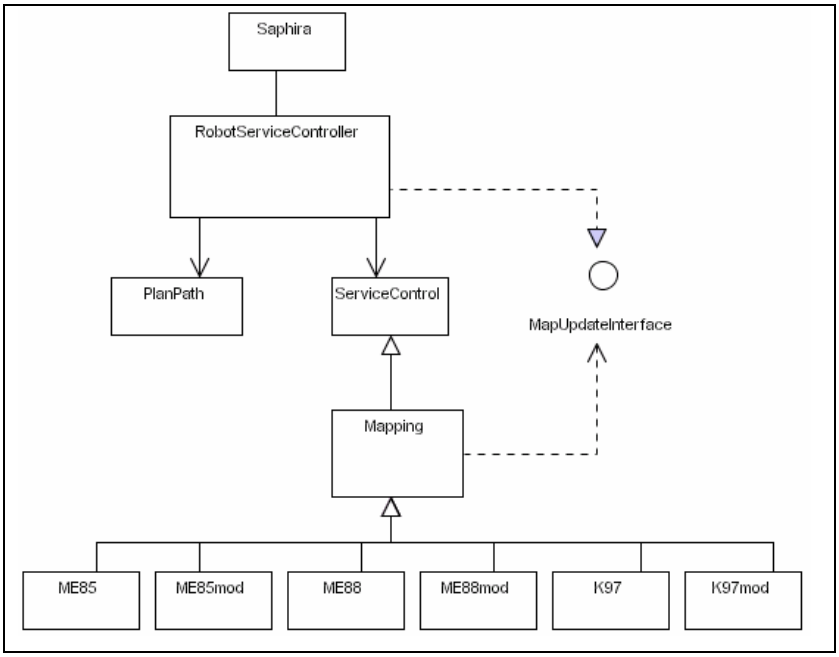


Fig 4.10 *RobotServiceController* class interacting with Saphira and the Mapping and Localisation classes.

The *RobotServiceController* class *realises* the *MapUpdateInterface* interface, meaning that it contains a function called *updateLocalCopy*. This is called by the map building services the class has registered with as a callback to update the local copy of the map. This is necessary, as in a distributed environment the network overhead of having to request the entire map to be transmitted every 100ms for real-time visual feedback would be too great. Instead, a local copy of the map is kept, and this is updated incrementally by the service that is building the map using the *updateLocalCopy* callback function. Of course, it is only necessary to keep a local copy of the map if real-time visual feedback is required – otherwise this is an unnecessary overhead, and no incremental updates are carried out.

```
class RobotServiceController:public ArAction, public SfArtifact,
                             public MapUpdateInterface
{
   public:
    SFEXPORT RobotServiceController();
    SFEXPORT virtual ~RobotServiceController();

    SFEXPORT virtual ArActionDesired *fire(ArActionDesired currentDesired);
    static RobotServiceController *invoke();
    SFEXPORT void draw(SfWin *w);

    void updateLocalCopy(double value, long x, long y,  string systemName);

  private:
    ArSerialConnection myConnection;
    ArActionDesired myDesired;
    int squareSize;
    bool socketConnected;
    long maxX[50], maxY[50], minX[50], minY[50];
    SensorSet<MapUpdateInterface> mySensors;
    bool sonarsOn;
    CommandSet commands;
    ArThread commandServe;

    ME85* me85;
    ME85mod* me85mod;
    ME88* me88;
    ME88mod* me88mod;
    K97* k97;
    K97mod* k97mod;

    RecordTestRun *recordTestRun;
    SpecularEstimator *specEst;

    GridMap<double>* me85Map;
    GridMap<double>* me85modMap;
    GridMap<double>* me88Map;
    GridMap<double>* me88modMap;
    GridMap<double>* k97Map;
    GridMap<double>* k97modMap;
    GridMap<double>* viewAMap;
    GridMap<double>* recordTestRunMap;

    int me85ClientNo, me85modClientNo,me88ClientNo, me88modClientNo;
    int k97ClientNo, k97modClientNo, recordTestRunClientNo;

    void drawMap(SfWin *w);
    void toggleSonars(){sonarsOn = !sonarsOn;};
    void commandServer();
};
```

Fig 4.11 RobotServiceController class

4.8.1 Interface To Saphira

Saphira interfaces with the class through four functions:

- *sfLoadInit*(), an external function called by Saphira that invokes the class and links it to the Saphira application. It is similar to the function *main* in a simple C program.
- *fire*(), which is called by Saphira once every clock cycle, usually every 100ms though this is adjustable. This is used to execute all of the real-time operations of the system, with the exception of the graphical output.
- *invoke*(), a static function which returns a *new* instance of the class. This is called by *sfLoadInit*() when the dynamic library is loaded into Saphira.

- *draw*(), which is called once every clock cycle to draw to the Saphira window.

The first two functions must be present in every coherent unit of functionality in the system. The *draw*() function must be present if the behaviour is required to draw objects on the Saphira window. The *invoke*() function can take any name, but there must be a static function in the each object registered with Saphira that creates an instance of that object. It is merely convention to call that function *invoke*.

4.9 ServiceControl Class

This class standardises the registration procedure for all services in the robotics framework. To this end, the *ServiceControl* class contains a function *registerClient*, which a client calls to register with the service and to receive a client number which must be used whenever sending sonar readings to the service. This function also requires the client to tell the service about the configuration of its robot, the positions of the sensors, the size of the robot, the width of the sonar beam, as well as providing a reference back to the client in order for the service to perform a callback to the clients. In the case of the mapping services the callback is used to update the clients local copy of the map.

Once the client has registered with the service, it continually sends data to the service. This data can take two forms:

- Robot pose information and sonar range readings can be sent to the service using the *pushSonar* function call.
- Robot pose information, with no sonar range information, can be sent to the service using the *pushPose* function call. This capability was provided for services that do not require sensor range information, such as recording the path a robot took, as well as services created by others in the UL Robotics Group that perform pursuit and evasion strategies.

The service takes the data from the client and places it in a queue. A separate function, *processQueue*(), must be defined by the child class to pop the client information off the queues and process it in whatever way the specialisation (e.g. *ME85* or *ME88mod*) of the class requires. For example, the *ME85* class simply takes the reading and passes it to the *updateMap* function, but the *ME85mod* class will pass

the readings to the *SpecularEstimator* class first to ascertain its confidence in the accuracy of the sonar's readings. The *processQueue* function can either run in a separate thread, in parallel with the client for non real-time applications such as map building, or can be called every time another reading is pushed onto the queue for real time applications such as *SpecularEstimator*. *ServiceControl* handles all the threading issues, such as creating and destroying threads, as well as mutually exclusive access to shared data.

As service must define its own particular method of dealing with the sensor readings, and therefore the *processQueue* is a pure virtual function (i.e. it has no definition in the class *ServiceControl*), making *ServiceControl* and abstract base class. This means that it is impossible to declare an instance of the class *ServiceControl* without another class inheriting from it, so it cannot be used in isolation from *ME85*, *K97* etc.

```
template <class T>
class ServiceControl
{
  public:
    virtual ~ServiceControl();
    virtual int registerClient(T* functionPointer, int noOfSonars,
        SosPose* sonarPositions, double beamWidth, long robotRadius);
    bool pushSonar(SosPose robotPose,long* sonarRanges,int clientNumber,
                   double anythingElse = 1);
    bool pushPose(SosPose robotPose, int clientNumber);
    bool unRegisterClient(int clientNumber);

  protected:
    int maxClients;
    vector<SensorSet<T> > clientProperties;
    const bool isMultiThreaded;

    bool clientNumbers[MAX_CLIENTS];
    int numOfClients;
    char systemName[50];
    SosThread* processReadingsQueue;
    SosFunctorC<ServiceControl> *queueProcessor;

    ServiceControl(bool childIsMultiThreaded = true);
    virtual void processQueue() = 0;
    SosPose transformSonar(SosPose robotPose,int sonarNum,
                           int clientNumber);
    bool popSonarReadingsQueue(SensorReadingSet&);
    bool popPoseQueue(SensorReadingSet&);

 private:
    queue<SensorReadingSet> sonarReadingsQueue;
    queue<SensorReadingSet> poseQueue;
    SosMutex *sonarQueueMutex;
    SosMutex *poseQueueMutex;
};
```

Fig 4.12 ServiceControl class.

4.10 Mapping Class

As can be seen in Fig 4.11, the *Mapping* class is a superclass of each of the six mapping systems tested during this research. It contains the operations and attributes common to all of the map-building systems. These include some mathematical functions, such as *getDist*, and *getDiffInAngle*, as well as other operations common in the map building domain, such as *transformSonar*, *getSonarBoundingBox*, *isInArc* and *isInRobot*.

The class also provides some public functions for use by the client, such as the ability to save and load the map, to add a line to the map, as well as to import a map in the format used by Saphira – storing maps as a series of points, with each point representing the end of a wall.

```
class Mapping : public ServiceControl<MapUpdateInterface>
{
  public:
    Mapping();
    virtual ~Mapping();

    double getPointVal(long x, long y);
    virtual bool importPointMap(string fileName);
    virtual void addLine(long x1, long y1, long x2, long y2);
    virtual void blurMap(int = 5);
    virtual bool save(string filePath);
    virtual bool load(string filePath);
    virtual void reset();

  protected:
    long squareSize;
    GridMap<double>* myMap;
    long maxX[MAX_SONARS], maxY[MAX_SONARS], minX[MAX_SONARS], minY[MAX_SONARS];
    bool sonarsOn;

    virtual void updateMap(SensorReadingSet readings) = 0;
    bool isInArc(double x, double y, SosPose sonarPose, double range, int clientNumber);
    bool isInRobot(double x, double y, SosPose robotPose, int clientNumber);
    void getSonarBoundingBox(SosPose sonar, int range, long &xMax, long &yMax,
                             long &xMin, long &Ymin, int clientNumber);
    long getDist(long x1, long y1, long  x2, long y2);
    double getDist(double x1, double y1, double  x2, double y2);
    double getDiffInAngle(SosPose sonar, double x, double y);
    void updateMapValue(double value, long x, long y);
};
```

Fig 4.13 Mapping class, parent of all the map-building system classes.

4.11 ME85 map-building system

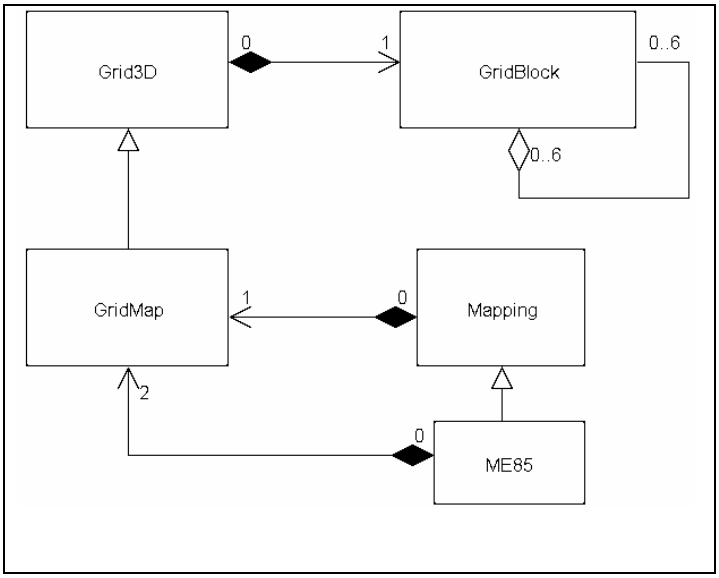


Fig 4.14 The ME85 class integrating with the GridMap, Grid3D, Mapping and GridBlock classes.

The *ME85* class is a map building system written for this thesis that accepts sonar readings from a client, models the sonar sensors, then integrates the readings with previous readings to generate a map of the region that the clients robot is exploring. It uses the logic and equations presented in Moravec and Elfes 1985 paper, *High Resolution Maps From Wide Angle Sonar* [46].

As mentioned in Chapter 2, the mapping theory put forward by Moravec and Elfes advises storing the probability of occupancy or emptiness as a number in the range [-1,1]. They also advise maintaining three maps, one, *Occ*, for the occupancy probability in the range of [0,1], another, *Emp*, for the empty probability in the range of [0,1], and one, *Main*, which is built from the other two maps as follows:
If, for a particular point (x,y):

- Occ(x,y) >= Emp(x,y), then our main map, Main(x,y) = Occ(x,y).
- Emp(x,y) > Occ(x,y), then Main(x,y) = Emp(x,y) * -1.

This means that if it is more likely that a cell is occupied than unoccupied, it is marked as occupied. Conversely, if the probability that the space is empty is greater than the probability of it being occupied, the main map is updated with the empty value multiplied by minus one. The map therefore contains numbers in the range [-1, 1]. This indicates that as the value of Main(x,y) approaches 1, the greater the chance that that point is occupied. As it approaches -1, the greater the chance that the point is unoccupied. If the value is around zero, then little or nor information exists to distinguish the cell as either occupied or unoccupied.

```
class ME85 : public Mapping
{
  public:
    ME85(int sizeOfSquare=100, char* mapToImport = 0);
    ~ME85();

    void blurMap(int kernelSize);
    bool importPointMap(string mapToImport);
    void addLine(long x1, long y1, long x2, long y2);
    bool load(string filePath);
    virtual void reset();

  private:
    GridMap<double>* occMap;
    GridMap<double>* emptyMap;

    void updateMap(SensorReadingSet readings);
    void processQueue();
};
```

Fig 4.15 ME85 class.

4.11.1 Updating The Map

When it comes to adding information to the map from a set of sonar readings, the first step is to decide which points might possibly need updating. The most obvious, and most inefficient, way of doing this is to check each point in the map to see if it is within the arc of a given sonar with the *isInArc*() function, and updating its value if it is.

Unfortunately, in a complete map of n*n points, this leads to an $O(n^2)$ running time for each sonar. A much quicker way to do this is to model the space covered by the sonar with a bounding box around the arc, giving a minimum and maximum X and Y in which to search. This box encompasses all points that are inside the arc, and some that are outside, as seen in Fig 4.16. This brings the running time down to $O((maxRange*2)^2)$, where maxRange is the maximum range of the sonars. For example, if the maximum range of a sonar is set at 2.5 metres, and each cell is 10mm by 10mm in size, then the maximum number of cells that must be operated on for that sonar is 25 * 25 = 625. This is considerably smaller than $O(n^2)$, and is also very pessimistic, as the

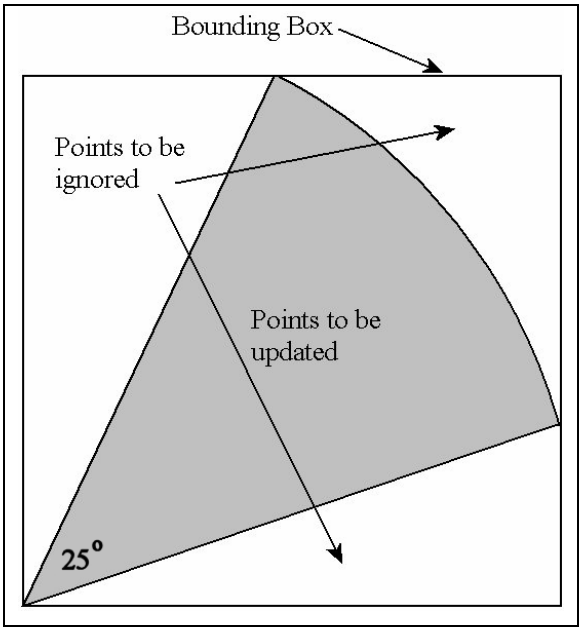


Fig 4.16 Bounding box around a sonar arc, representing a fragment of the real map.

running time does not approach this upper bound on a Pioneer 1 robot, even when in a completely empty space.

With the bounding boxes in place, the *updateMap* function iterates through the points inside the box, taking steps of 100mm at a time. This means that the map is made up of squares, each 100mm to a side. If a point is in the unoccupied area of the map, that is the area between 0 plus the error and the range reading minus the error (Fig 4.17), then Moravec and Elfes advise to update the Empty map with the simple formula:

$$Emp(X,Y) = Emp(X,y) + Emp_k(X,Y) - Emp(X,Y) * Emp_k(X,Y)$$

In the above formula, the value of the new $Emp_k(X,Y)$ is inversely proportional to the distance from the central beam and the distance from the sonar. This gives a gaussian spread from the point of origin. The formulas to derive it can be found in Chapter 2.

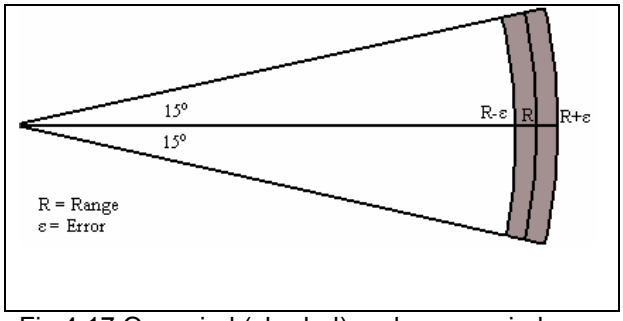


Fig 4.17 Occupied (shaded) and unoccupied areas of a sonar arc.

If a point is inside the occupied area of the arc, i.e. the area covered between the range reading taken less the expected error and the range reading plus the error (Fig 4.15), the update is a little more complicated.

First we must *Cancel* the occupied reading:

$$Occ_k(X,Y) = Occ_k(X,Y) * (1 - Emp(X,Y))$$

to account for the fact that previous readings disagree with this one. This means that if previous readings said that this space was empty, then we are less inclined to believe readings that say it is occupied. Next we *Normalise* the occupied value to sum to one, as we are making the assumption that there is a single target in the occupied area (which Konolige[15] later improved upon with his MURIEL method). Finally, we *Enhance* the prior occupancy value with the new value, as in the Empty map.

$$Occ(X,Y) = Occ(X,Y) + Occ_k(X,Y) - Occ(X,Y) * Occ_k(X,Y)$$

4.12 ME85mod Map-Building System

The *ME85mod* class is similar to the *ME85* class in that it is a child of the *Mapping* class and updates its map using the theory laid down by Moravec and Elfes, but it also contains certain modifications not mentioned in Moravec and Elfes paper. These issues with the original theory and the solutions applied to them and are explained below.

4.13.1 Problem 1 – The *Cancel* Step Used In Updating the Map Is Biased Towards Freespace Readings

A major problem with the approach taken by Moravec and Elfes in [46] was their assumption that the sensors were perfect, or at least that if there was an obstacle in the beam of the sonar that it would reflect diffusely back to the sensor. As stated in Chapter 2, this is certainly not the case, as we often see specular reflection playing havoc with the readings. This assumption led to the creation of a map update strategy that did not allow for the fact that a freespace reading could be erroneous, and therefore if a sonar sensed an obstacle, before this information was integrated into the map the degree to which the map would be updated is reduced by the probability that the cell is unoccupied.

Unless the obstacle is either rough in texture or close to perpendicular to the sensor, the sonar beam either rarely echoes back to the source, or it comes back after multiple reflections, both of which are completely useless and contain no information about the relative location of the obstacle (see Fig 2.9). Even worse, it contains misinformation, claiming the space to be empty. Because of this misinformation, the update formula for the Empty map is inadequate. The function is monotonically increasing i.e. it never decreases, and the value it generates is later used to decrease the certainty of the Occupied cell, in the *Cancel*ling step.

```
class ME85mod : public Mapping
{
  public:
    ME85mod(int sizeOfSquare=100, char* mapToImport = 0);
    ~ME85mod();
    virtual int registerClient(MapUpdateInterface* functionPointer, int noOfSonars,
              SosPose* sonarPositions, double beamWidth, long robotRadius);

    void blurMap(int kernelSize);
    bool importPointMap(string mapToImport);
    void addLine(long x1, long y1, long x2, long y2);
    bool load(string filePath);
    virtual void reset();

  private:
    GridMap<double>* occMap;
    GridMap<double>* emptyMap;

    SpecularEstimator **specEst;
    PoseBucket *poseBucketOcc;
    PoseBucket *poseBucketEmp;

    void updateMap(SensorReadingSet readings);
    void processQueue();
};
```

Fig 4.18 The ME85mod class

Once enough incorrect readings have been received, no amount of correct readings can reverse the error, for as Emp(X,Y) approaches 1, Occ(X,Y) is multiplied by (1 – Emp(X,Y)), which approaches zero. This is not a symmetrical update procedure, which the authors explain by stating that the occupied area represents a single occupied cell, while the unoccupied area represents each cell being unoccupied, as any occupied cell in that area would have caused a reflection. This only explains the *Normalis*ation step, however, and not the *Cancel*ling, which favours a cell being empty rather than occupied. If anything, the balance should be in the other direction, as it is usually better to err on the side of caution than to not recognise an obstacle in the robots path.

To this end, the *Cancel*ing step was removed from the Occupied cell update procedure, and a modified version of Konolige's Dynamic Mixture Model was used. This is fully explained in section 4.17. Essentially this means that if just one cell in the freespace part of the sonar beam has a high probability of occupancy, then the range reading is assumed to be noisy, and the effect of the reading on the map is significantly weakened.

4.13.2 Problem 2 – Specular Readings Often Cannot Be Detected Directly From Historical Data

The above alteration to the basic map building algorithm has a weakness in that it relies on receiving enough correct readings to cancel out the bad readings. Unfortunately this rarely happens. Take for example a robot travelling straight down a corridor. The sonars along it's front do not detect the walls to either side of due to the angle between them. The side sonars should detect the wall correctly. However there will be many more specular readings than correct readings since as the robot approaches any particular position in the wall it will accumulate many incorrect readings for its corresponding cell from its front sonars, and only receives relatively few correct readings from its side sonars as it passes the cell. The resulting probability of occupancy of the cell will be low.

Konolige went some way towards solving this problem with his pose buckets. Using pose buckets prevents multiple readings from similar poses affecting the same cell. Pose buckets were therefore used with *ME85mod* and did improve the resulting map somewhat. However experiments showed that pose buckets on their own were not enough since there are many more positions the robot can be in in relation to a wall that will yield specular readings than positions that yield correct readings. It would be better if the specular readings could be identified as soon as they were received and thus ignored. Konolige claims his MURIEL method can do this, but it is flawed in that it needs good readings at a cell *before* it can identify new readings as being specular. As it is often the case, as in the corridor example above, that many incorrect readings are received before any good readings, most of the specular readings are still incorporated into the map.

This chicken-and-egg problem means that in order to compensate for specular readings without prohibitively expensive recomputation of previous readings it is necessary to make predictions regarding features in the environment that the robot has not sensed *yet*. To this end the *SpecularEstimator* service is used with *ME85mod* to create a local map around the robot built from features it predicts from the sensor readings using the Feature Prediction algorithm described in Chapter 3. It then uses this map to detect specular readings. The *SpecularEstimator* module is discussed in greater depth later in the chapter. Due to the fact that the *SpecularEstimator* only

stores a local map of the area surrounding the robot, when the *ME85mod* is being used by multiple robots simultaneously it will need a separate *SpecularEstimator* for each robot. To this end it redefines the *registerClient* function from the *ServiceControl* class to ensure than whenever a client registers with this class, it automatically registers with a new *SpecularEstimator* class also.

4.14 ME88 Map-Building System

The *ME88* class is very similar to the *ME85* class. It is based on the theories put forward in Matthies and Elfes 1988 paper [42]. It is a child of the *Mapping* class, and has all of the same functions as *ME85*. The chief way in which it differs is in how it integrates new sonar readings with prior information. Rather than using the formulas

$$P(EMP|R) = P(EMP) + P(R|EMP) - P(EMP) * P(R|EMP) \text{ and}$$

$$P(OCC|R) = P(OCC) + P(R|OCC) - P(OCC) * P(R|OCC)$$

from *ME85*, a new Bayesian update formula is used.

$$P(OCC \mid R) = \frac{P(R \mid OCC)P(OCC)}{P(R \mid OCC)P(OCC) + (1 - P(R \mid OCC))(1 - P(OCC))}$$

Here, $P(OCC)$ and $P(EMP)$ are the prior probabilities of the square being occupied and empty respectively. $P(R|OCC)$ and $P(R|EMP)$ are the probabilities of getting the range reading R given our prior belief that it is occupied and empty respectively. This is our model of the sonar, using the gaussian noise model described in section 2.8.1. $P(OCC|R)$ is the posterior probability, that is the probability of the square being occupied *after* the range reading has been integrated into the model. See section 2.8.2 for more information on this.

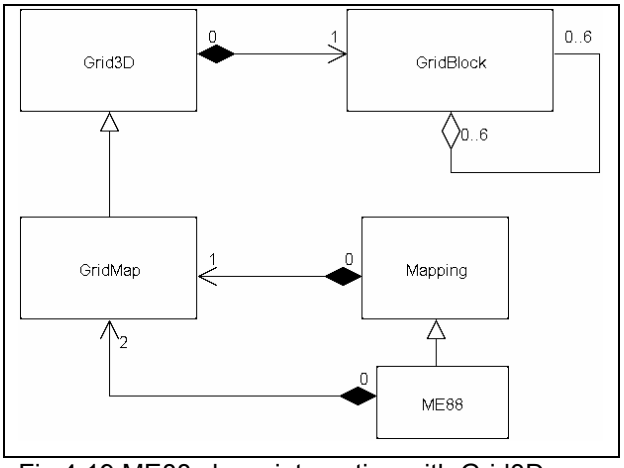


Fig 4.19 ME88 class, interacting with Grid3D, GridMap, Mapping and GridBlock classes.

Whereas in *ME85* three maps must be maintained, a map storing the occupancy values of the cells, a map for the freespace values and a map combining the other two maps, in *ME88* only a single map needs to be stored. This stems from the fact that in the formula presented above, Emp(X,Y) = 1 –

Occ(X,Y). The values in this map have a range of [0,1], with 0.5 representing *unknown*, i.e. if a square has a value of 0.5, we don't know if it is occupied or not. If it has a value of 0, it is empty, and if it is 1 it is occupied.

```
class ME88 : public Mapping
{
  public:
    ME88(int sizeOfSquare=100, char* mapToImport = 0);
    ~ME88();

  private:
    void updateMap(SensorReadingSet readings);
    void processQueue();
};
```

Fig 4.20*ME88* class definition. The vast majority of processing takes place in the *updateMap* function, which integrates a set of sonar readings with the map being generated.

The *ME88* class also integrates positional uncertainty into its model. It takes a measure of how certain it is of its current position and orientation, between zero and one, and integrates it with $P(R|OCC)$ using the Bayesian update formula. It receives this measure from the client, who may also interface with a localisation service and will then be able to receive a value representing the probability that the robot is in any given pose, which it then passes on to the *ME88* module.

4.15 ME88mod Map-Building System

The *ME88mod* mapping service is similar to *ME88* in many ways. It contains just a single map, uses the same Bayesian update rule and integrates positional uncertainty into the model. In addition it uses the *SpecularEstimator* class to identify specular readings, as well as the *PoseBucket* class to remove redundant readings, as with *ME85mod*.

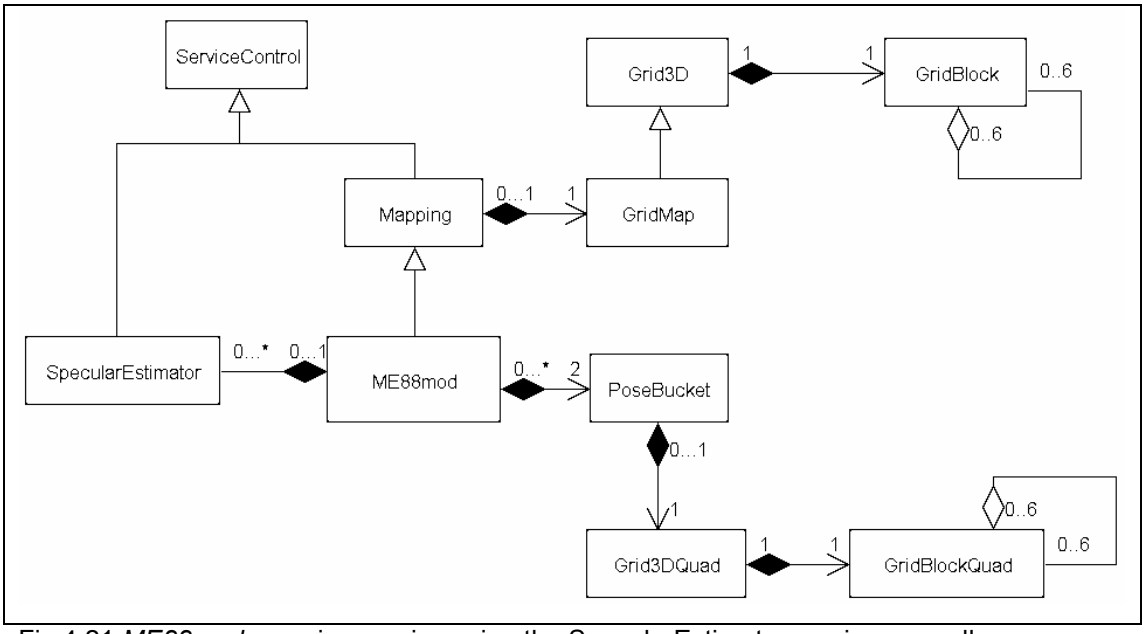


Fig 4.21 *ME88mod* mapping service using the SpecularEstimator service, as well as *PoseBucket* and *GridMap* objects

ME88mod also employs the same modified version of Konolige's Dynamic Mixture Model as *ME85mod* and *K97mod*. This is fully explained in section 4.17.

```
class ME88mod : public Mapping
{
  public:
    ME88mod(int sizeOfSquare=100, char* mapToImport = 0);
    virtual int registerClient(MapUpdateInterface* fnPointer,int noOfSonars,
                SosPose* sonarPositions, double beamWidth, long robotRadius);
    virtual void reset();
    ~ME88mod();
  private:
    PoseBucket *poseBucketOcc;
    PoseBucket *poseBucketEmp;
    SpecularEstimator **specEst;

    void updateMap(SensorReadingSet readings);
    void processQueue();
};
```

Fig 4.22 *ME88mod* class definition. It contains two sets of pose buckets, one for freespace readings, one for surface readings. The object of type SpecularEstimator, *specEst*, performs feature prediction. The *registerClient* function inherited from the *ServiceControl* abstract base class via the *Mapping* parent class is redefined here. This is to ensure that each time a new client registers with the service, they are automatically registered with an object of type *SpecularEstimator,* as each client must have their own object for Feature Prediction.

4.16 K97 Map-Building System

The K97 mapping service uses the map building theory from Konolige's 1997 paper [33] Improved Occupancy Grids for Map Building. Similar to the *ME85* service, it uses three grids to store the map. One, *occMap*, stores the values for each cell which a sonar reading claimed to be occupied. Another map, *emptyMap*, stores the values for each cell that sonar readings claim to be empty. These two maps are combined and the result is placed in a third map, *myMap*, which is inherited from the *Mapping* class. The formulas and theory behind the methods used in the *updateMap* function were previously explained in chapter two.

```
class K97 : public Mapping
{
  public:
    K97(int sizeOfSquare=100);
    ~K97();

    void blurMap(int kernelSize);
    bool load(string filePath);
    virtual void reset();

  private:
    GridMap<double>* occMap;
    GridMap<double>* emptyMap;

    PoseBucket *poseBucketOcc;
    PoseBucket *poseBucketEmp;

    void updateMap(SensorReadingSet readings);
    double calcNormalDistribution(double x,
                double mean,double standDev);
    void processQueue();
};
```

Fig 4.23 K97 mapping service class definition.

As explained in chapter two, the application of Konolige's sonar model is based on statistical normal distribution, which is stored in a table, and can be used to calculate the area beneath a normal curve in a two dimensional graph. The *calcNormalDistribution* function is used for this purpose.

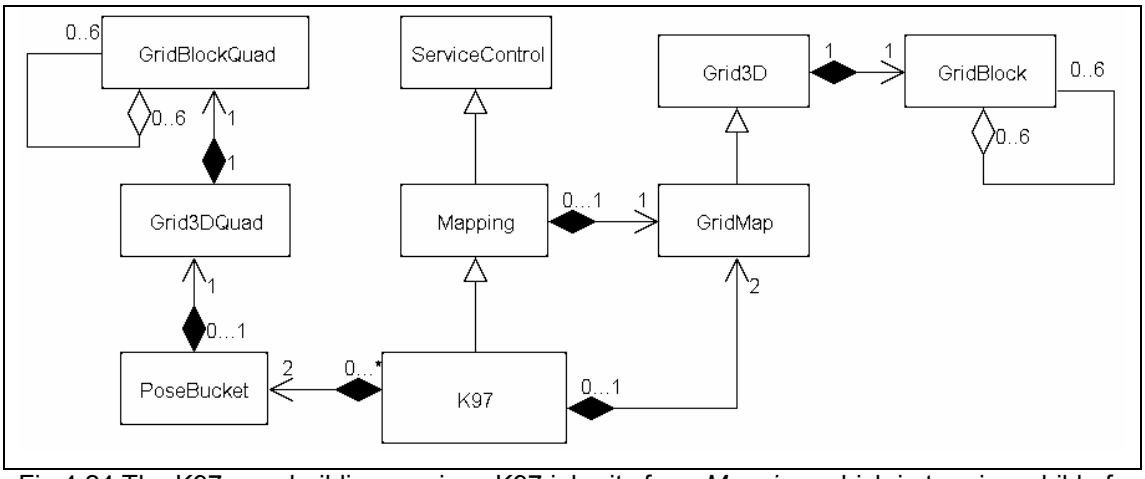


Fig 4.24 The K97 map building service. K97 inherits from *Mapping*, which in turn is a child of the *ServiceControl* class. K97 contains three *GridMap*s, one inherited from *Mapping* and two of its own. It also contains two *PoseBucket*s, one for freespace readings, one for occupied readings.

K97 contains two pose buckets, *poseBucketOcc* and *poseBucketEmp*. The former ensures that each cell can only be updated as occupied once from any given pose of

the robot, the latter ensures that each cell can only be updated as unoccupied once from a particular robot pose. Using these pose buckets prevents multiple incorrect readings accumulating at a cell when the robot receives many erroneous specular reading from similar poses. This obviously is advantageous, and leads to better maps being generated. Unfortunately, while multiple incorrect readings are prevented, one incorrect specular reading from each position is still incorporated into the map. It would be better if these noisy readings could be identified and discarded.

Konolige's *dynamic mixture model* is a method for doing just that. It uses previously generated map information to estimate the specularity of new sonar readings. As discussed in chapter two, this has the drawback that correct sonar readings must be received before noisy readings can be identified, and if noisy readings are received first then this method does not work very well. However, when correct readings are received first, this works quite well, and overall helps create better maps than if it were not used.

While *K97* contains some variable constants which can be tuned for different environments, the K97 mapping service is intended to be as close to Konolige's original system as possible, and as such all of his settings are used where possible. The value λ_s is set to 1.5, meaning that once the occupied value of a cell reaches 1.5 it is completely believed to be occupied, and all readings that later report the cell to be empty are believed to be 100% noise. Each *PoseBucket* allows a cell to be updated from a particular angle three times, and from three different distances. [33] never specifies what value F - the probability of an object existing in the sonar beam at a distance other that the range measured - takes, merely referring to it as a 'small constant', so a value of 0.05 was assigned to it.

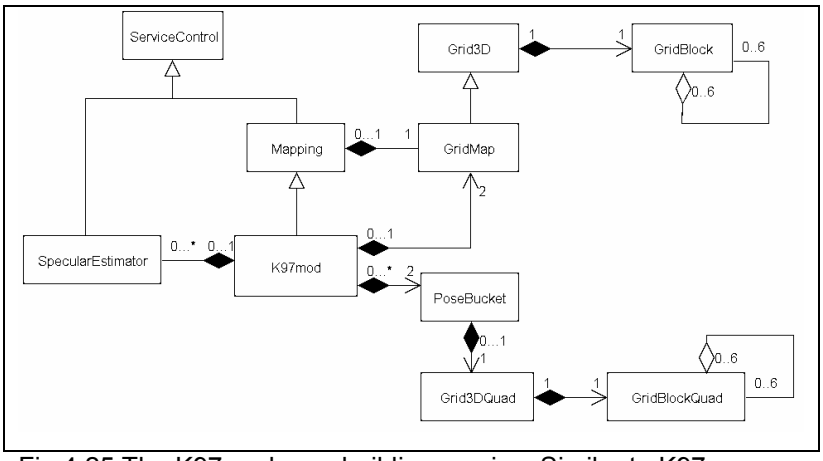


Fig 4.25 The K97mod map building service. Similar to K97 service, but the addition of the *SpecularEstimator* service.

4.17 K97mod Map Building System

The K97mod map building service is similar in many respects to K97, insofar as it uses the sonar model described in [33], uses three *GridMap*s, two *PoseBucket*s, and combines the occupied and freespace maps using the formulae specified by Konolige. However, it also alters the basic algorithm, as well as using the *SpecularEstimator* class to better identify and discard noisy specular readings.

4.17.1 Problem 1 – Probability of Specularity For A Single Sonar Reading is Independent From Cell To Cell

One of the criticisms Konolige made of his MURIEL method's dynamic mixture model was that the probability of specularity, $P(S)$, was computed for each cell based only on the surface readings *at that cell*. This means that for any one sonar scan, some cells may give a high $P(S)$, while others, with fewer surface readings, may give a low $P(S)$. It can be argued that once a single cell claims the reading to be specular, no other cell should be able to claim that it is a correct reading – all that is to be decided is the probability of specularity of the beam, and this value, at the very least, should be applied to all cells in the beam. To this end, an initial conservative approach was taken so that the highest value of $P(S)$ of all the cells in the freespace part of the beam is applied to all cells in the beam.

```
class K97mod : public Mapping
{
  public:
    K97mod(int sizeOfSquare=100);
    ~K97mod();

    virtual int registerClient(MapUpdateInterface* functionPointer,
                               int noOfSonars, SosPose* sonarPositions,
                               double beamWidth, long robotRadius);
    void blurMap(int kernelSize);
    bool load(string filePath);
    virtual void reset();
  private:
    GridMap<double>* occMap;
    GridMap<double>* emptyMap;

    PoseBucket *poseBucketOcc;
    PoseBucket *poseBucketEmp;
    SpecularEstimator **specEst;

    void updateMap(SensorReadingSet readings);
    double calcNormalDistribution(double x, double mean, double standDev);
    void processQueue();

};
```

Fig 4.26 *K97mod* mapping service class. The *registerClient* function from the *ServiceControl* class is redefined so that every new client of the class is automatically registered with a *SpecularEstimator* class at the same time. As the *SpecularEstimator* class maintains a local map for the robot, there is a separate one for each client of the *K97mod* mapping service.

When this method of calculating $P(S)$ was applied and tested it performed considerably better than the original method. However it was observed that many readings from the front sonar, which seemed to be correct, were discarded when the robot was travelling parallel to a wall, for example when performing wall following in a corridor. This issue came from the fact that, since the $P(S)$ from one cell is applied to all cells in the beam, the possibility exists that extreme parts of the sonar beam may not give a correct return range reading from an occupied cell. This could occur for a number of reasons. One possibility is that the object is smooth and the beam reflected off at an angle such that it didn't return to the sensor. Another possibility is that the strength of the beam is so weak at that distance that it reflected properly but dissipated before returning to the sensor. A third possibility is that the cell is not really occupied and some previous readings were inaccurate or interpreted incorrectly. For whatever reason the modelled sonar beam is inconsistent with previous surface readings, giving equal weight to the $P(S)$ of cells far from the sonar emitter and to the $P(S)$ of cells near the robot causes many readings to be given overly-weak freespace updates.

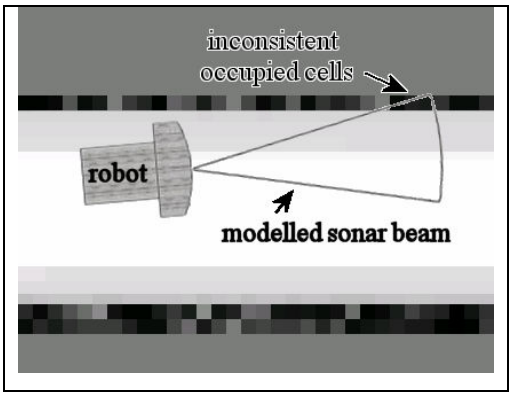


Fig 4.27 (a) Front sonar of a robot which is slightly inconsistent with the previously modelled map, with only an extreme edge of the beam overlapping an highly occupied cell.

For example, in Fig 4.27(a) the extreme edge of the sonar beam passes over a cell with very high occupancy, giving it a probability of specularity $P(S) \approx 1$. If this probability is applied to the whole beam, the reading will more or less be discarded. Robots exhibiting a wall-following behaviour would discard many readings from their front sonars despite the fact that they are more or less correct. It can therefore be argued that the beam still contains information, since the majority of the beam is correct, there are no obstacles in

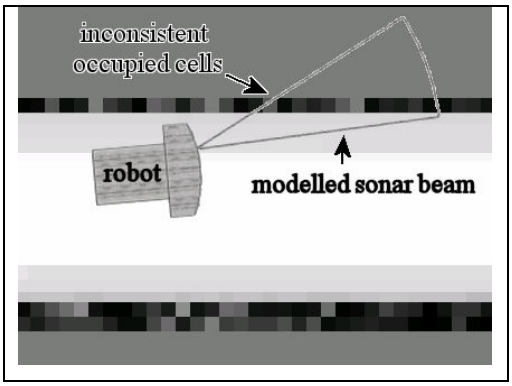


Fig 4.27 (b) Diagonal sonar of a robot which is very inconsistent with the previously modelled map, with an occupied cell in the freespace part of the beam, very close to the sonar emitter.

most of the freespace area. Compare this to Fig 4.27(b), where it would be better to strongly reduce the strength of the map update since the inconsistent high surface-value cells are much closer to the sonar emitter and therefore affect a much larger area of the beam.

For these reasons, when choosing the largest $P(S)$ of all the cells in the beam, the $P(S)$ of each cell is weighted by their distance from the sonar emitter and their angle from the central beam. This is done using Konolige's formula for the sonar model $p_{2m}\left(r = D|C_i\right)$ - see chapter two for explanation of this formula. This gives the weighted probability of specularity for each cell:

$$WP(S) = P(S) * p_{2m}\left(r = D|C_i\right)$$

The largest $WP(S)$, $WP(S)^*$ is applied to every cell in the beam that has a $P(S) < WP(S)^*$, otherwise if the probability of specularity for that cell is greater than $WP(S)^*$ it remains unchanged.

$$P(S)' = max\left(P(S), WP(S)^*\right)$$

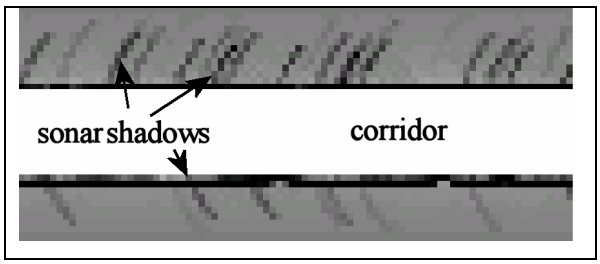


Fig 4.28 A typical corridor scene with '*sonar shadows*' resulting from noisy specular readings

4.17.2 Problem 2 – The Probability of Specularity Only Diminishes The Strength Of The Freespace Update, Not The Occupied Update

A second issue with the original application of the probability of specularity in [33] was that it was only used to reduce the effect of the freespace update. The update of the occupied cells was left unaffected, leading to 'sonar shadows' behind obstacles, as in Fig 4.28. If a reading is specular, then its effect on the map should be reduced to a degree based on its probability of specularity, and this should be applied to both the freespace and occupied regions.

The original method does not do this, as it uses the formula when merging the freespace and surface maps to create the overall map:

$$\log \lambda_T = \log \lambda_s + \log\left(\lambda_f\left(1 - P(S)\right) + P(S)\right)$$

where only the freespace update λ_f is weakened by the probability of the range reading being specular $P(S)$, while the occupied, or surface, update is incorporated into the main map at full strength, regardless of the value of $P(S)$. The formula in K97mod is changed to

$$\log \lambda_T = \log\left(\lambda_s (1 - P(S)) + P(S)\right) + \log\left(\lambda_f (1 - P(S)) + P(S)\right)$$

in order to apply the same measures to the surface update as to the freespace update. This has lead to a marked decrease in the number of '*sonar shadows*' behind walls, see Fig 4.28.

4.18 SpecularEstimator Service For Feature Prediction

The SpecularEstimator class is used by the *ME85mod*, *ME88mod* and *K97mod* mapping services to estimate the confidence that should be attributed to the sonar's range readings. It does this using the Feature Prediction methods laid out in chapter three.

It is also possible for the client to add permanent features to the map if they are somehow certain that they exist using the *putWall* function, for example if a wall has been detected by some other sensor like a touch sensor or a laser.

```
class SpecularEstimator : public ServiceControl<int>
{
  public:
    SpecularEstimator();
    ~SpecularEstimator();

    double getSonarConfidence(int sonarNum);
    int getNumberOfWalls();
    WallObj getWall(unsigned int wallNumber);
    void putWall(WallObj wallToAdd);
    void reset();

  private:
    double *sonarConfidences;
    vector<WallObj> prevWalls;
    vector<WallObj> currentWalls;

    virtual void processQueue();
    double getDist(double x1, double y1, double x2, double y2);
    double combineConfidences(double confidence1, double confidence2);
    int correspond(WallObj, WallObj);
};
```

Fig 4.29 SpecularEstimator class.

4.19 PlanPath Class

The *PlanPath* class creates a path from one point to another on a static map using a modified A-Star algorithm. The variant on the A-Star algorithm used in this class is explained in depth in Chapter 5, and will therefore not be covered here. Suffice to say, this class can plan a path quickly and to near-optimality in a static environment.

It is very robust at avoiding both local and global minima, unlike the standard A-Star algorithm. The robot can be made keep a certain safe distance from obstacles using the *setRobotRadius* function. It divides a path into a discrete set of waypoints, with a straight line of sight between each waypoint and the one before and after it. Calling the generatePath function, passing the start and endpoints in Cartesian format as parameters, generates a path. Waypoints can be retrieved using the *getSubGoal* function.

```
class PlanPath
{
  public:
    PlanPath(long lookahead = 200);
    ~PlanPath();

    bool setMap(char* filepath, long sqSize = 100);
    bool setMap(GridMap<double>*, long sqSize = 100);
    bool generatePath(long startX,long startY,long goalX,long goalY);
    int getNumSubGoals();
    Node* getSubGoal(int goalNum);
    void setRobotRadius(unsigned long radius);
    Node* createPath(GridMap<double> *myMap,long startX,long startY,
                     long goalX,long goalY, long& pathLength);


  private:
    GridMap<double>* myMap;
    GridMap<double> *markedSquares;
    long squareSize;
    long horizon;
    long robotRadius;
    int numGoals;
    Node* subGoals;

    double canDrawLine(GridMap<double>* myMap, long x1, long y1,
                       long x2,long y2, double threshold = 0.49999);
    double calcFvalue(double gVal,double hVal,double iVal,double origDist);
    double getDistance(long, long, long, long);
};
```

Fig 4.30 The PlanPath Class implements a modified version of the A-Star path planning algorithm. Line fitting methods are used to reduce the number of local minima, while an exploration strategy reduces the frequency and severity of global minima.

Chapter 5: Benchmarking – What makes a good map?

5.1 Introduction

In order to accurately gauge the effectiveness of a map building technique, a comprehensive analysis of the maps produced by it must be undertaken. In this chapter, a variety of such benchmarking methods are presented, each of which has been applied to the maps generated using the software systems described in Chapter 4, with the results presented in Chapter 6. These benchmarking methods include:

- An image comparison algorithm (Image Correlation [1])
- A direct comparison method called Map Scoring specifically designed for probabilistic maps [39].
- A modified version of the Map Scoring method that only tests the correctness of the obstacles in the map, ignoring the freespace areas
- A benchmarking suite designed as part of this thesis called Path Comparison, which tests the usefulness of a map as a means of navigation rather than treating it as if it were a picture. This bears some resemblance to a benchmarking technique developed by Thrun [58], which uses Voronoi graphs to convert a grid-based metric map into a topological map in order to compare the cost of paths generated in grid maps with the cost of paths generated in topological maps. The approach taken here is more focused on examining the freespace and occupied regions of a map than the cost of the paths, and therefore differs considerably from Thrun's approach.

A paper detailing the benchmarking techniques presented in this chapter [14] entitled "Developing an extensible benchmarking framework for map building paradigms" has been accepted for publishing in the Ninth International Symposium on Artificial Life and Robotics (AROB) 2004 in Oita, Japan.

5.2 Traditional Approaches to Evaluating Map Fitness

Two dimensional maps are, by their nature, relatively simple to visualise, and are intuitively easy for the human eye to examine and evaluate. For this reason, many researchers have felt it sufficient to present qualitative results only [15, 33, 46, 59],

often with little or no quantitive analysis being carried out. For example, in Fig 5.1, given the ideal map of an environment (Fig 5.1(a)), it is quite easy to see that the map in Fig 5.1(b) is much less accurate than the map in Fig 5.1(c). This can largely be attributed to the fact that map building is a relatively new field, and most researchers have opted to concentrate on developing new map building methods and rely on qualitative analysis rather than spend time creating a comprehensive suite of map benchmarking techniques.

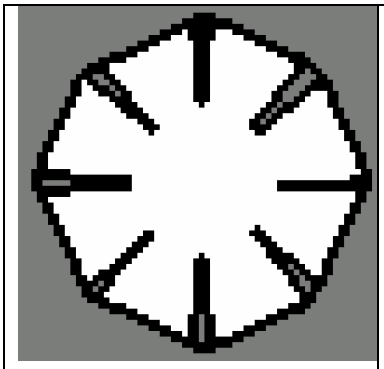


Fig 5.1 (a) The ideal map of an environment.

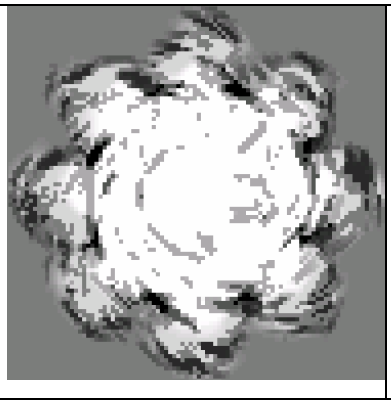


Fig 5.1 (b) An inaccurate map generated by running a robot around the environment from Fig 5.1(a).

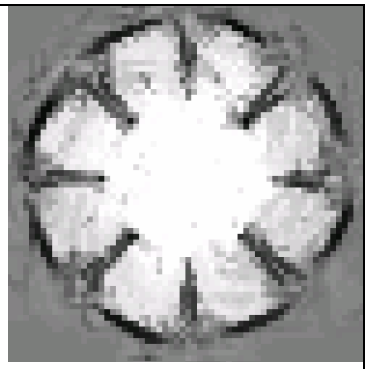


Fig 5.1 (c) A more accurate model of the environment from Fig 5.1 (a) than Fig 5.1 (b).

Unfortunately, the human eye is easily deceived, and basing ones conclusions upon purely qualitative measurements can leave the results open to question. Also, in fields where an exact measurement of the value of a map is required, for example the automated learning of a sonar model [57, 39], a fitness function must be used to evaluate the worth of the map created by the last iteration in order to guide the learning process in the future. For these reasons it is necessary to have a good method of benchmarking the maps created by the map building software. In the methods described below, the map being evaluated is compared to an ideal map of the environment, which was either taken from architectural drawings, or measurements of the real world environment.

5.2.1 Cross Correlation Between Maps

Given the aforementioned likeness of a two dimensional map to an image, it is possible to calculate the similarity between two maps, in this case the ideal map and

the generated map. A fitness measure of the map generated by the system is calculated using Baron's *cross correlation coefficient* [1]:

$$C_N(y) = \frac{\langle I_T T \rangle - \langle I_T \rangle \langle T \rangle}{\sigma(I_T)\sigma(T)}$$

This is based on *template matching*, where $C_N(y)$ is the cross correlation coefficient, I_T is the map to be matched, T is the original map being matched against, <> is the average operator which calculates the average value of all the freespace and occupied cells, and σ is the standard deviation over the area being matched. The result of this comparison, C_N, gives a percentage match figure that specifies the similarity of the two maps.

One of the most important features of any image/map comparison algorithm is that of *normalisation* – the map in question must be normalised in some way to ensure that neither scale, rotation nor translation affect the outcome of the comparison.

- The translation of all maps is made identical by anchoring both maps to same coordinate system, in this case both of their points of origin, position (0,0), are placed at Saphira's origin. In each of the experiments, the robot begins its run at the origin point.
- The problem of differences in scale of the maps being compared is solved by only comparing the area covered by the smallest map. After all, it wouldn't be very fair to compare a map-building strategy's abilities on an area it has never visited. Both maps also must use the same measurement scale; in this case each grid cell is 10 by 10 millimetres.
- The issue of differences in rotation of the maps being compared is resolved by initialising the robot with the same orientation at the beginning of each experiment. This step was taken to simplify the map comparison methods necessary, and would not affect the quality of the map built, although localisation routines may be slightly more accurate as a result.

The average, or mean, of a map M, is calculated using the following formula:

$$\langle M \rangle = \frac{\sum_{m_{X,Y} \in M} m_{X,Y}}{numCells}$$

with all the values in the map $m_{X,Y}$ being added together and the result divided by the number of cells in the map, *numCells*. The average of two combined maps, $\langle MN \rangle$, is achieved in a similar manner, with each corresponding cell in either map being multiplied with the cell in the other map, then added to the total, which is finally divided by the total number of cells, *n*.

$$\langle MN \rangle = \frac{\sum_{m_{X,Y} \in M, n_{X,Y} \in N} \left(m_{X,Y} * n_{X,Y} \right)}{n}$$

The standard deviation of a map can be calculated as one would expect, by subtracting the mean from each cell, summing these values, dividing by the number of cells in the map, and getting the square root of the result:

$$\sigma = \sqrt{\frac{\sum_{S_{X,Y} \in M} \left(S_{X,Y} - \overline{M} \right)^2}{numCells}}$$

Calculating the correlation coefficient between two maps is a relatively simple, quick procedure. It will get similar results between two maps if robot odometry error causes obstacles to appear distorted in the generated map, for example a straight corridor will appear to be curved. This is due to the fact that it factors the average and standard deviation of the map into the final result, as well as doing a cell by cell comparison.

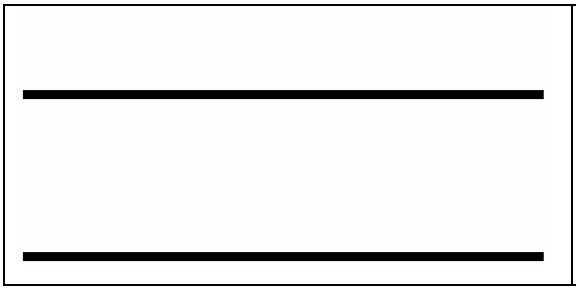


Fig 5.2 (a) Correlation can give a high percentage match to two maps even if they are quite different. This is because they have a similar number of equally valued cells.

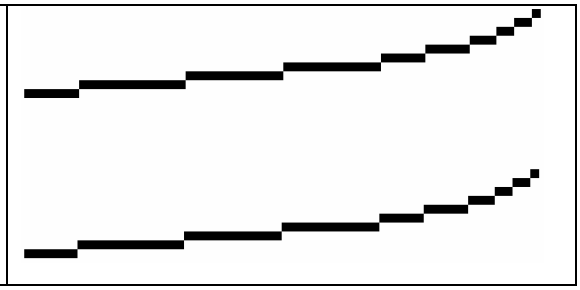


Fig 5.2 (b) This corridor, curved due to odometry error, would have very high correlation with Fig 5.2a as it has an equal number of occupied and freespace cells. However, it is certainly not an very good match in reality.

Herein lies the weakness of using correlation as a fitness measure for maps – it is very possible that an inaccurate map can appear accurate and vice versa since correlation takes into account the average cell value and standard deviation of cell value from the

mean. If two maps' average values are similar through coincidence rather than by being very similar then they might be given a high correlation value (See Fig 5.2).

Map correlation was implemented in the *GridMap* class discussed in Chapter 4, in the *correlation* function, and was applied between all generated maps and the ideal hand-drawn map of the environment, with the results in Chapter 6.

5.2.2 Map Score

Martin and Moravec [39] developed a map comparison measure called *Map Score* in order to facilitate the automatic learning of sensor models. Unlike correlation, map scoring compares two maps on a cell-by-cell basis, meaning that in order for it to give a meaningful result, the two maps must be in exactly the same position and orientation.

Given two maps, *M* and *N*, the score between them is calculated as the sum of the squared differences between corresponding cells:

$$Match = \sum_{m_{X,Y} \in M, n_{X,Y} \in Y} \left(m_{X,Y} - n_{X,Y}\right)^2$$

where $m_{X,Y}$ is the value of the cell at position (x,y) in map *M* and $n_{X,Y}$ is the value of the cell at position (x,y) in map *N*. This formula gives a positive value representing the difference between the two maps, so the lower the number, the more alike the two maps are.

This formula works well with the application for which it was designed, learning a sonar model, since the only requirement is to minimise the score/fitness on a single map. However, this score is not normalised, so the score a sonar model receives on one map bears no relation to the score it may receive on a second map. For example, on a small map with 1000 cells, a score of 500 would mean that for each cell in the ideal map, the generated map had an average difference of 0.5 – a huge error. Whereas on a large map of 1,000,000 cells, a score of 500 would mean that for each cell in the map there was an average error of 0.0005 – a very small error. As the sonar models developed for this thesis have been tested on a variety of environments, it became necessary to change the *Map Scoring* method slightly to allow normalisation so results from one map could be compared with the results from all other maps.

5.2.3 Normalising The Map Score

This change involved finding the worst possible map that could be compared to the ideal map. The ideal map only has three possible values, 0, 0.5, and 1, or empty, unknown and occupied respectively. A naïve method of finding the worst map would be to set each empty cell to 1, each occupied cell to 0, and each unknown cell to either 0 or 1 (it doesn't matter which, both have a difference of 0.5). This does not take into account that there are places in the map that the robot cannot go, for example there may be a large expanse of unknown area with no access to it. It therefore makes sense to create the worst possible map of an environment based on where the robot *can* go. This is done by setting the empty cells to 1, as above, but only change the value of an unknown or occupied cell if it is close enough to an empty cell to be detected if the robot happened to be at that empty location. In the experiments undertaken, a maximum range distance for the sonar sensors was set at 2.5 metres, so only those unknown or occupied cells within 2.5 metres of an unoccupied cell are changed, from 0.5 to 1 and from 1 to 0 respectively.

5.2.4 Testing Only Occupied Cells With Map Score

Another weakness of the *Map Scoring* technique is that mapping algorithms that overestimate the empty regions of space and do not identify many obstacles can receive a better score than algorithms that identify obstacles much more accurately, but don't make quite as strong a statement as to the emptiness of a region. The reason for this is that in many environments, there are large amounts of unoccupied spaces with boundary wall, with perhaps a few small obstacles distributed within that space. This means that there are often many more unoccupied cells than occupied cells. If a sonar model marks most cells as empty – not uncommon since bad sonar models are easily confused by noise and thus miss an obstacle – it is only slightly penalised since it computed the wrong value for a small number of cells. Unfortunately it just so happens that those are the very cells we most wish to identify correctly.

To redress the balance, a second *score* test is used that only compares the occupied cells in both maps. For any two maps M and N, if either the value $m_{X,Y} > 0.5$ or $n_{X,Y} > 0.5$, then the squared difference between those two cells is added to the final score. Otherwise they are ignored.

This method, in conjunction with the earlier method of comparing the complete maps against each other, has proven to give a very good indication as to the strengths and weaknesses of the sonar model used to build the map. They can identify when a sonar model updates either the occupied or unoccupied spaces too strongly or too weakly.

They also punish specular reflections that return a range reading and those that don't. If a specular reading doesn't return a range to the sensor causing an obstacle to not be identified, then both the first and second *score* methods will give a worse (higher) score. Also, if a specular reflection returns a reading that is too long, usually from reflecting off multiple objects before returning, then a sonar shadow behind the obstacle is created. In Fig 5.3, if the entire map were to be *score*d against an ideal map, then it would seem to be a very good map, since the freespace corridor is well identified and this makes up a large part of the map. However when comparing just the occupied cells, we can see that there are many more occupied cells than there should be, and this is more clearly shown by the second *score* method.

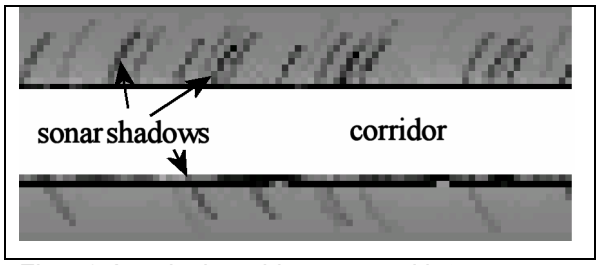


Fig 5.3 A typical corridor scene with '*sonar shadows*' resulting from noisy specular readings.

A similar test of the unoccupied areas was not carried out because sonars have very little trouble detecting open spaces, so comparing the two complete maps with each other gives a very similar result to what a test of just the unoccupied areas would.

Unfortunately, the greatest strength of this method – being able to very accurately compare two maps on a cell by cell basis – is also it's greatest weakness. Because it relies on the two maps being in the exact same orientation and translation, this map matching method is only valid either in simulation with no odometry error, or with very effective localisation algorithms continually correcting the estimated position of the robot. However, given these assumptions, it provides a very good indication as to the strengths and weaknesses of sonar models, feature prediction algorithms, the use of pose buckets etc.

5.3 Benchmarking A Map Based On Its Usefulness To A Robot

5.3.1 Introduction

Most quantitive analysis presented on the grid maps focus on techniques similar to those discussed in the earlier part of this chapter – namely comparing the values in the cells to each other. When training a sonar model in a small area this is quite a good approach to take, but to properly evaluate the worth of a map, one must get away from the idea of a map being like a picture, and ask the question "If a robot were to use this map, how well would it perform?"

The purpose of a map is, after all, to enable a robot to get from point *A* to point *B* as quickly and as safely as possible. However, many researchers ignore the fact that a map need not *perfectly match* the ideal map of an environment to be *perfectly usable* in that environment. It is not necessary for a map to be an exact replica of the surrounding environment, it is just necessary for it to be an abstraction of that environment which, when combined with an appropriate path planning algorithm, generates a true real world path for the robot to follow. It is the quality of these paths that truly give the value of the map, a value based on the use to which the map will be put rather than a metric based on techniques that can be seen as an extension of human vision such as map matching and image correlation.

5.3.2 Path Comparison - Testing a Map's Usefulness to The Robot

To get the true worth of a map, two elements of the map must be tested.

1. The degree to which the robot should be able to plan a path from one position to another using the generated map, but cannot – false negatives.
2. The degree to which paths calculated in the generated map would cause the robot to collide with an obstacle, and are therefore invalid – false positives.

Both of these are tested using only the map to guide the robot's movement.

There are four steps required to calculate the two above items of information from a generated map.

1. Calculate all possible paths, in the ideal map P^I, (see Fig 5.4). This is done by generating a Voronoi graph which is explained in section 5.3.3. Record the end-points of each path, E^I. For each pair of endpoints in E^I that have a path between them in the ideal map, attempt to create a path in the generated map between those two endpoints using a path planning algorithm (not the Voronoi graph).
2. Count the number of paths between endpoints in E^I that could not be completed in the generated map due to obstacles existing in that map where they do not exist in the ideal map – false negatives. The percentage of false negatives is then:

$$\frac{\text{number of incompleteable paths in the generated map}}{\text{total number of edges in the Ideal map}}$$

3. Calculate all possible paths in the generated map, once again using a Voronoi diagram.
4. Superimpose each Voronoi edge from the generated map onto the ideal map, and count the percentage of edges that pass through occupied spaces, and would therefore cause the robot to crash – false positives. This is equal to:

$$\frac{\text{number of edges that pass through occupied cells in the Ideal Map}}{\text{total number of edges in the Voronoi graph of the Generated Map}}$$

The Voronoi graph divides a map into separate freespace regions, with each node in the graph being the centre of a freespace region. It can therefore be used to identify 'places of interest' that the robot may with to visit, rather than a human hand-picking the paths to be tested within a generated map.

The paths in a Voronoi graph are not necessarily optimal in length, as they are designed to maximise clearance from all obstacles rather than minimise the length of the path.

It is also designed to find all possible paths in a map, and not to plan a specific path from one point to another. For these reasons, once the Voronoi graph has been used to identify the start and end-points of the paths to be tested, a separate path planning algorithm must be used. A modified version of the A-Star algorithm is used for this purpose, and is described in section 5.3.5.

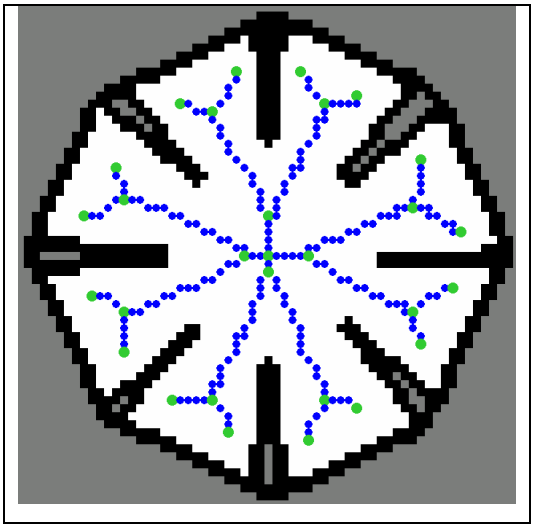


Fig 5.4 Star ideal map with inscribed Voronoi graph of all possible paths in the environment.

5.3.3 Voronoi Graphs

As mentioned earlier, a strategy must be employed to choose start and endpoints of the paths that are to be applied to a map. One possible solution to this problem, as mentioned above, is that a human pick the points in the map that seem to be 'of interest', and plot paths between them – for example, the ends of corridors, the centre of a room, corners etc.

Unfortunately this introduces the problem of the lack of objectivity of the human – any two humans are almost guaranteed to pick different points in the same map, and even a single human is very unlikely to pick the same set of points on the same map two times in succession.

Handpicking the points would also require more man-hours. It is therefore desirable to automate the selection of these 'points of interest'. This is done by generating a Voronoi graph based on the occupied areas in the map. A Voronoi graph

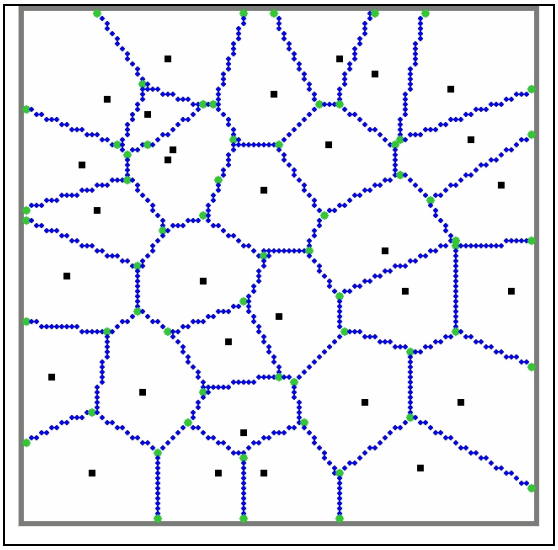


Fig 5.4 A Voronoi graph of an open area with multiple small obstacles represented black square dots. This shows all possible paths through the environment while maximising the clearance from all obstacles.

separates a map into 'Voronoi Regions', which are areas of freespace surrounding an obstacle (see Fig 5.4). Each node in the graph is the centre of an unoccupied region in the map, and is as such a 'place of interest' which the robot may wish to visit. The edges in the graph represent all possible paths through the environment which maximise the robot's clearance from obstacles.

Each of the edges in the Voronoi graph are either short straight lines as in Fig 5.4, or have very slight curves in them. Either way, planning a path from one Voronoi node to an adjacent node is a very simple task for a path planner, and will always result in an optimal or near optimal solution.

5.3.4 Generating Voronoi Graphs

A Voronoi graph is a graph of all possible paths in a map. These paths are not necessarily the shortest paths possible, but they do maximise the robot's clearance from all obstacles in the map. Every point on the graph is exactly equidistant from it's two closest obstacles, called *basis points*. Nodes on the graph (places where two or more edges meet) are points in the environment that are equidistant from three obstacles. Therefore, each point on the graph is maximally distant from all obstacles.

To generate a Voronoi diagram, first we observe that all points equidistant from any two obstacles o_1 and o_2 lie on the bisector B of the line L between those two obstacles (see Fig 5.5). All points on B are candidates for inclusion in the Voronoi graph as they are all equidistant from a pair of obstacles, o_1 and o_2.

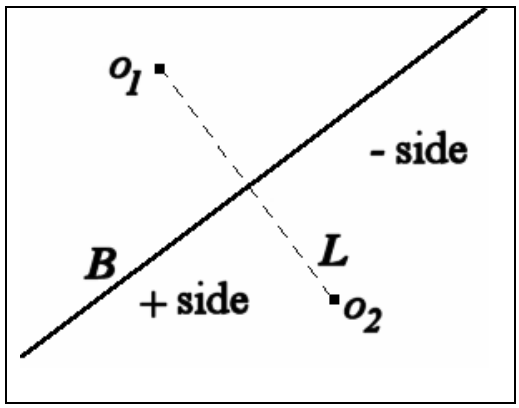


Fig 5.5 Two obstacles (black dots) and the points which are equidistant from them *B*. The line of equidistant points *B* bisects the line between the two obstacles *L* (dashed line).

What must be ascertained is whether there are any other obstacles closer to any point on the bisector line *B* than either o_1 or o_2. If there is another obstacle, o_3, closer to *B* than o_1 or o_2 then *B* is truncated at the point which is the centre of the circle that inscribes all three points (see Fig 5.6). This is possible, as a geometric rule states that, for any three points in a plane that are not arranged in a straight line, a circle can be drawn with all three points on its circumference. The reason for truncating the line *B* is that the centre of the circle, (c_x,c_y), which all three points lie on the circumference of, is equidistant from all three points, and any point on the line *B* closer to o_3 than the centre of the circle will be closer to o_3 than to either o_1 or o_2, and therefore not part of the Voronoi graph since it doesn't have two equidistant basis points.

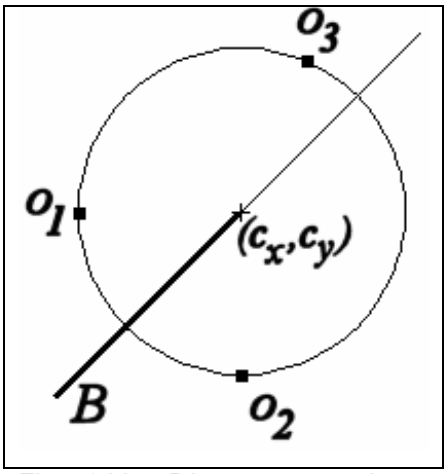


Fig 5.6 Line *B* between o_1 and o_2 is truncated at the centre of the circle inscribing o_1, o_2 and o_3. All points after the circle centre are closer to o_3, and not part of the Voronoi graph.

For each pair of obstacles, in this case each pair of occupied cells, a line *B* is created which bisects the line *L* between them. This line is checked against all other occupied cells in the map to find the truncation points that lead to the shortest line, *B*.

5.3.4.1 Finding The Truncation Points For the Bisector Lines B

Each line *b* can have 0, 1 or 2 truncation points. If there are only two obstacles in the map, the line *b* will have no truncation points. If there are three obstacles in the map, the line *b* will have at most one truncation point if at some point the third obstacles is closer to the line than the two obstacles which it bisects. If there are four or more obstacles in the map, the line *b* will have at most two truncation points, one at the either end of the line. All occupied cells being tested against the line are divided into two groups: those on the positive side of the line *L* and those to the negative side of *L* (see Fig 5.5). What side of *L* any given point is on is calculated by finding the perpendicular distance between that point and *L* using the following formula.

Given that L is a line passing through two points (x_1,y_1) and (x_2,y_2), and we wish to find the perpendicular distance to the point (x,y):

$$A = y_2 - y_1$$

$$B = x_1 - x_2$$

$$C = (x_2 * y_1) - (x_1 * y_2)$$

with the perpendicular distance to the occupied cell (x,y) being

$$((A * x) + (B * y) + C) / \sqrt{A^2 + B^2}$$

This formula gives either a positive or negative value depending on what side of the line (x,y) is on. For example, two points could be at a distance of 5 and –5 respectively from a line. While they are at equal distances from the line, they would be on opposite sides of it.

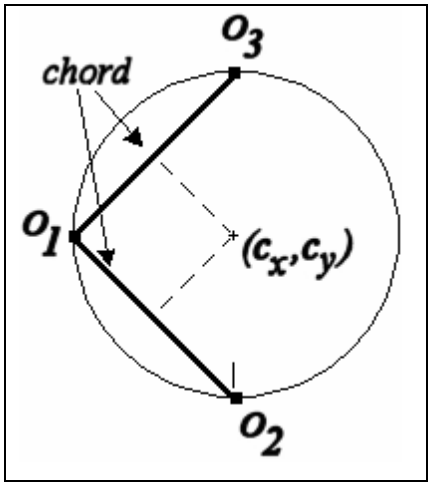


Fig 5.7 To find the centre (c_x, c_y) of a circle inscribing three points, bisect any two chords between the points, and get their intersection.

For each obstacle o_3 in the map, the point on the line B that is equidistant from that obstacle, o_1 and o_2 is found by bisecting any two of the three possible chords between o_1, o_2 and o_3 and finding their intersection (see Fig 5.7). The perpendicular distance of that point to L is calculated.

For all points on the positive side of the line L, the truncation point T^+ with the most negative value is chosen, and for all points on the negative side of L the truncation point T^- with the most positive value is chosen. The only time a circle cannot be inscribed between three points is if they are in a straight line. In this case, if o_3 is between o_1 and o_2 the line is discarded, since it will be closer to all points on B than either o_1 or o_2. Otherwise it is ignored.

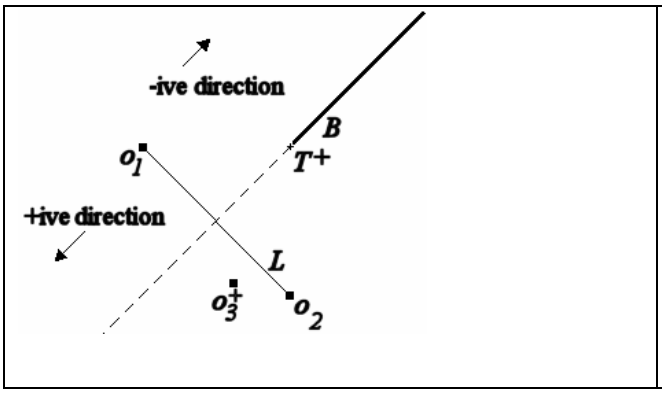


Fig 5.8 (a) The obstacle o_3 is on the positive side of the line L (hence the + superscript), with the centre of the circle inscribing the three points at T^+ (the + superscript indicating that it is the truncation point caused by a point on the positive side of the line). All points on the line B in the positive direction of T^+ are discarded.

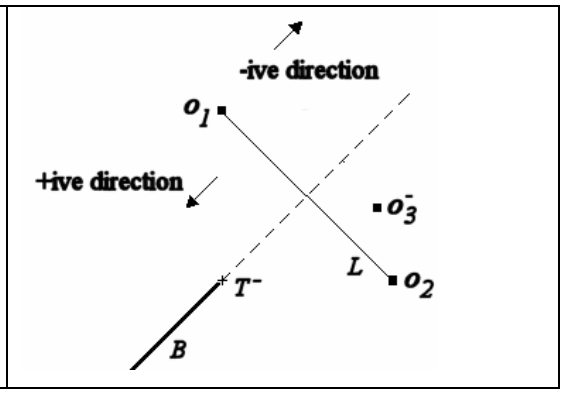


Fig 5.8 (b) The obstacle o_3 is on the negative side of the line L, with the centre of the circle inscribing the three points o_1, o_2, and $o3$, at T^-. All points on the line B in the negative direction of T^- are discarded.

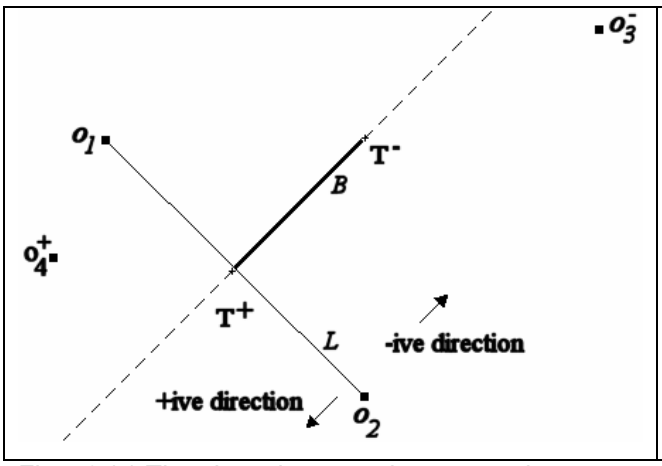


Fig 5.8 (c) The obstacles o_3 and o_4 are on the negative and positive sides of L respectively. The centre of the circle through o_1, o_2 and o_3 is at T^-, and the centre of the circle through o_1, o_2 and o_4 is at T^+. All points in the positive direction of T^+ are discarded, and all points in the negative direction of T^- are discarded.

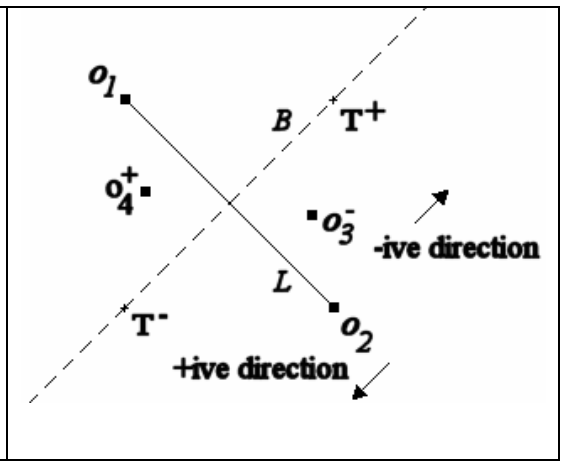


Fig 5.8 (d) The obstacles and circle centres are as in Fig 5.8 (c), but this time the complete line B is discarded since T^+ is to the negative side of T^-. All points in the positive direction of T^+ are discarded, and all points in the negative direction of T^- are discarded, but in this case this is the complete line.

If T^+ is less than T^- then the line B is discarded as there are points on both the positive and negative side that are closer to all points on the line B. Otherwise, if a point T^+ was found then all points on the line B that have a perpendicular distance

from L greater than the perpendicular distance of T^+ are discarded. If a truncation point T^- was found, all points on B with a perpendicular distance from L less than the perpendicular distance of T^- from L are discarded. Fig 5.8 shows the four possible scenarios for eliminating candidate points from the Voronoi graph.

5.3.4.2 Computational Cost of Generating Voronoi Graphs

Generating Voronoi graphs is a very costly process in terms of computation time. Given that there are n occupied cells in a map, a total of $\frac{n(n-1)}{2}$ bisector lines B must be tested for inclusion in the graph. This is because a line must be created and tested between the n^{th} occupied cell and all other cells, another line between the $(n-1)^{th}$ occupied cell and all cells from the $(n-2)^{nd}$ cell on downwards etc, so the series looks like:

$$(n-1) + (n-2) + \ldots + (n-(n-1))$$

This is a very common sequence, which equals $\frac{n(n-1)}{2}$.

Each line must then be compared with a theoretical maximum of n cells in order to ensure that the correct truncation points are chosen. This gives an $O(n^3)$ algorithm. For a large map with many thousands of occupied cells, even on a very fast computer this can take days to complete. For example, a map with 10 occupied cells (a very small number) would have to examine the combination of $10\left(\frac{10(10-1)}{2}\right) = 450$ cells, whereas a map with 11 occupied cells has to examine the combination of $11\left(\frac{11(11-1)}{2}\right) = 605$ cells, a large increase. Therefore as the number of occupied cells increases, the number of calculations necessary increases exponentially – one of the maps generated in experiments had 16,000 occupied cells, requiring 2,047,872,000,000 iterations!

It is possible to cut down the running time somewhat by performing some simple pre-processing of the data. Firstly, any occupied cell that is completely surrounded by other occupied cells cannot be a basis point, as for any point in the freespace one of the cells surrounding it will be closer to that freespace cell than itself. Therefore these

cells are not used to create bisectors. The more densely packed a map is with occupied cells, the more effective this step is at reducing the dimensionality of the problem.

A second pre-processing step that has proven to drastically improve processing times is as follows. Every line *B* passes through a certain number of freespace cells. Rather than evaluating *B* against every occupied cell in the map, we observe that, for example, at a given *y* level in the map, there is an occupied cell to the left of the line, then any cells to the left of that occupied cell *must* be farther from the line and can therefore be ignored. The same goes for cells above, below and to the right of the line.

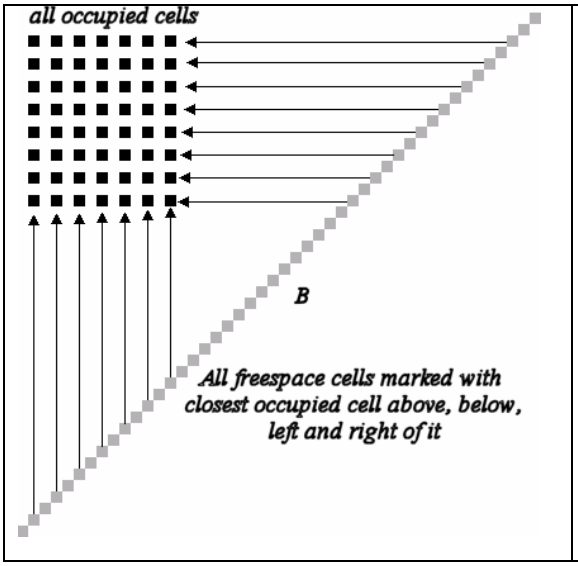


Fig 5.9 (a) All freespace cells record the closest occupied cell above, below, left and right of it.

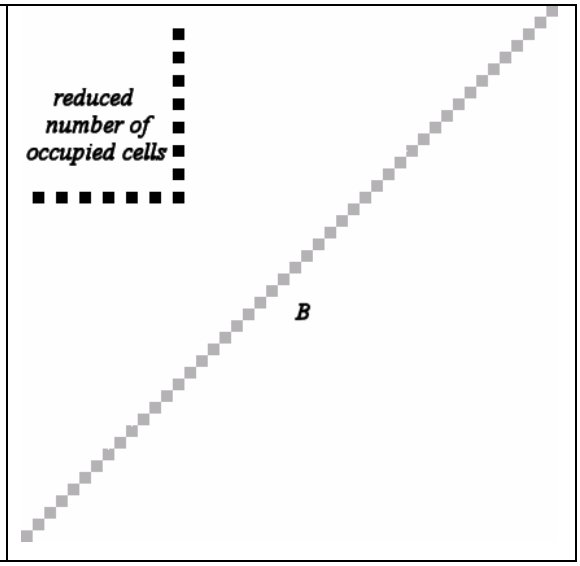


Fig 5.9 (b) Only the occupied cells referenced by the freespace cells *B* passes through are tested for their truncation points *T* since they are closer than all other occupied cells.

The map is pre-processed, storing at each freespace cell the position of the closest occupied cell above, below, left and right of it (see Fig 5.9). Each time a line *B* is being tested for inclusion in the Voronoi graph, the freespace cells it passes through are identified, and only those occupied cells referenced from the freespace cells are tested.

5.3.5 The A-Star Path Planning Algorithm

The A-Star path planning algorithm is a relatively quick and simple method of plotting a path from once freespace position in a map to another. It operates as follows.

Each cell in the map has three values associated with it, the *G* value, *H* value and the *F* value.

The *G* value is an estimate of the value of being in that position based on the straight line distance from it to the goal. It is used to direct the search in the direction of the goal rather than testing all possible cells and paths.

$$G = \frac{dist(currentCell, goalCell)}{dist(startCell, goalCell)}$$

The *H* value is an estimation of the value of being at that cell given the probability of colliding with an obstacle if it were there, i.e. the occupancy value of that cell.

$$H = probOcc(currentCell)$$

The *F* value is a weighted combination of the *G* and *H* values. For example, if it were desirable to stay away from obstacles as much as possible, the *H* value would be weighted much more strongly than the *G* value. If it were decided to allow the robot to come close to objects, the *H* value's weight could be lower. In my implementation, they are weighted equally, at 50% each. *F* then becomes

$$F = (G * weightG) + (H * weightH)$$

The algorithm works as follows:

```
PUSH(orderedList, start cell)
WHILE(orderedList not empty)
        currentCell = POP(orderedList)
        IF(currentCell == goalCell) THEN
                GOAL FOUND
                FINISH
        ELSE
                FOR i = 0 TO 8
                        IF(CHILD(currentCell, i) NOT OCCUPIED)
                                ASSIGN_F_VALUE(CHILD(currentCell, i))
                        PUSH(orderedList, CHILD(currentCell, i))
                END-IF
                END-FOR
        END-IF
END-WHILE
```

Fig 5.10 A-Star algorithm for generating a path in a metric grid-based map.

As detailed above, the A-Star algorithm pops whichever cell is currently valued the lowest off the list, and pushes each of the 8 'child' cells surrounding it onto the list, providing the child cell is not occupied by an obstacle. Each child cell is assigned an *F* value before being pushed onto the list. This continues until the goal is reached or all possible avenues have been attempted and discarded.

The main problem with the A-Star algorithm is its tendency to become trapped in local minima. Because it does not test every possible path, and instead uses the linear distance to the goal to estimate the best direction to take, in complex or cyclic environments it can often produce paths that are far from optimal.

In Fig 5.11 the A-Star algorithm plots a path from the start point to the end goal while maintaining a distance of at least 200mm, or two cells, from any obstacle. Not only is the complete path far from optimal, it also becomes trapped in some local minima. The path would be much shorter if the robot circumvented the maze at the beginning, but because the end-goal point is below the start point the A-Star algorithm only searches in a downward direction, resulting in a traversal through the maze.

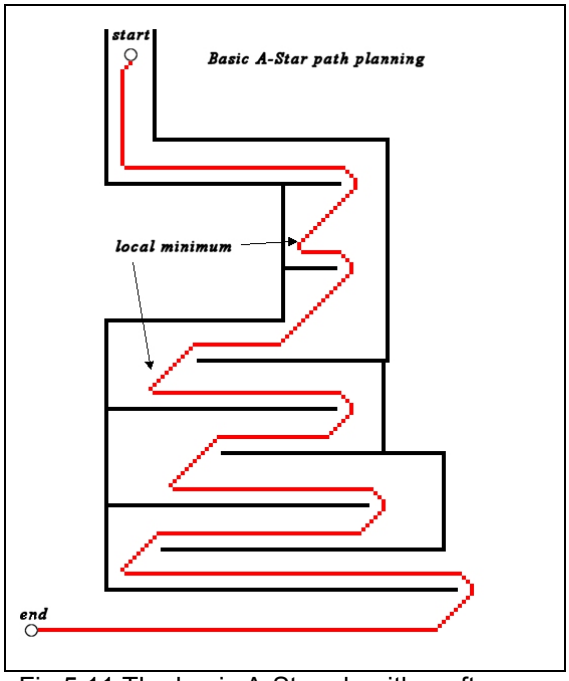


Fig 5.11 The basic A-Star algorithm often becomes trapped in local minima due to its using linear distance to the end-goal as an estimation of the worth of a path.

One improvement that can be made in order to reduce the effect of local minima is applying a line fitting algorithm to the overall path. This involves attempting to draw straight line segments between points in the path. If the straight line doesn't come too close to an obstacle then the path is replotted between the end points of the line. The A-Star algorithm, in its basic form, represents a path as a continuous series of points. Applying a line fitting algorithm to the path not only reduces the local minima, it also discretises the path into a set of waypoints, which is useful for giving the robot motion commands since it can be told to go towards a particular point, and know what next point to go to after that

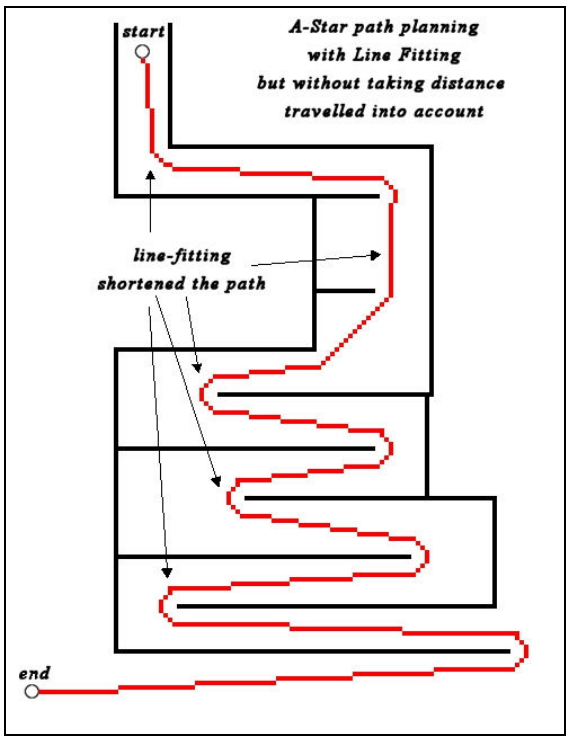


Fig 5.12 The A-Star algorithm after line fitting has been applied to the map. Line fitting is useful in reducing the effect of local minima, but does not prevent the algorithm from becoming stuck in global minima.

etc. Fig 5.12 shows the results of applying line fitting to the path generated by the A-Star algorithm in Fig 5.11.

Unfortunately, while line-fitting helps to reduce the effect of local minima, the path can still become trapped in global minima, as can be seen in Fig 5.12. The reason for this is that although the best path would involve the robot first travelling *away* from the end-goal, this will never happen because each point of the path in the Fig 5.12 is closer to the goal than all the points above the start point, and therefore have a lower *F* value, or overall cost. This is similar to the problem often encountered in Evolutionary Computation (EC), where a global optimum solution to the problem is surrounded by local minima. The solution to problem is often to perform more general exploration, as opposed to the more computationally efficient directed search.

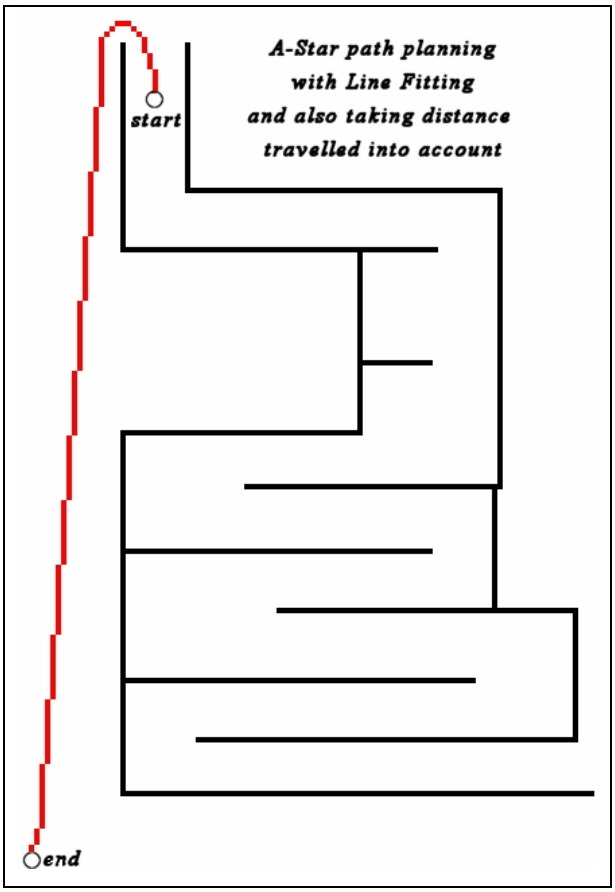


Fig 5.13 The A-Star algorithm with line fitting, as well as taking into account the length of the path *and* the linear distance to the goal, as opposed the linear distance to the goal on its own.

The solution described above also works when applied to the problem of path planning. In this case it involves introducing a fourth variable to the equation used to calculate the *F* value, or cost of being at any particular point in the map. This variable, the *I* value, is the length of the path up to that point.

$$I = pathLength(startCell, currentCell)$$

This is factored into the calculation of the overall cost of being at that node, *F* as follows:

$$F = (G * weightG) + (H * weightH) + (I * weightI)$$

The only constraint is that the weight of *G* must be greater than the weight of *I*, otherwise, as the path approached the goal *F* would increase rather than decrease. For example, if the path moved from one cell to another cell in the direction of the goal, with each cell being 100mm to a side, it would come 100mm closer to the goal, but the path would also be 100mm longer. If *weightG* > *weightI*, then the cost of the path decreases as it approaches the goal. However, if *weightG* < *weightI* then the cost of the path will increase as it approaches the goal, because *G* decreased at the same rate as *I* increased.

As long as the path approaches the goal, the *F* value will decrease because although $I * weightI$ becomes larger, $G * weightG$ decreases by a more significant amount. However, when the path continues to lengthen but does not come closer to the goal, the *I* value continues to increase since the path continues to lengthen, but the *G* value either stays the same or increases, causing the cost of the path to increase. When the path becomes sufficiently long, and has not become significantly closer to the goal, the cost of the next node on the path currently being explored becomes greater than the cost of earlier nodes, causing alternative routes to be explored, as in Fig 5.13.

Chapter 6: Experimentation Results

6.1 Introduction

Experimentation involved testing multiple systems on identical environments, both simulated and real world, and comparing the results. The systems are based on papers reviewed in chapter two by authors Moravec, Elfes, Matthies, and Konolige, as well as containing modifications created for the purpose of this thesis. All code used was written from scratch for this thesis in C++ and Flex, interfacing with the Saphira APIs.

6.2 Platform

The first stage of experimentation took place on the Saphira client simulator (see chapter 4), interfacing with the Pioneer robot simulator. This simulator is very robust, well supported, often updated with additions directly requested by the users, and designed specifically for the ActivMedia Pioneer robots upon which experiments were performed.

The second stage of experimentation was performed using a Pioneer 1 robot and the Saphira client, running map building algorithms.

The decision had to be made whether to develop on the Linux or Windows platform, and it came down in Linux's favour. There were three primary reasons for this. Firstly, the current Saphira release, version 8.1, is more robust and better supported on Linux than on Windows. Secondly, the majority of documentation and support is from a Linux perspective, with Windows only occasional being mentioned. Thirdly, the long term goal of the UL robotics group is to develop a wide range of inter-operable robotic service modules to create a truly autonomous robot, able to map a location, plan a route, localise within that environment, deal with noisy data and dynamic real life situations. The software architecture presented in chapter four shows the flexible architecture designed for this purpose, with the initial additions of mapping modules developed for this thesis, as well as pursuit and evasion modules and mapping modules based learned sonar models developed by others in the group.

In order to ensure that all members modules worked with each other, it was necessary to decide on a group-wide platform to develop on, and the majority decision favoured Linux for its robustness, its very flexible programming environment as well as its familiarity to the members of the group.

6.3 What Is To Be Proven?

Two sets of results are presented:

Section 6.9 aims to show the strengths of the sonar models and probability update strategies put forward by Matthies, Elfes [42, 46], Moravec [46] and Konolige [33], as well as the modifications on the basic theories described in chapter four.

There are two sonar models tested. *ME85, ME85mod, ME88* and *ME88mod* use the same two dimensional gaussian sonar model, as described in [46] and section 2.8.1. *K97* and *K97mod* use the sonar model developed by Konolige in [33], which is based on the standard normal distribution, although *K97* has a stronger bias towards occupied readings, whereas *K97mod* has more of a bias towards freespace readings.

The modifications applied to the original theories, as described in detail in chapter four are:

- Improved dynamic mixture model for detecting specular readings from past data – *ME85mod, ME88mod* and *K97mod*.
- Feature Prediction - *ME85mod, ME88mod* and *K97mod*.
- Pose Buckets as an additional feature – *ME85mod, ME88mod*.
- Removed a bias towards freespace readings – *ME85mod*.
- Reduced a bias towards surface readings – *K97mod*.

Section 6.10 aims to show the strengths and weaknesses of the feature prediction method presented in chapter 3, as well as the strengths and weaknesses of using pose buckets to ensure the independence of sonar readings. The combination of these two methods is also discussed, with the output of the mapping systems with neither of these additions being taken as a base line benchmark. Only three mapping systems were used to provide the results for section 6.10, *ME85mod, ME88mod*, and *K97mod*.

This is because only these three systems used both feature prediction and pose buckets, and to include any of the other systems in the base line benchmark would skew it since these systems would be included in some results and not others, making the comparisons between results less accurate.

6.4 Experiment Plan

6.4.1 Real World Experiments

Experiments were performed with a number of different environments and different mapping systems, both on simulator and on real physical systems. No localisation was performed during test runs, either on the simulator or in the real world experiments, as localisation is beyond the scope of this research. For real world navigation of a mobile robot however, localisation is required to recover from odometry errors such as wheel slippage and angular drift. The need for localisation is especially obvious in large cyclic environments, such as the *eCSB* environment described later, where straight walls can appear curved, and where a point visited earlier by the robot may seem to be a new location when returned to. A technique called Simultaneous Localisation and Mapping, or SLAM, is often used to combat this effect. However, in small environments the effect of odometry error can be minimised, therefore the real world experiments take place in the *eStar* environment described in section 6.4.5, which is small enough that the robot's reported pose is within an acceptable tolerance.

6.4.2 Simulated Experiments

The Pioneer simulator simulates two types of noise. The first is sonar noise, in which it models the specular manner in which sonar beams reflect off smooth obstacles. The second is odometry error, in which it simulates a set degree of wheel slippage and angular drift. For the simulated experiments, odometry error was turned off to facilitate the benchmarking of the maps generated, since the benchmarking techniques rely on localisation being present to account for wheel slippage. Due to the fact that localisation is beyond the scope of this thesis, running simulated experiments with no odometry error results in more accurate and meaningful benchmark figures when testing map building methods. This also facilitated the comparison of the resulting

maps since they can be directly compared because they all have the same coordinate system.

6.4.3 Averaging Multiple Results For A Statistically Valid Sample

Each simulated experiment was performed five times, with the exception of the experiments in the *Star* environment, which due to its small size and homogeneity were only performed three times. For real-world experiments, three runs were executed around the *Star* environment.

The average result of all five runs of the robot around an environment was taken as the overall result. The reason for not simply basing the result on a single test run is that, for example, sometimes a robot will be turned at a particular angle and not detect a wall which would result in a poor map being generated, whereas if it turned slightly it might pick up the obstacle and incorporate it into the map. The reverse is also true in that just because it detected an obstacle in one test run it doesn't mean it will detect it in another. Therefore in order to get a true indication of the worth of all the various mapping methods, the robot traversed each environment multiple times, taking different paths, different speeds and moving differently – going straight, moving in a snake-like manner etc. Five test runs was considered to provide a statistically valid sample, especially when the many varied movements and paths traversed are taken into account.

6.4.4 Naming Convention

The naming convention used in the following systems and test environments is as follows:

Systems: To reiterate the naming convention described in Chapter 4, each system is based on a paper published by a certain author in a given year. The name of the system begins with the first letter(s) of the surname of the author(s) of the paper. This is followed by the year of publication of the paper. E.g. *ME88* is the system based on the 1988 paper by Larry Matthies and Alberto Elfes. If the system features modifications of the theories in the original paper, its name is followed by *mod*.

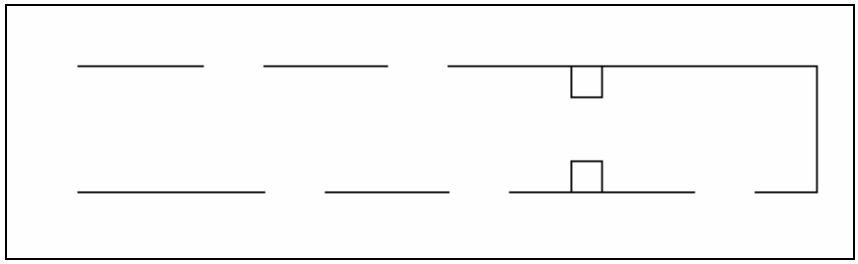
Environments: All environments begin with the letter *e* to distinguish them from the systems. The name of the environment follows in caps. If the environment is on the

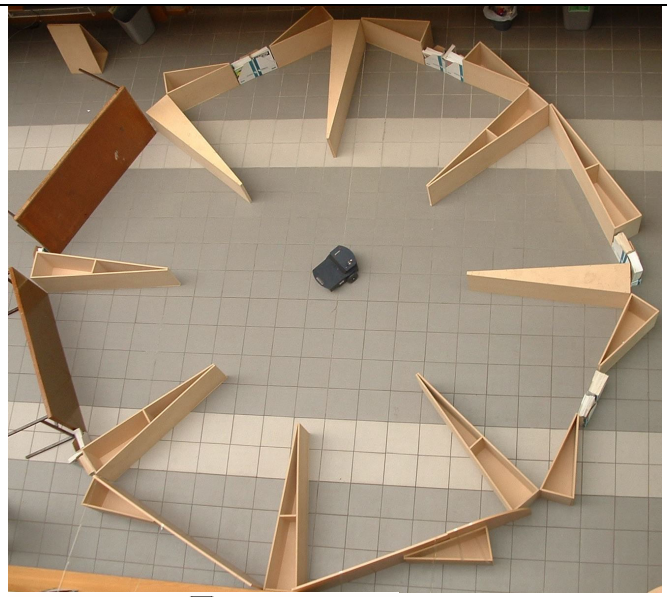
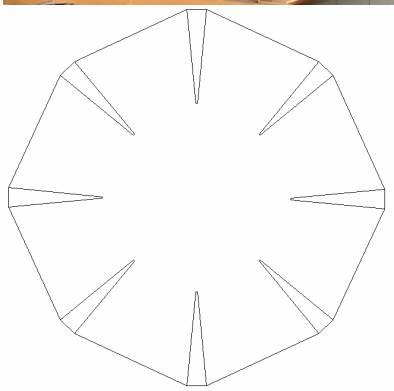
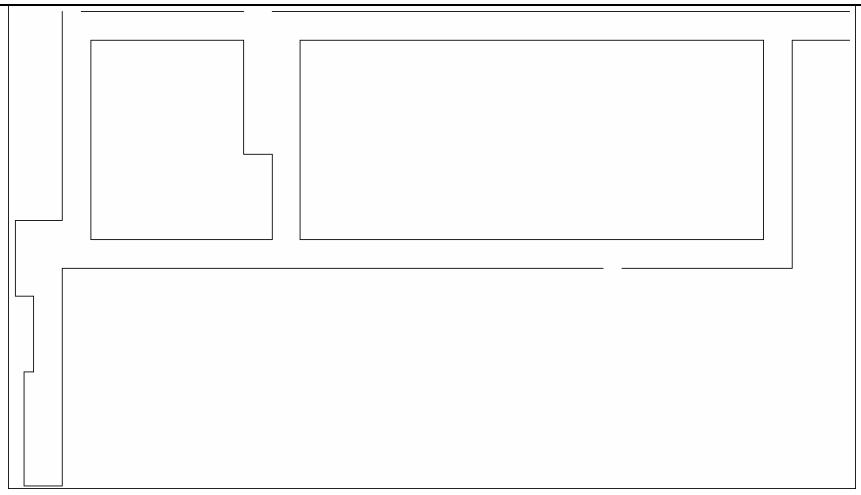
simulator, the name ends in *sim*. If it is in the real world, it simply contains the environment's name.

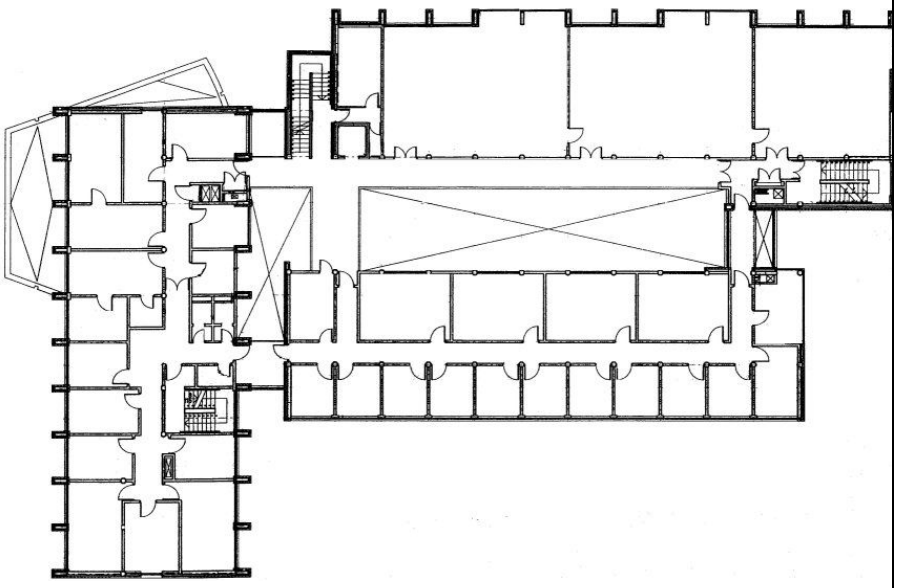
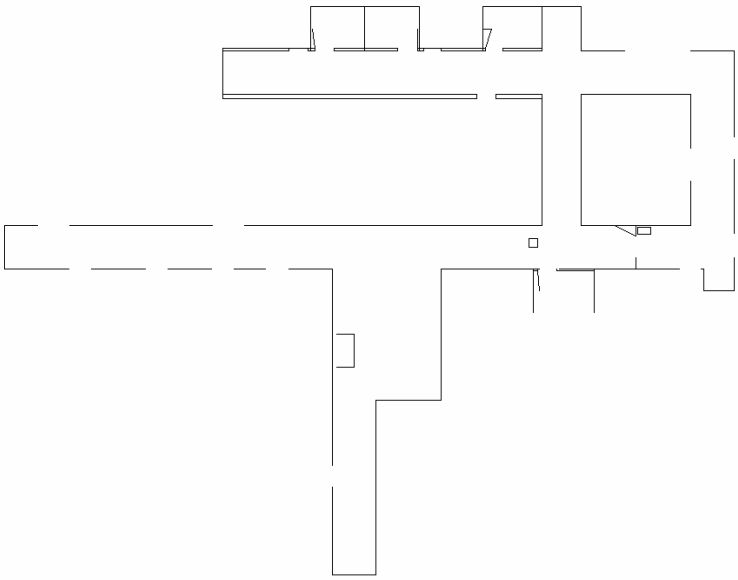
Four different environments were tested in simulator, and two in real-world experiments. All systems were tested on each environment. However, while real-world experiments were carried out on the large environment, *eCSB* (see the map later in this chapter in Fig 6.1), the lack of localisation routines means that the resultant map was very skewed. As the benchmarking techniques are reliant on the robot having quite a good sense of its position, only the experimental results from the *eSTAR* environment are used because it is quite small, and therefore odometry errors can be minimised.

6.4.5 Simulated And Real World Environments Used In Experimentation

The six environments are as follows.

Environment Name	**Description**
eCORRsim	 A corridor with obstacles. Width = 12000mm , Height = 2000mm

eSTAR & eSTARsim	  A star shaped environment designed to show the extreme effects of specular reflection on the accuracy of a map. The real-world environment, *eSTAR*, was built using triangular wooden blocks as can be seen in the picture above. Width = 5977, Height = 5977
eCSB & eCSBsim	 The first floor of the Computer Science Building in the University

	of Limerick. This environment consists of a number of corridors, with doors and stairs. The map above was generated from the architectural drawing below, factoring out all doors not accessible such as private offices. Otherwise all measurements are correct to the millimetre.  Width = 44077mm, Height = 19000mm
eAICsim	 A large office space with corridors, doors and miscellaneous obstacles, based on the *aic.wld* map included in the Saphira software suite, which is an architectural blueprint of the office area used by the Saphira development team.

	Width = 38000mm , Height = 30000mm

Fig 6.1 Environments used in experimentation.

6.5 Data Capture

6.5.1 Simulated Data Capture

For three of the four simulated environments, e*CORRsim, eCSBsim* and *eAICsim*, five test runs were performed, with three test runs being performed on *eSTARsim*, giving a total of eighteen test runs. The data collected from these test runs comprised of a list of records, each recorded every 100ms from the robot. Each record consisted of the robot's Cartesian position, the angle it was facing, and the range readings from all its sonars at that point in time. This information was stored in a file at the end of the test run for further processing later.

Each of the eighteen simulated test runs were processed many times using a batch processor programme, to fully illustrate the properties of all sonar models, mathematical update methods, feature prediction and pose buckets. Each test run was used to generate sixteen maps as explained in Fig 6.2, giving a total of 18 * 16 = 288 maps generated.

Mapping System	**Variation(s) used**: **FP = Feature Predication, PB = Pose Buckets**				**Number of maps generated per test run**
ME85	No FP. No PB				1
ME88	No FP. No PB				1
K97	No FP. No PB		No FP. With PB		2
ME85mod	No FP No PB	No FP With PB	With FP No PB	With FP With PB	4
ME88mod	No FP No PB	No FP With PB	With FP No PB	With FP With PB	4
K97mod	No FP No PB	No FP With PB	With FP No PB	With FP With PB	4
Total Number of Maps Generated Per Test Run					**16**

Fig 6.2 Enumeration of the maps generated from each test run performed.

Averaging of the maps was done on the basis that any subset of the total maps that were based on the same environment, generated with the same mapping system and using the same variations on the basic mapping system would be averaged. For example, on the *eAICsim* environment, the *ME85mod* system with feature prediction turned on and pose buckets turned off was used to generate five different maps, one for each test run. These five maps were averaged.

Once the averaging process was complete on the maps generated from simulated data, sixteen maps remained for each environment used. These sixty-four averaged maps are the basis upon which all simulated experimentation results presented later are founded.

While each map is itself sufficient for performing the first three benchmarks outlined in chapter five, Correlation, Map Score, and Map Score Occupied Cells, for the final two benchmarks Voronoi graphs must be generated for each map. This meant that sixty-four Voronoi graphs were generated, one per map.

6.5.2 Real World Data Capture

Real world data capture took place in the *eSTAR* environment by tele operating a robot over a radio modem connection. The data generated from this experiment is identical in format to the data in the simulated experiments – a list of records, with each record containing the robot pose and the range readings for its sonars taken once every 100ms during the test run.

Three test runs were performed in the *eSTAR* environment, as with the simulated embodiment of *eSTAR, eSTARsim*. The data was processed in an identical fashion to the simulated data, with sixteen maps being generated for each test run, giving a total of $3 * 16 = 48$ maps. Once averaging was applied to these map, sixteen maps remained from which to illustrate the real world performance of the mapping systems.

6.6 Offline Processing Of Maps For Benchmarking

Most offline processing of the maps took place on an application specifically created for this thesis, called *MapViewer*. The *MapViewer* application was used to average

maps, calculate correlation coefficients, and calculate map score for the complete map and for occupied cells. In addition to performing tasks directly related to experimental results, the application can also:

- Generate paths in a loaded map using the *PlanPath* object described in chapter four.
- Generate Voronoi graphs of a loaded map. Due to the computationally intensive nature of generating Voronoi graphs, however, the MapViewer application was not used to generate all the Voronoi graphs required. Instead, a batch processing program was used that could run on many machines simultaneously, returning the generated Voronoi graph to a central server, often after a period of days.
- Display the path a robot took during a test run, as well as animating the path. This feature was implemented for the members of the UL Robotics Team performing research into pursuit and evasion techniques, who required visual feedback from their offline experiments of one robot chasing another.

The application contains three modes, *Map Mode* as in Fig 6.3 (a), *Path Mode* as in Fig 6.3 (b), and *RobotRun Mode*, in Fig 6.3 (c).

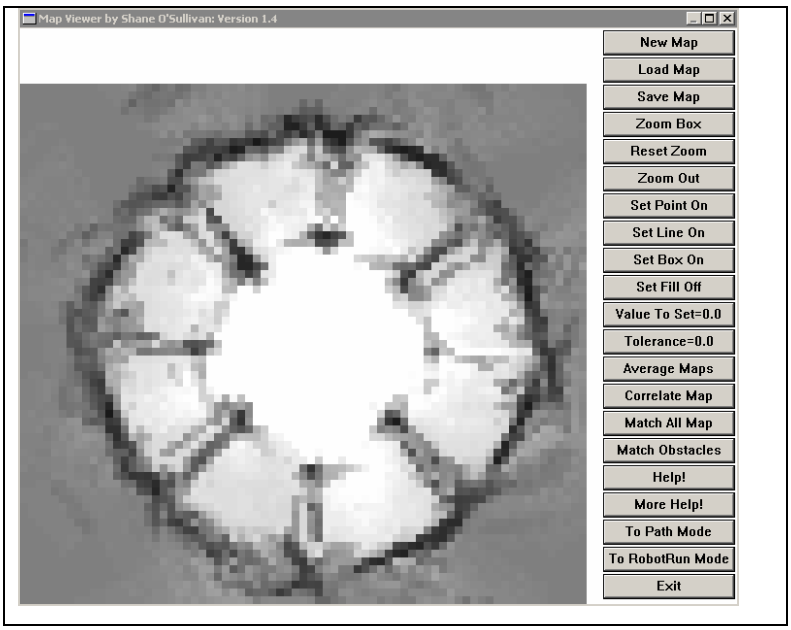


Fig 6.3 (a) The MapViewer application in *Map Mode*.

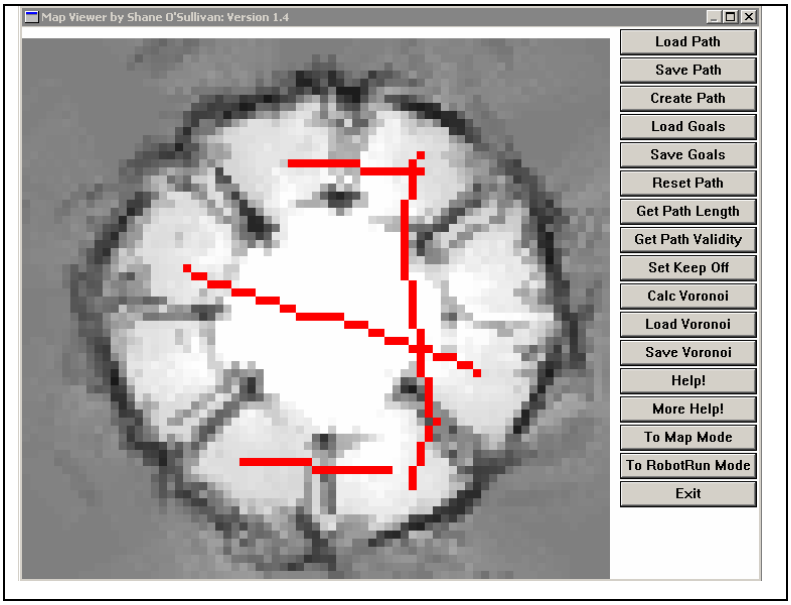


Fig 6.3 (b) The MapViewer application in *Path Mode*, with four paths displayed.

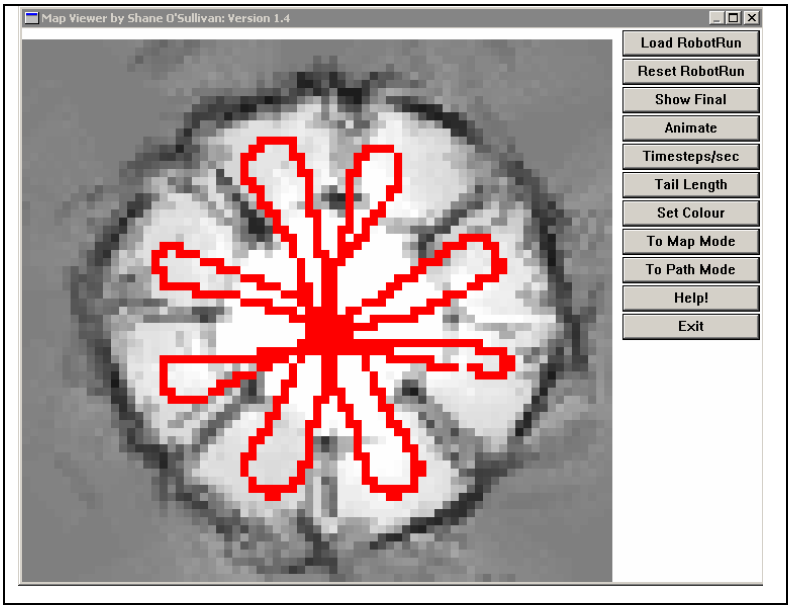


Fig 6.3 (c) The MapViewer application in *RobotRun Mode*. The path the robot took is displayed inside the *eSTAR* environment. Either the complete path can be displayed, or it can be animated to show multiple robots moving relative to each other in temporal space.

While the MapViewer application is capable of calculating the first three benchmarks, Correlation, Match All Cells, and Match Occupied cells, a further two programs were written to carry out the final two path comparison benchmarks, each of which is based on Voronoi graphs and the modified A-Star path planning algorithm.

The first program, *VoronoiGenerator*, took a map as a parameter and generated a Voronoi graph from it. However for large maps such as *eCSBsim* and *eAICsim*, generating a Voronoi graph on a desktop computer can take a week or more. For this reason the *VoronoiGenerator* application was deployed on twenty computers running in parallel, each of which reported back to a central server with the completed Voronoi graph for the map they were provided with. Using this configuration resulted in the generation of the sixty-four Voronoi graphs taking less than three weeks in total.

The second program used to perform the final two benchmarks, called *PathEvaluator*, took a map and a Voronoi graph as parameters and generated all the necessary results.

6.7 Experimentation Results

Three types of direct map comparison are applied to each experiment, *Correlation*, *Match All Cells* and *Match Occupied Cells*. *Correlation* uses image correlation techniques to compare two maps and is measured in percentages, with a higher percentage signifying that the two maps are more similar than a lower percentage. Both *Match All Cells* and *Match Occupied Cells* measure the difference between two maps, therefore a lower score signifies that the two maps being compared are more similar than a higher score. The two maps being compared have to be at exactly the same rotation and displacement, but if these conditions can be met then it is a more accurate comparison than *Correlation*.

Match All Cells matches each cell in the first map against the corresponding cell in the second map, while *Match Occupied Cells* only compares the cells in either map that are marked as occupied. The reason for carrying out two *Match* comparisons is that, when the complete map is compared using the *Match All Cells* method, relatively inaccurate maps can achieve a good score. This is due to the fact that the majority of cells in many maps are marked as empty, and specular sonar readings can cause far too much of a map to be marked empty. The *Match Occupied Cells* benchmark compensates for this by evaluating the degree to which a mapping method marks freespace areas as being occupied. When the two *Match* benchmarks are used in conjunction, they give a very accurate measure of the accuracy of a map.

Two methods of evaluating the usefulness of a map as a means of navigation are also applied to each experiment. Firstly the percentage of paths in a generated map that would pass through an occupied cell in the ideal map is calculated. This tests the degree to which a mapping method does not recognise obstacles, causing it to plan paths through areas where the robot cannot go. Secondly the percentage of paths in the real world that could not be completed in the generated map is established. This tests the degree to which the mapping system creates obstacles where they do not actually exist, causing it to be unable to create a path between two freespace areas which it should be able to.

Two sets of results are presented. Section 6.9 tests just the basic sonar models of the six map building systems, without any pose buckets or feature prediction in order to directly compare the various sonar models and mathematical probability update procedures. Section 6.10 examines the benefits of using pose buckets and feature prediction as means of discarding sonar readings.

Sixty-four maps generated from simulated test runs were used to derive the results presented in later sections, in addition to sixteen maps from real world experiments. In order to extract trends from the large body of data, all figures for a particular benchmark result are averaged. For example in section 6.9.1, the correlation benchmark figure for *ME85* was calculated by averaging the correlation results for *ME85* on all maps, over all test runs. The averaging is performed differently in section 6.10. In section 6.10 the goal is to illustrate the effect feature prediction and pose buckets have on the generation of a map. Therefore to calculate, for example, the correlation result for maps that were generated using feature prediction but not pose buckets averages the correlation figures for all maps generated using either *ME85mod, ME88mod* or *K97mod* with feature prediction turned on and pose bucket disabled. Figures from maps generated by other three systems are not included in the results presented in section 6.10 as they do not use both feature prediction and pose buckets, and therefore to include them in some of the results in section 6.10 and not in others would distort the findings.

6.8 Ideal Maps

Below in Fig 6.4 are four occupancy grid maps based on the environments presented in Fig 6.1. The colour key is as follows:

- White squares are freespace. During experimentation care was taken that the robot came within 2.5 metres of all freespace areas in order to ensure that all areas of the map were detected. Were this not done, maps generate from test runs with reduced coverage would receive a worse benchmark score than maps generated from test runs in which the covered every section of the map fully.
- Black squares are obstacles.
- Grey squares are unknown areas. During the test runs the robot should never have detected any of the unknown areas, and therefore the maps generated from the test runs that changed the values from their default value 0.5 were penalised during benchmarking.
- The graphs contained in the freespace areas are the Voronoi graphs of the freespace areas of the map, and therefore are not actual physical features that the robot could detect.

These ideal maps were used for two purposes. The first is as an environment for the Pioneer simulator, where the robot is simulated traversing each of these environments. The second is as an ideal map on which to base all five benchmarking techniques described in chapter five – i.e. these are the maps that, if a mapping method were to produce them after a test run, would receive a *Correlation* of 100%, and both *Match* scores would be zero (i.e. no difference between maps).

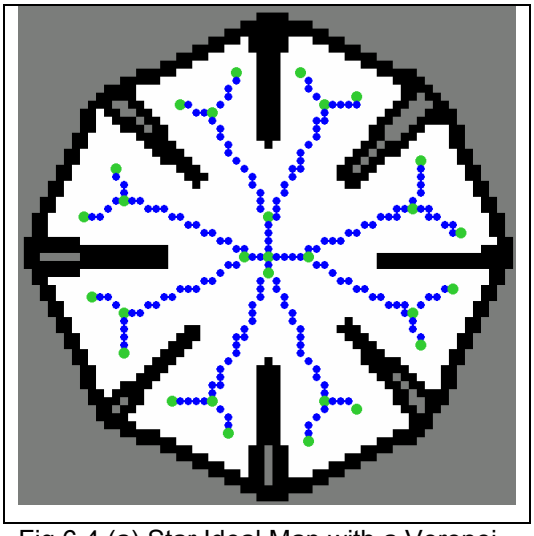


Fig 6.4 (a) Star Ideal Map with a Voronoi graph.

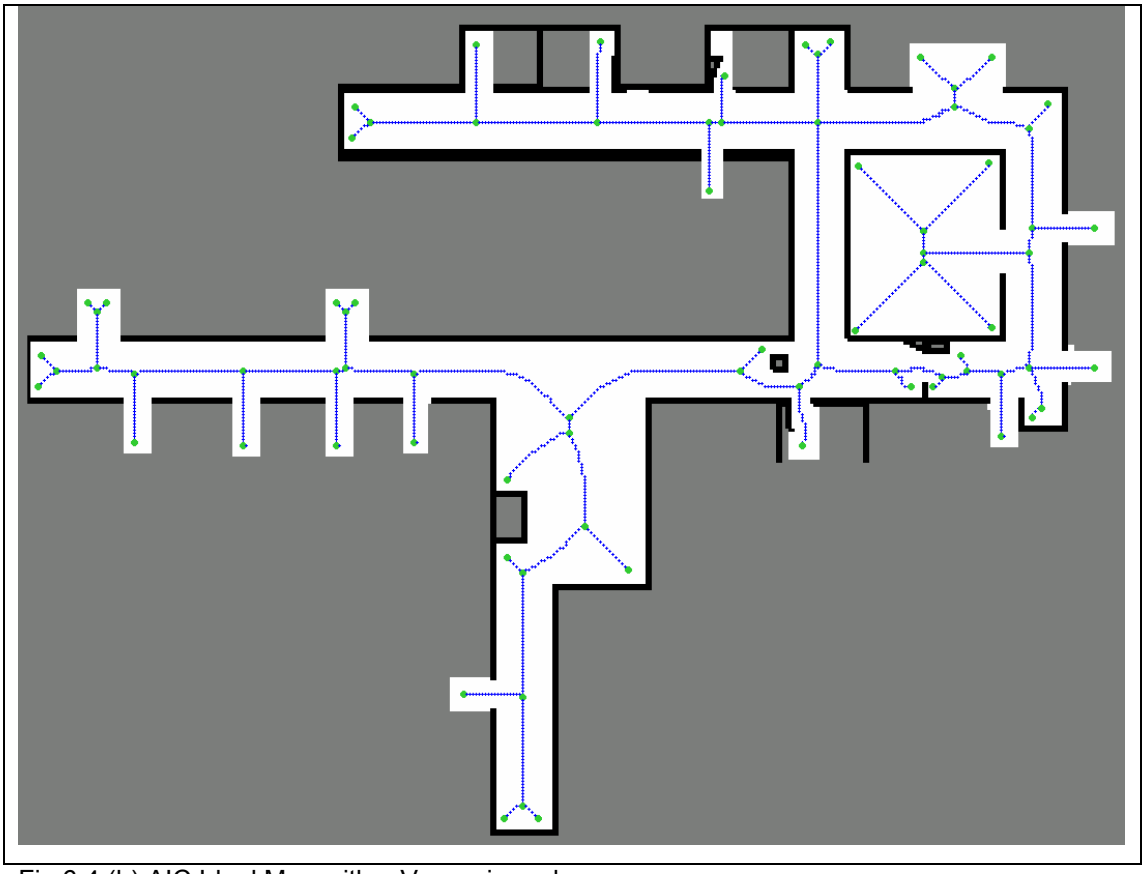


Fig 6.4 (b) AIC Ideal Map with a Voronoi graph.

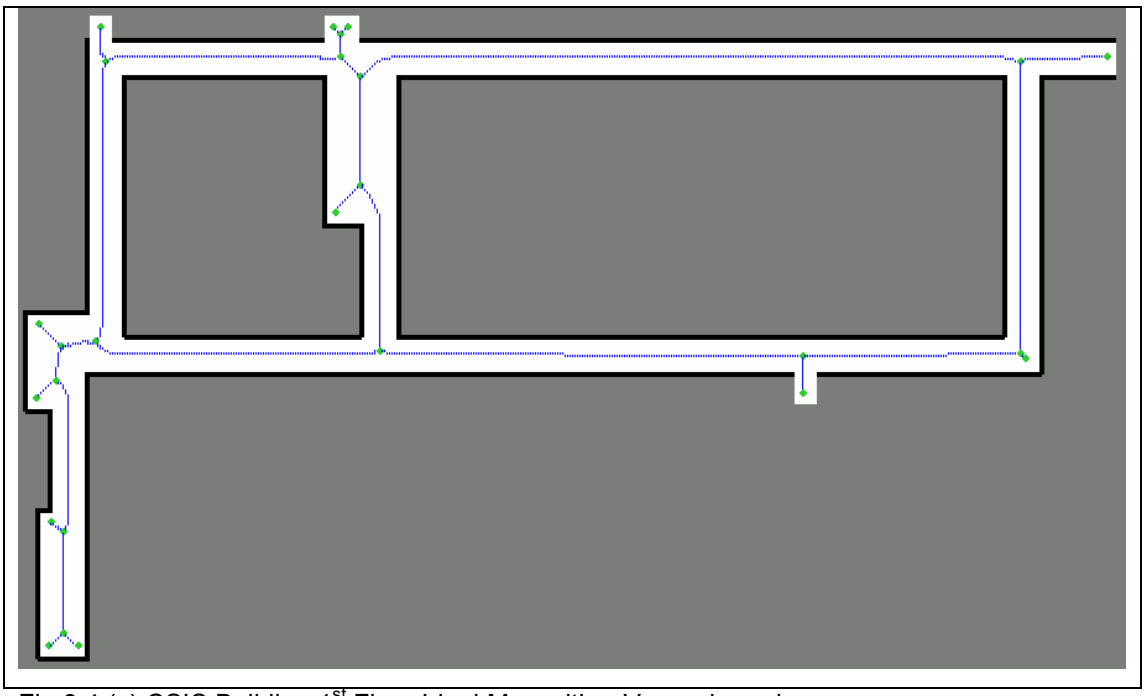


Fig 6.4 (c) CSIS Building 1st Floor Ideal Map with a Voronoi graph.

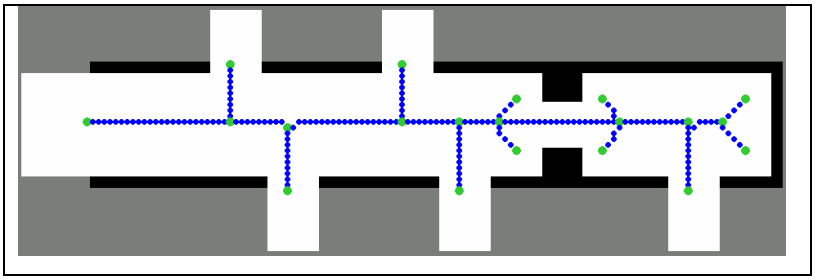


Fig 6.4 (d) Corridor Ideal Map with a Voronoi graph.

6.9 Comparison Of Sonar Models And Mathematical Update Strategies, With Neither Pose Buckets Nor Feature Prediction

A number of mathematical update strategies and sonar models are discussed in this thesis. The mathematical update methods tested and presented here include *ad hoc* formulae [46], a simple Bayesian update [42], as well as using log odds in association with Bayesian formulae [33]. Alterations to these formulae were presented in Chapter 4, and are the results of experiments with them are presented here with the mapping systems *ME85mod, ME88mod* and *K97mod*.

Two sonar models are tested, the two-dimensional gaussian from [46] which is used in the mapping systems *ME85, ME88, ME85mod* and *ME88mod*, and the sonar model from [33] that is used in *K97* and *K97mod*.

This section merely tests the sonar models and the mathematical formulae used to build maps, leaving out the use of Pose Buckets and Feature Prediction for filtering noisy specular readings. Of particular interest are the following questions.

1. Is there any advantage to using a simple Bayesian update over *ad hoc* update methods, other than mathematical elegance?
2. Do the changes to the update procedures and biases introduced in *ME85mod* and *ME88mod* offer any improvements over the systems upon which they are based, *ME85* and *ME88* respectively.
3. Is the sonar model put forward by Konolige in [33] any better than the gaussian sonar model from Moravec and Elfes' paper [46]?

Both simulated and real world results are presented in each section. The simulated results are based on data collected from robot runs in the four simulated environments introduced earlier in the chapter. The real world results are based on data collected from three robot runs in a single environment, the *Star* environment shown earlier in the chapter.

Due to the fact that the simulated results are far more extensive than the real-world results, they are the primary source from which conclusions are drawn. The real-world results aim merely to show that trends observable in the simulated data, for example the *ME88mod* performs better than *ME88*, are also present in the real world.

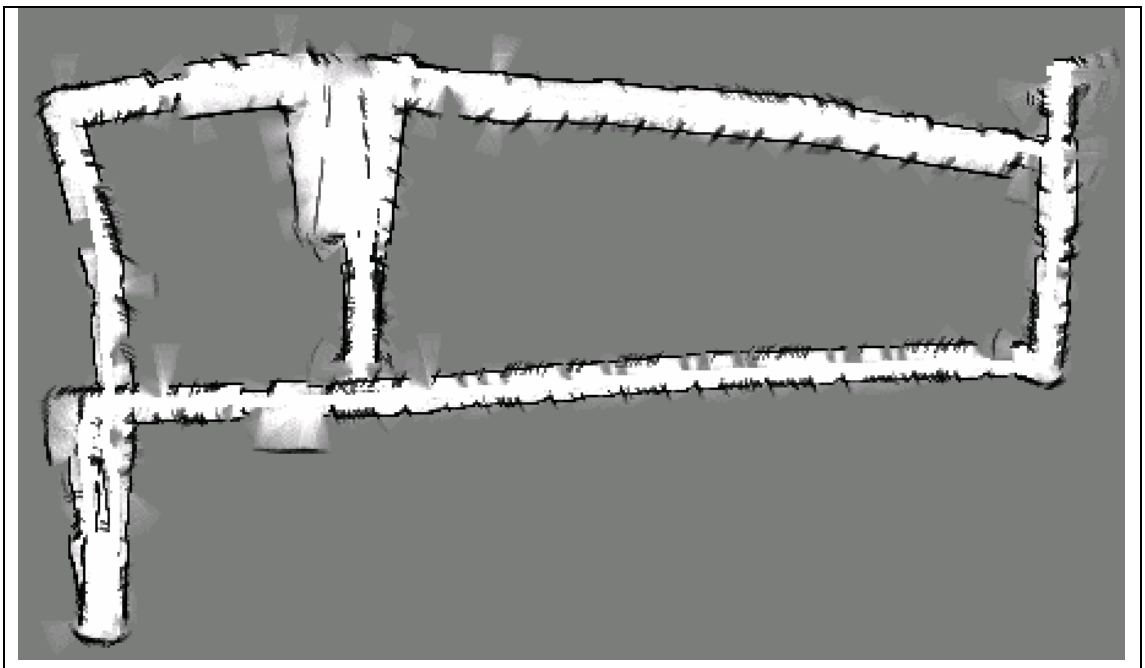


Fig 6.5 Map of the CSIS building produced by the *K97mod* map building system using both feature prediction and pose buckets

The primary reason for using a small environment for observing similar trends in the data, as opposed to performing extensive tests on large environments is that the benchmarking techniques applied to the results are reliant on the estimated position of the robot being accurate. In real-world robotics this involves performing Simultaneous Localisation and Mapping (SLAM). However, localisation is beyond the scope of this thesis, meaning that the odometry errors inherent in a robots' movement can significantly alter the quality of the map produced. For example, a single run around the real-world first floor of the CSIS building, which was simulated by the *eCSBsim* environment in the simulated experiments, produced the map below.

As can be seen, odometry errors have caused the map to become very distorted, and therefore difficult to accurately compare to an ideal version of the map.

Over short distances, however, odometry errors can be minimised, as there is insufficient time for errors to accumulate. For this reason it is possible to quite accurately map small environments and compare the resultant map to an ideal map for benchmarking. Three real-world test runs were performed on the *Star* environment, which is sufficiently small so that the errors caused by odometry inaccuracies is not very great.

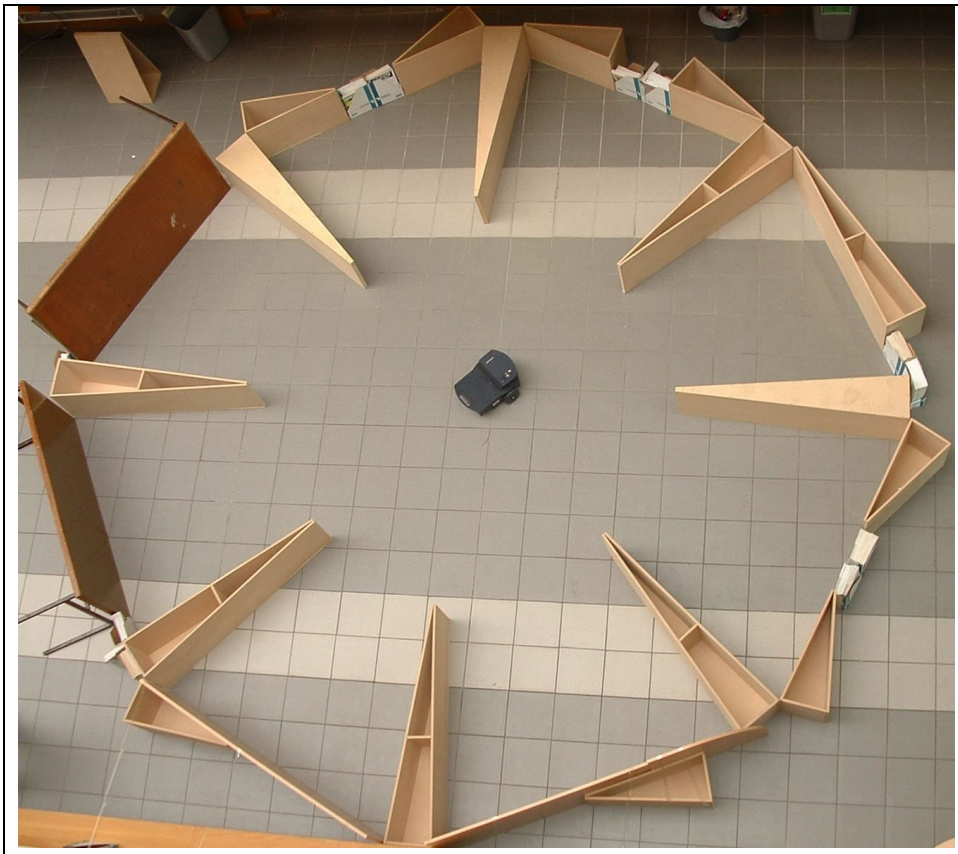


Fig 6.6 The Pioneer robot in the Star world test environment.

Above can be seen the robot in the *Star* environment. As it desirable for the generated map to as comparable as possible to the map generated in the simulated run, the robot was driven by hand in the same path that the robot took in the three simulated test runs. The goal here is to show that, while the real-world results will differ from the simulated results, the same trends can be observed in both, for example that pose buckets improve the quality of a map or that *K97* usually scores better than *ME85* etc.

6.9.1 Correlation with Ideal Map

Simulated Results

Mapping Method	Correlation
K97	70.48%
K97mod	63.28%
ME85mod	61.56%
ME88mod	60.31%
ME85	54.52%
ME88	52.82%

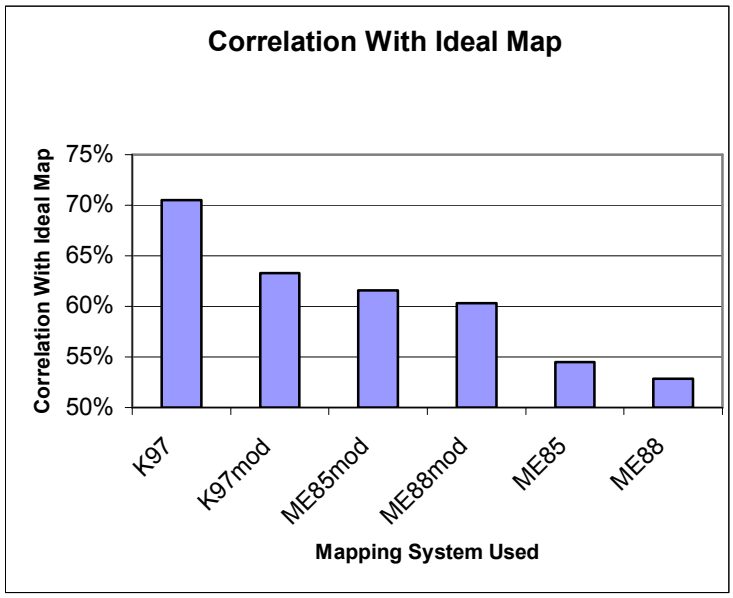


Fig 6.7 Correlation between the generated maps and the ideal maps. Correlation measures the similarity between maps, therefore a higher percentage signifies a greater similarity between maps than a lower percentage does.

In the results of the correlation between the generated maps and the ideal map, a higher degree of correlation reflects a higher level of similarity. The two sonar models based on work by Konolige achieved the first and second best percentages in this test, with the basic version, *K97*, performing significantly better than the modified version, *K97mod*. This can be attributed to the fact that *K97mod* updates freespace areas more strongly than *K97*, and in the absence of pose buckets and feature prediction this leads to non-detection of obstacles. It must be noted however that while *K97*'s average correlation value was a great deal higher than all the others, when faced with the very noisy *Star* environment, it performed significantly worse, getting a correlation value of 45.94%, as opposed to 57.52% for *ME85mod*. This is because *K97* updates occupied areas quite strongly, and in the face of numerous specular readings it takes the over-conservative approach of believing a disputed cell to be occupied rather than unoccupied. Fig 6.8 displays the difference between the maps generated by *K97* and *ME85mod*, where it can clearly be seen the effect a significant percentage of specular readings can have on the maps generated.

ME85mod and *ME88mod*, the modified versions of *ME85* and *ME88*, performed better than their ancestors, as predicted, even without using pose buckets or feature prediction. Interestingly, *ME85* and *ME85mod* performed better than their Bayesian based siblings using a simple *ad hoc* cell update method, with little mathematical validity.

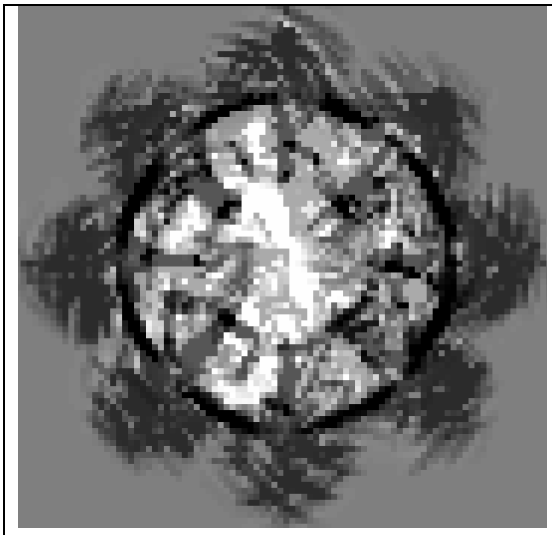


Fig 6.8 (a) The map of the *eSTARsim* environment produced by *K97* when no pose buckets are used. It's bias towards marking cells as occupied is readily apparent.

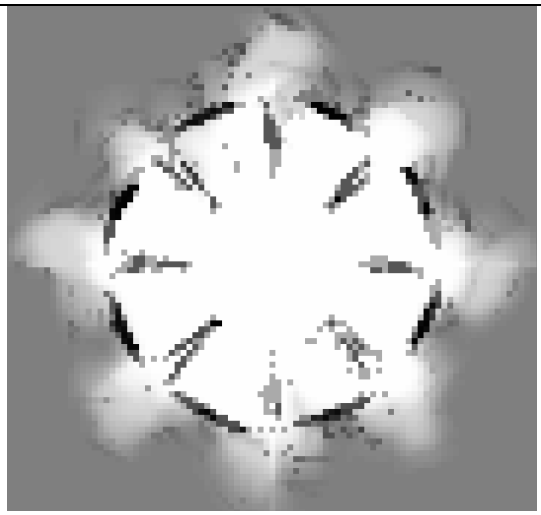


Fig 6.8 (b) The map of the *eSTARsim* environment by *ME85mod* when no pose buckets or feature prediction is used. The free space areas are far more accurately defined than the map generate by *K97*.

Fig 6.9 shows that in environments where there are relatively few specular readings, at least in comparison to the highly specular *eSTARsim* environment, the *K97* mapping system performs considerably better than the *ME85mod* mapping system, as well as all other mapping systems tested.

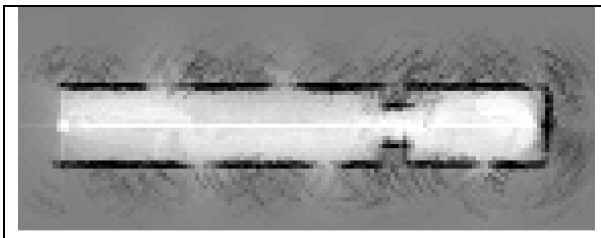


Fig 6.9 (a) The map produced by *K97* without pose buckets of the *eCORRsim* environment. Note that it is much more accurate, due to the lower degree of specularity of the environment in comparison with the *eSTARsim* environment.

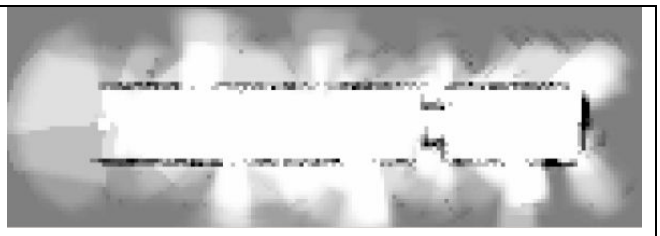


Fig 6.9 (b) The map produced by *ME85mod* of the *eCORRsim* environment without pose buckets or feature prediction.

Real-World Results

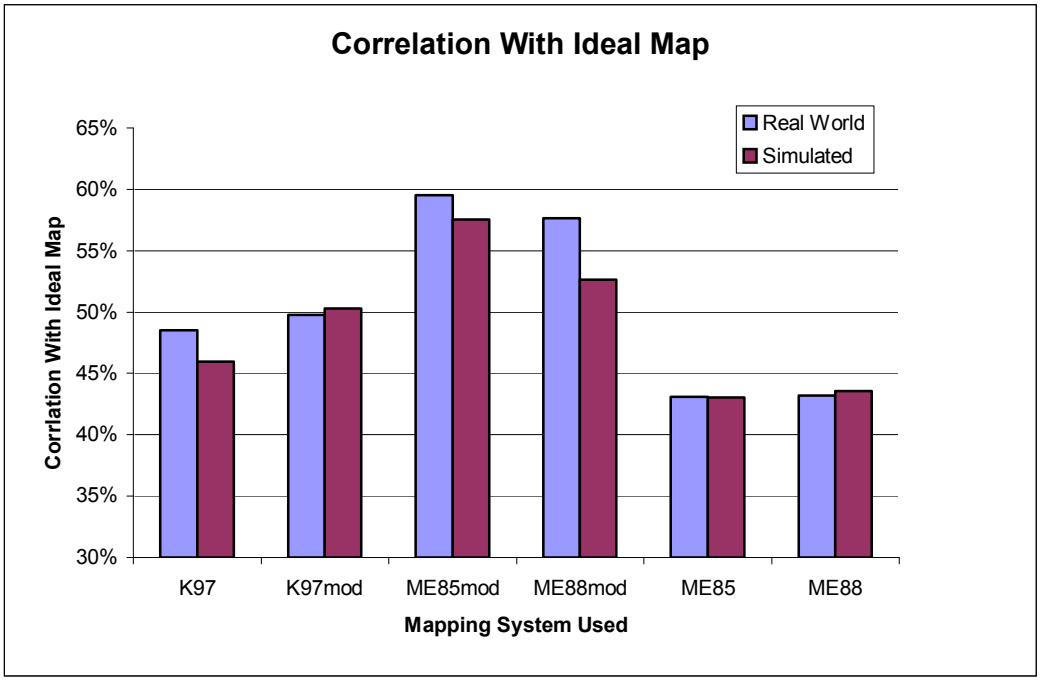


Fig 6.10 Comparison of the Correlation results for the simulated *Star* environment and the real-world *Star* environment

The figure above shows the correlation values for the various map building systems in the *Star* environment. It can be clearly seen that the trends in the simulated data are also present in the real-world data, though the actual values are different. This is inevitable, as odometry errors cause differences between the simulated environment and the real world measurements. The difficulty in simulating the sonar readings received by the robot exactly also leads to difference between the simulated and real-world results.

6.9.2 Match All Cells Between Generated Maps and the Ideal Map

Simulated Results

Mapping Method	Match All Cells
K97	10.33%
K97mod	12.11%
ME85mod	12.96%
ME88mod	13.13%
ME88	16.77%
ME85	16.78%

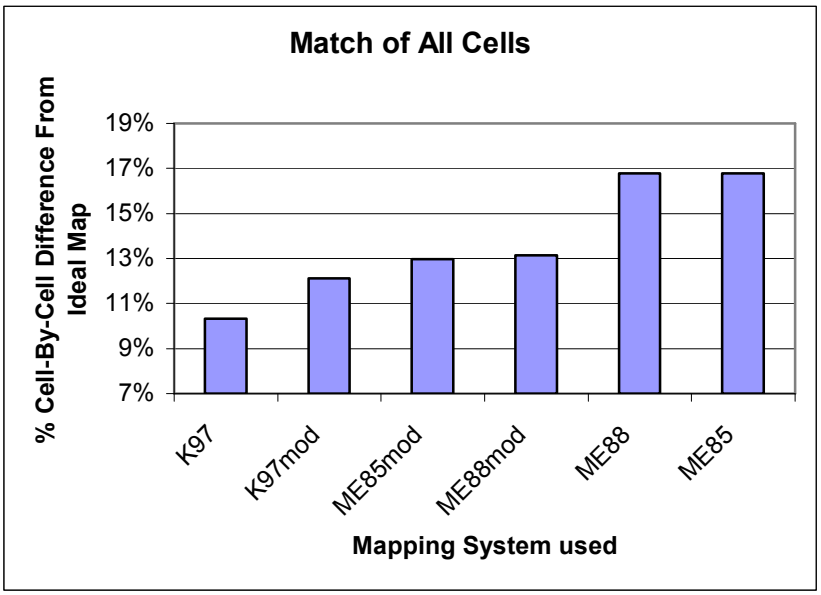


Fig 6.11 The *Match All Cells* benchmark measures the cell-by-cell squared difference between two maps, therefore the lower the percentage, the more similar the two maps are.

When comparing the squared difference between the complete generated map and the ideal map, once again *K97* performs best, achieving the lowest percentage difference from the ideal, with *K97mod* again coming second for the same reasons as discussed above in the correlation section. As with the correlation value for *K97* however, when faced with the very noisy *Star* environment *K97* performs very poorly, achieving difference of 24.08% from ideal as opposed to 18.03% for *ME85mod*, the best score on that map.

ME85mod and *ME88mod* perform better than their ancestors *ME85* and *ME88* respectively. This improvement is a result of the improved dynamic mixture model, as well as alterations of the mathematical update procedure in *ME85mod*. However, both *ME85mod* and *ME88mod* perform more or less identically, again calling into question the need for Bayesian update rules over simple *ad hoc* probability update procedures.

Real-World Results

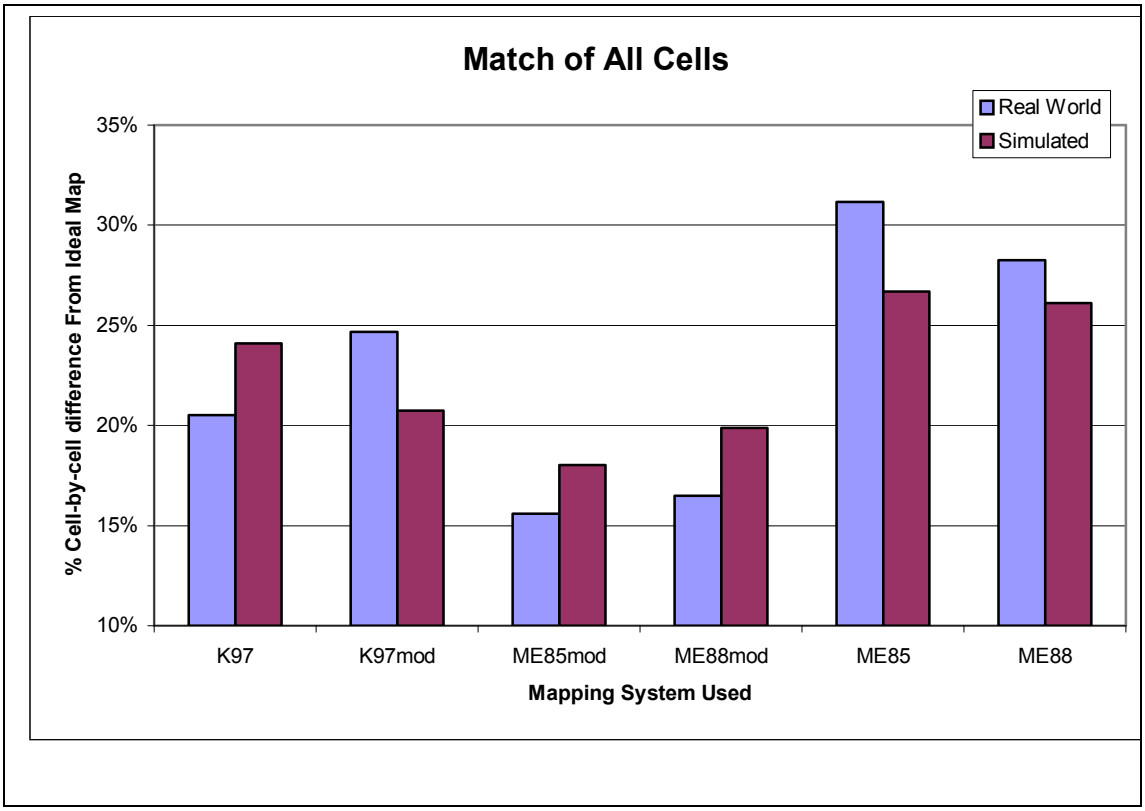


Fig 6.12 Comparison of simulated and real-world results for the percentage of the cell-by-cell difference from the ideal map of the *Star* environment as a percentage of the difference between the worst possible map and the ideal map.

The figure above shows that the overall trend is the same between the simulated and real-world results. *ME85mod* performs better than *ME88mod*, *ME88* performs better than *ME85*. *ME85mod* and *ME88mod* perform better than *ME85* and *ME88* respectively, with *K97* and *K97mod* being somewhere in between.

The only change is in the difference between *K97* and *K97mod*. In the simulated results *K97mod* performs better, but in the real-world results *K97* performs better. This can largely be attributed to difficulty of accurately simulate the behaviour of a sonar beam in complicated environments. Whereas the simulator works very well in structured environments with 90^{o} angles, in a strangely shaped environment such as the *Star* world it returns sonar readings that are too short or too long. This, allied with *K97*'s tendency to update occupied cells too strongly, caused many freespace cells to marked as occupied in the simulated results. *K97mod* takes a more conservative approach when updating occupied cells, and therefore performed better in the simulated results. The real-world experiments showed the sonar behaving slightly

differently to the simulated sonar. While many readings were still received that were longer than they should have been, due to specular reflection, few if any range readings were too short. This led to *K97* performing better since its dynamic mixture model helped filter out some specular readings, and it didn't incorrectly update as many freespace cells. *K97mod*, on the other hand, updates freespace areas more strongly than *K97*, and therefore incorrectly marked more occupied cells as being unoccupied, therefore performing worse.

6.9.3 Match of Occupied Cells Between Generated Maps and the Ideal Map

Simulated Results

Mapping Method	Match Occupied Cells
K97	8.48%
ME85mod	9.20%
K97mod	9.70%
ME88mod	10.15%
ME85	10.82%
ME88	12.53%

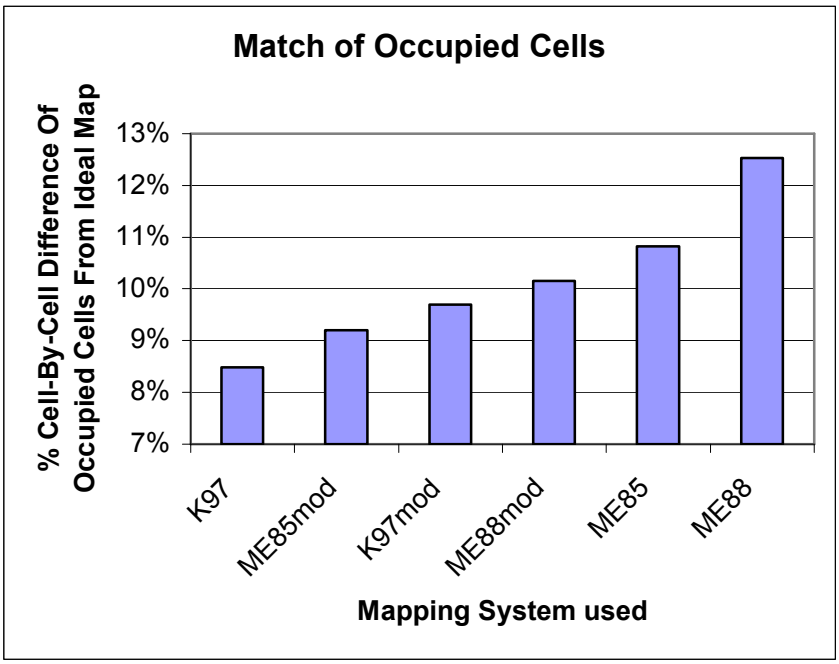


Fig 6.13 Match between the occupied cells in the generated maps and the ideal maps.

In calculating the ability of a mapping system to correctly identify obstacles, *K97* once again comes out on top, although it performed the worst of all six systems in the highly specular *Star* environment with a difference of 20.48% from the ideal, as opposed to the best score of 14.03% achieved by *ME85mod*.

Both *ME85mod* and *ME88mod* identified obstacles in the map much more effectively than either *ME85* or *ME88*, but surprisingly the Bayesian models *ME88mod* and *ME88* were vastly outperformed by their less mathematically elegant siblings *ME85mod* and *ME85* respectively. This resulted from the fact that the Bayesian update equation converges very swiftly to either a 0 or a 1 with only a few sonar readings, and once these end-points are reached they can never be. Other update methods, such as the one used in *ME85* and *K97*, require more readings before converging to a finished state, so a small number of incorrect specular readings can later be corrected by accurate diffuse readings, whereas the Bayesian update method converges to a 0 with just a few incorrect specular readings, and can never rise above that value once it is reached, no matter how many correct readings are received afterwards.

Real-World Results

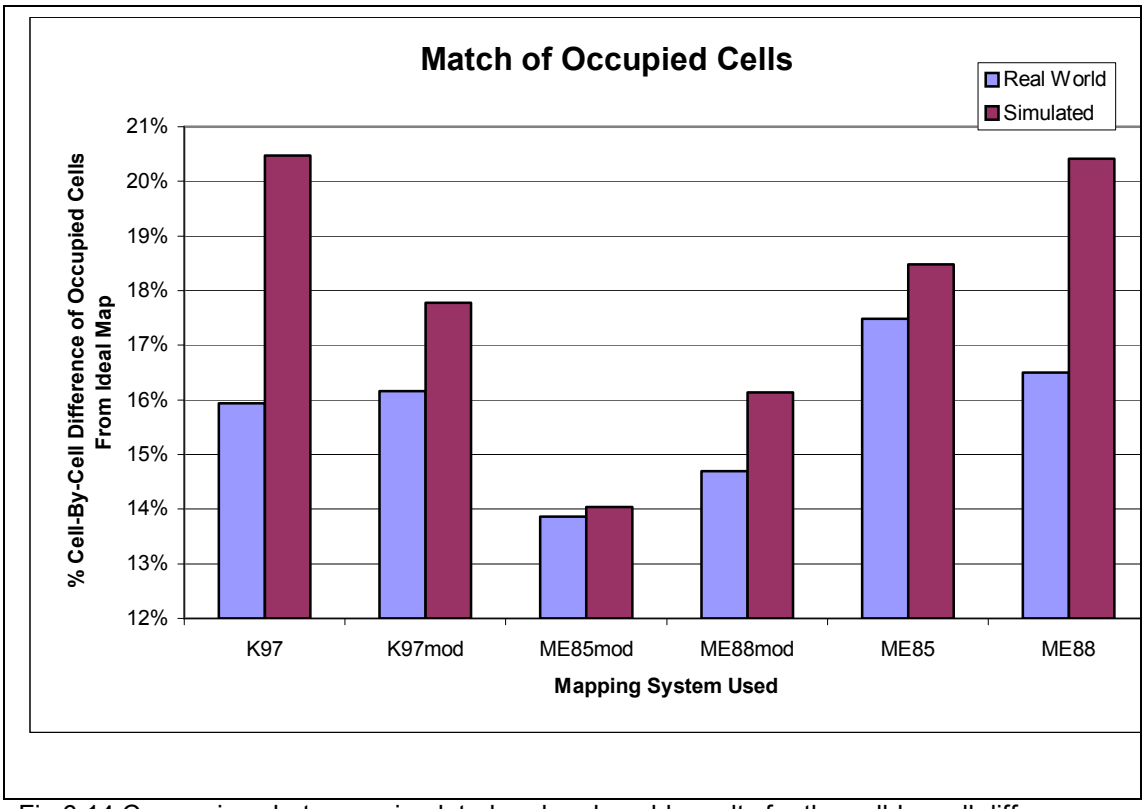


Fig 6.14 Comparison between simulated and real-world results for the cell-by-cell difference between the generated map and the ideal map, only taking into account the occupied cells in both maps.

The trends in the figure above are similar to those in the previous section, with *ME85mod* and *ME88mod* performing better than *ME85* and *ME88* respectively etc. Once again, *K97* sees a large increase in accuracy, but so does *K97mod*. In fact, the performance of all systems increases significantly over the simulated results. As in the previous test, this can attributed to the fact that there were few if any sonar readings that were too short, and therefore fewer freespace cells marked as occupied.

6.9.4 Percentage of False Positive Paths in Generated Maps

Simulated Results

Mapping Method	% False Positive Paths
K97	10.40%
ME85mod	61.57%
ME88mod	65.03%
ME88	72.06%
ME85	72.39%
K97mod	73.10%

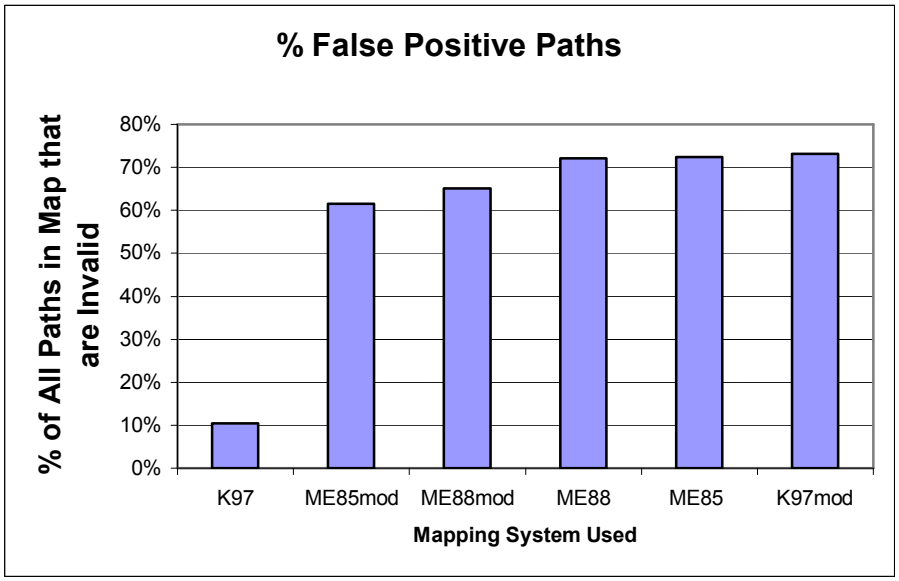


Fig 6.15 The percentage of false positive paths in the generated maps. These are paths that possible for the robot to traverse according to the generated map, but which in reality would cause it to collide with an obstacle.

As detailed in Chapter 5, this test of a map measures the safety of traversing paths generated using the map. A Voronoi graph of all possible paths in the map is created, which is then overlaid on the ideal map. If an edge of the Voronoi graph passes through an occupied cell in the ideal map, the complete edge is marked as invalid as it would cause the robot to crash were the robot to follow it.

Mapping systems that perform poorly in this test are those that tend to update freespace areas too strongly, causing paths to be plotted through occupied spaces. As such, *K97* came out on top of this test by a very large margin, having only 10.4% of it's paths passing through occupied spaces, whereas all other systems had over 60% of their paths being invalid.

ME85mod and *ME88mod* performed similarly to each other, and had fewer invalid paths than both *ME85* and *ME88*. In both cases this can be attributed to the fact that when one cell in a sonar beam claims the reading to be specular, all cells in the sonar beam are updated accordingly, rather than just that particular cell. Specular readings are thereby recognised and ignored more reliably, so fewer obstacles are marked as

freespace. *K97mod* performed the worst of all systems, which can be attributed to the fact that it strongly updates freespace areas, and without pose buckets or feature prediction it can miss many obstacles.

A problem with this test is that it is biased towards those systems that update freespace areas weakly, so systems that have far too many occupied cells in a map will seem to be very good. This is because if there are too many obstacles, very few paths will be generated as it will seem as if the robot cannot go anywhere in the map, as opposed to appearing as if the robot can go place where in reality it cannot. Therefore, if few paths are possible in the map being tested, it is far less likely that these paths will be invalid. Therefore a counter balance to this test is required to balance the deficits outlined here. Whereas this benchmark examines the freespace areas of a map, the fifth benchmark examines the occupied areas of a map. When these two benchmarks are used in conjunction, they provide an accurate fitness measure for a map.

Real-World Results

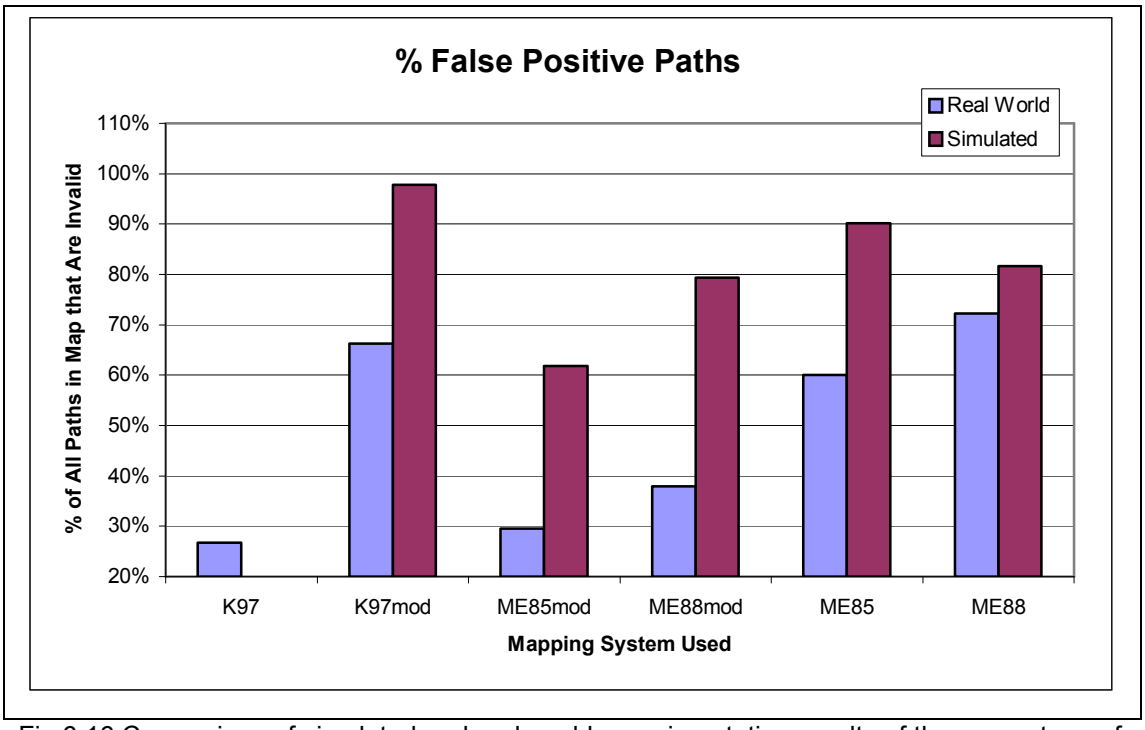


Fig 6.16 Comparison of simulated and real-world experimentation results of the percentage of all paths in the generated map that would cause a collision in the real world.

The trend of the real-world results follows the simulated results almost exactly, with the exception of *K97*. Whereas all other systems improve their performance, *K97* seems to perform worse. The reason for this, however, is that in the simulated *K97* map, almost all the freespace cells were marked as occupied, and therefore there were almost no paths that could be invalid, so while in this test the simulated *K97* seems to perform well, the next benchmark shows that this is not the case.

6.9.5 Percentage of False Negative Paths

Simulated Results

Mapping Method	% False Negative Paths
ME88mod	0.92500%
K97mod	0.92500%
ME85mod	0.92575%
ME88	0.95925%
ME85	0.95925%
K97	18.54685%

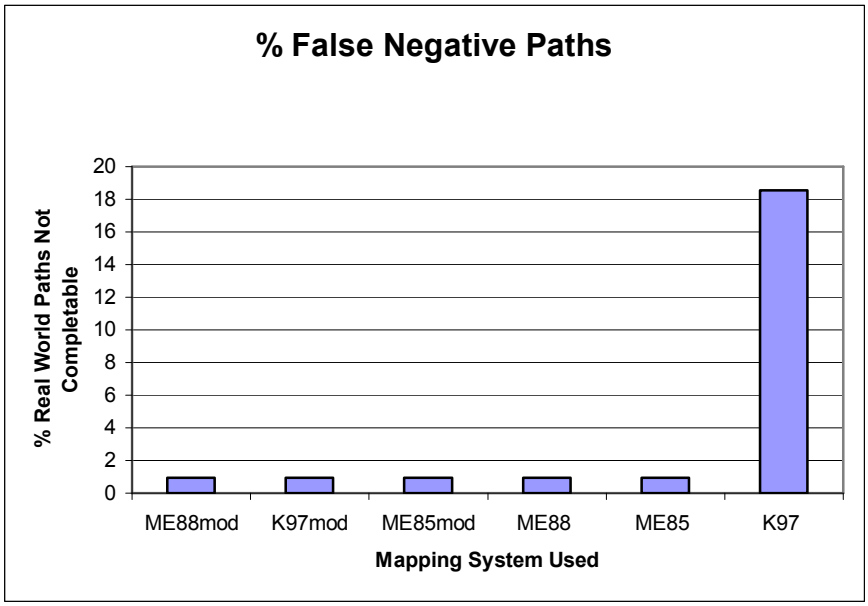


Fig 6.17 Percentage of false negative paths in the generated map. These are the paths that are possible to plot in the ideal map, but are not possible to complete in the generated map due to freespace regions being marked as occupied.

Whereas the previous test applied the question 'If the robot were to use this map to explore the environment, how safe would it be?', this test asks 'If the robot is required to move from one real-world position to another, would it be possible to use this map to plan the route?'. This benchmark presents the percentage of false negative paths in a map. These are paths can be completed in the ideal map, but not in the generated map. The only reason a path could not be completed in a map is if the mapping system has marked freespace areas as being occupied. This test is therefore essentially the inverse of the previous benchmark, which presented the percentage of false positive paths.

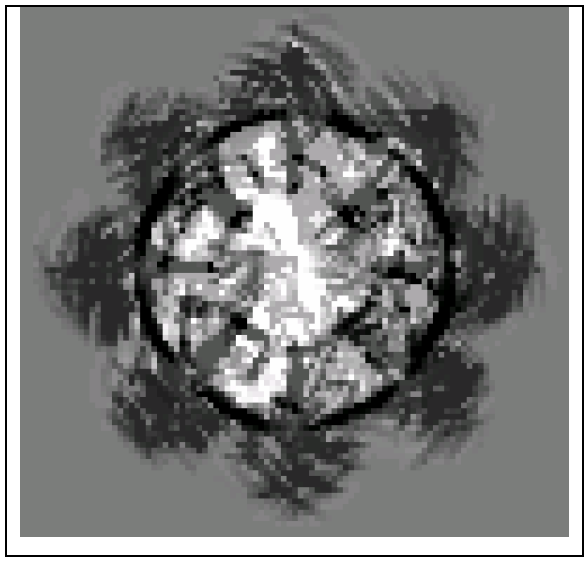


Fig 6.18 Map of the *eSTARsim* environment generated by the *K97* mapping system when pose bucket are not used.

All systems performed very similarly in this test, with the exception of *K97*. This can be attributed to the fact that *K97* takes a conservative approach towards specular readings, which means that it has a tendency to update occupied areas too strongly. In highly specular environment *K97* often marks many freespace cells as being occupied, making it difficult to plan a path using the map. The results above are skewed by this fact. In the three open corridor-like environments *K97* performed similarly to the other five mapping systems. Only in the very noisy *Star* environment did it perform spectacularly badly, giving the map on the right.

Real-World Results

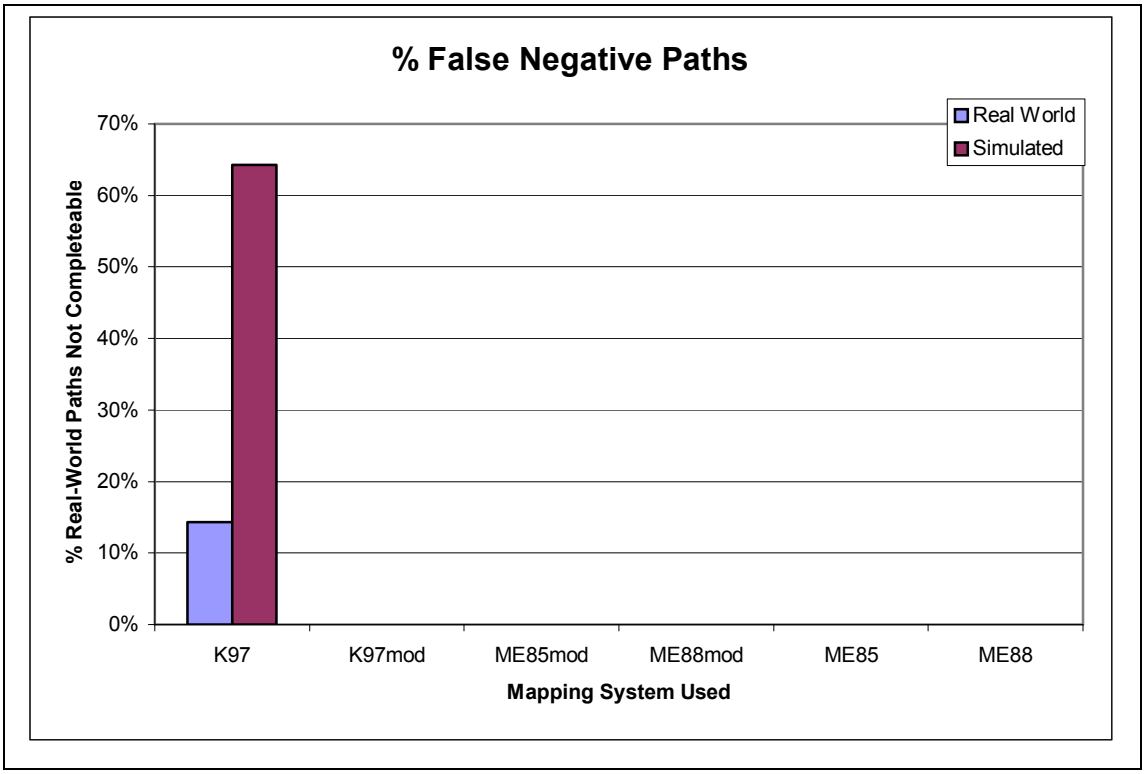


Fig 6.19 Comparison of the simulated and real world results of the percentage of paths from the real world that could not be completed in the generated map.

Both the simulated and real-world results follow the same pattern here. All systems except for *K97* have zero incomplete paths. This is largely due to the fact that they all overestimate the freespace areas in the environment, making it simple to complete a path. *K97*, on the other hand, overestimates the occupied areas of a map, and therefore in some areas it is impossible to plan a path between two points which, in the real world, it is possible to make a path through. The real world experiments show *K97* performing significantly better than in the simulated results, and for the

same reason as previously mentioned – the simulator finds it difficult to model the behaviour of a sonar beam in complicated environments like the *Star* world.

6.9.6 Evaluation of Results 1

The three questions being answered by the above series of tests are:

1. Is there any advantage to using a simple Bayesian update over *ad hoc* update methods, other than mathematical elegance?
2. Do the changes to the update procedures and biases introduced in *ME85mod* and *ME88mod* offer any improvements over the systems upon which they are based, *ME85* and *ME88* respectively.
3. Is the sonar model put forward by Konolige in [33] any better than the gaussian sonar model from Moravec and Elfes' paper [46]?

6.9.6.1 Question 1

As to question 1, in all the tests above the systems using an *ad hoc* probability update procedure, *ME85* and *ME85mod*, performed at least as well as their respective siblings, *ME88* and *ME88mod*, and sometimes better. All four systems were given the same gaussian sonar model, with *ME85* and *ME88* being identical except for the Bayesian update, and *ME85mod* and *ME88mod* being identical except for the Bayesian update. While both *K97* and *K97mod* use a modified Bayesian update procedure, and perform best of all the systems being tested, they also use a significantly different sonar model and therefore cannot be compared directly with the other systems on the basis of just the probability update mathematics. The only conclusion that can be drawn from these results is that the Bayesian update procedure offers no tangible performance enhancement over the simple mathematical update formulae used in *ME85* and *ME85mod*.

This result flies in the face of all research published since the mid 1980's in the field of probabilistic mapping, but the results are clear. While using Bayesian logic as a foundation for developing update strategies leads to mathematically robust formulae, the direct application of the most basic Bayesian probability update formula,

$$P(OCC \mid R) = \frac{P(R \mid OCC)P(OCC)}{P(R \mid OCC)P(OCC) + (1 - P(R \mid OCC))(1 - P(OCC))}$$

offers no performance improvement over the *ad hoc* update developed by Moravec and Elfes [46] and described in chapter 2.

6.9.6.2 Question 2

The second question posed was whether or not the modifications in *ME85mod* and *ME88mod* offered any performance improvements over *ME85* and *ME88*. These modifications included, in the case of *ME85mod*, removing the bias towards believing freespace readings over surface readings, and in the case of both *ME85mod* and *ME88mod*, using an improved dynamic mixture model to better compensate for specular readings using previously collected data.

In each of the tests, *ME85mod* and *ME88mod* performed better than their ancestors, usually much better, showing that their ability to better identify erroneous readings (an enhanced version of Konolige's Dynamic Mixture Model [33]) gives them a large performance boost, creating much more accurate maps.

6.9.6.3 Question 3

The third question was whether or not Konolige's sonar model performed any better than the simple gaussian model from Moravec and Elfes' 1985 paper [46]. In all of the tests save the last, *K97* performed the best of all systems. However it's overly conservative approach of updating occupied cells is a weakness in very noisy environments, as seen in the fifth test. *K97mod* also performed well, coming in the top two in most tests, and performing far better than *K97* in highly specular areas.

6.10 Comparison Of The Contribution of Pose Buckets and Feature Prediction To The Accuracy Of A Map

In order to evaluate the benefits of performing Feature Prediction to identify and discard noisy readings, as well as using Pose Buckets to ensure the independence of consecutive sonar readings, experiments were carried out on all simulated environments with all mapping systems that use both of these enhancements. These include *ME85mod, ME88mod,* and *K97mod* which use both Feature Prediction and

Pose Buckets. Experiments were also carried out in the real-world *Star* environment using the same three mapping systems.

The tests applied to the maps are the same as in section 6.9; the maps are tested against the ideal hand-drawn map using

- Correlation.
- Match All Cells in the generated against the ideal map.
- Match Occupied Cells in generated and ideal maps.
- Finding the percentage of false positive paths in the generated map.
- Finding the percentage of false negative paths in the generated maps.

The average of the results from all maps is shown in each test, with the performance of the systems:

- With neither feature prediction nor pose buckets.
- With pose buckets but no feature prediction.
- With feature prediction but no pose buckets.
- With both pose buckets and feature prediction.

Only the results from the systems *ME85mod*, *ME88mod* and *K97mod* are included in these results because only these three systems use both pose buckets and feature prediction, and it is desirable to observe how they behave with and without both of these enhancements. The results from *ME85*, *ME88* and *K97* are not included because do not use both pose buckets and feature prediction and should therefore not be included as part of the baseline benchmark.

6.10.1 Correlation with Ideal Map

Simulated Results

Mapping Enhancements	Correlation
No PB, No FP	61.72%
With PB, No FP	75.92%
With FP, No PB	78.45%
With FP, With PB	81.65%

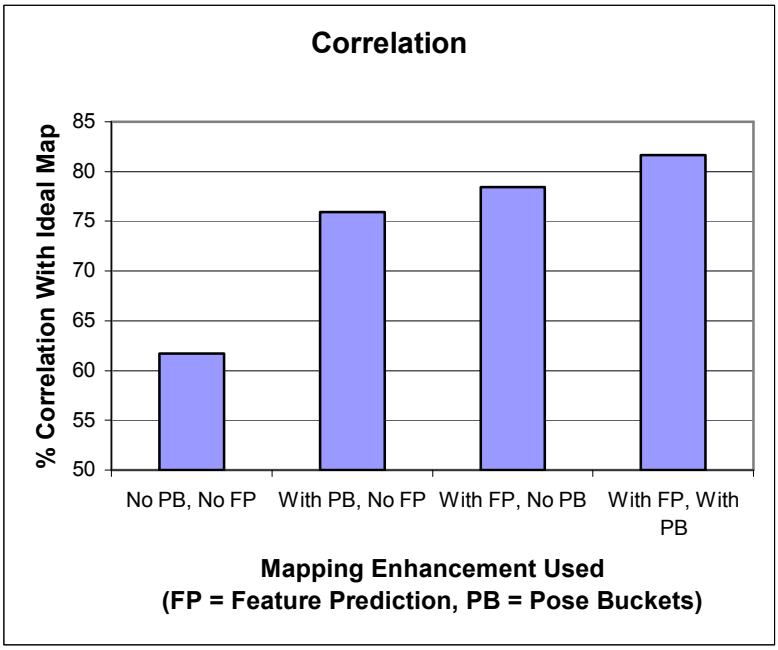


Fig 6.20 Average Correlation of all maps generated by *ME85mod, ME88mod* and *K97mod*, grouped by their use of feature prediction and pose buckets. A higher correlation value indicated a higher degree of similarity between two maps.

The performance of the mapping systems when using neither feature prediction (FP) nor pose buckets (PB) is quite poor, with a low correlation of 61% with the ideal map. A large improvement in the quality of the map, at least in terms of correlation, is observed as soon as redundant readings are removed by using pose buckets. While pose buckets treat both freespace and obstacle readings equally, the fact that there are generally many more specular freespace readings than correct obstacle readings means that many more erroneous readings are ignored than correct readings, leading to a better map being generated.

An even larger improvement in map correlation is achieved by using feature prediction. Whereas pose buckets improve the map quality by virtue of the fact that there are more erroneous readings than non-noisy readings, feature prediction directly targets noisy specular readings and removes them, and at the same time has little or no effect on correct readings.

Using both feature prediction and pose buckets together yields even more impressive results, showing that while they overlap in some of the incorrect readings they discard, where pose buckets may mistakenly believe a noisy reading, feature prediction will recognise and discard it, and vice versa.

Real-World Results

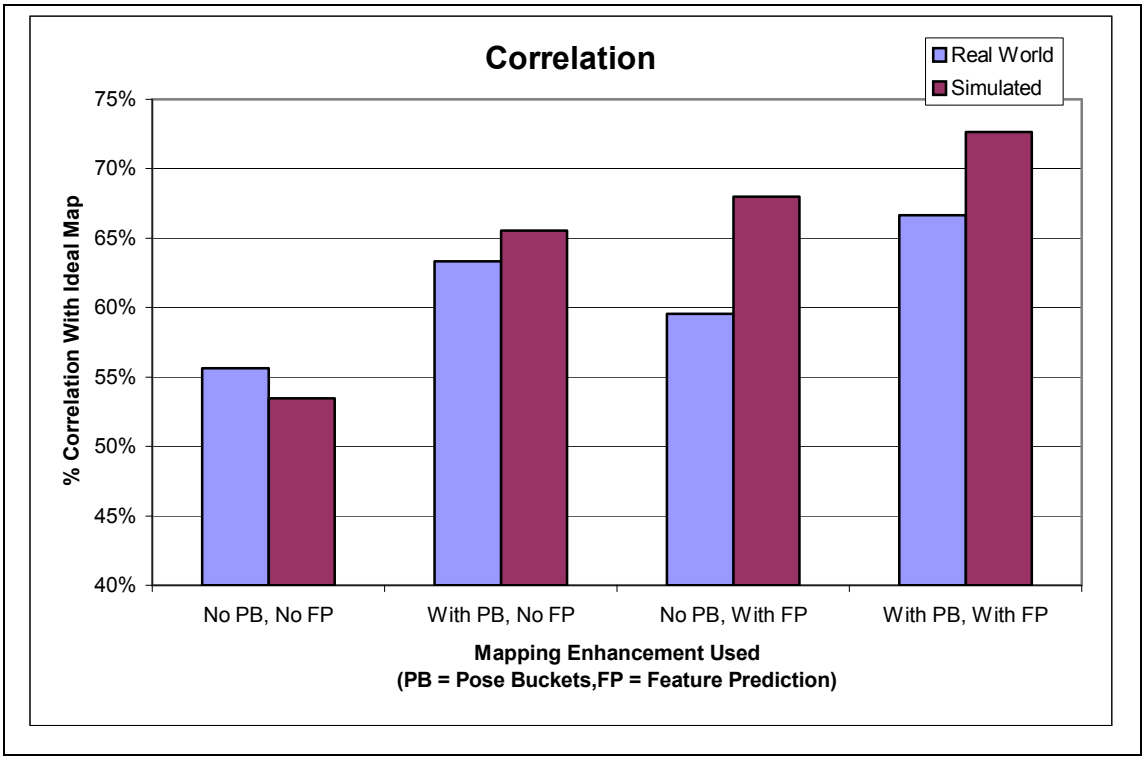


Fig 6.21 Comparison of the simulated and real-world results of the correlation between the generated map and the ideal map of the *eSTAR* environment.

The simulated and real-world results follow a similar pattern. They both show that Pose Buckets and Feature Prediction improve the quality of the map, even in such an irregular environment prone to specular reflections as the *eSTAR* world. They also both show that the best results of all are obtained by using a combination of Pose Buckets and Feature Prediction.

6.10.2 Match All Cells Between Generated Maps and the Ideal Map

Simulated Results

Mapping Enhancements	Match All Map
No PB, No FP	12.73%
With PB, No FP	7.31%
With FP, No PB	6.68%
With FP,With PB	5.60%

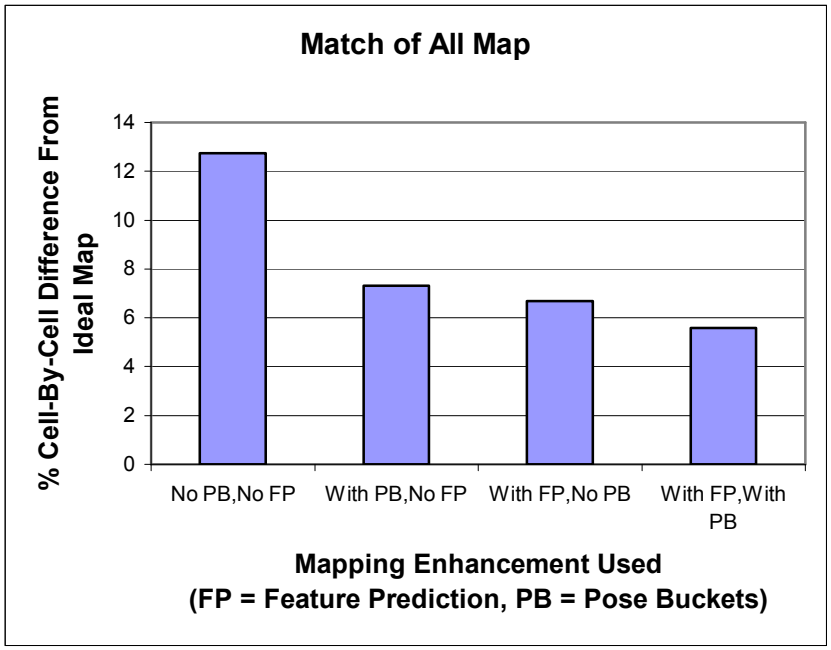


Fig 6.22 Average Match of all cells in all maps generated by *ME85mod, ME88mod* and *K97mod*, grouped by their use of feature prediction and pose buckets.

As with in correlation test, when deprived of pose buckets or feature prediction to remove specular readings, the mapping systems perform very poorly, with an average difference of almost 13% from the ideal map. This means that for each cell in a map with a value in the range [0,1], all cells are on average 0.13 from the value they should be. However this figure is merely an average. Given that many cells are much closer to their ideal value than 0.13, many cells can often be 0.6 or 0.7 from the ideal value. This means that many occupied cells are marked as empty, and vice versa.

Once pose buckets are introduced to the mapping process the error drops dramatically since many incorrect readings are discarded, along with a smaller number of correct readings. Feature prediction improves matters to a greater degree than pose buckets, once again discarding many incorrect specular readings, but keeping more correct readings than pose buckets.

One again, feature prediction and pose buckets prove to be complimentary to each other, and using them together gives a further improvement in map quality. As

explained in the correlation section, where feature prediction may mistake an incorrect reading for a correct one, pose buckets sometimes discard it, and vice versa.

Real-World Results

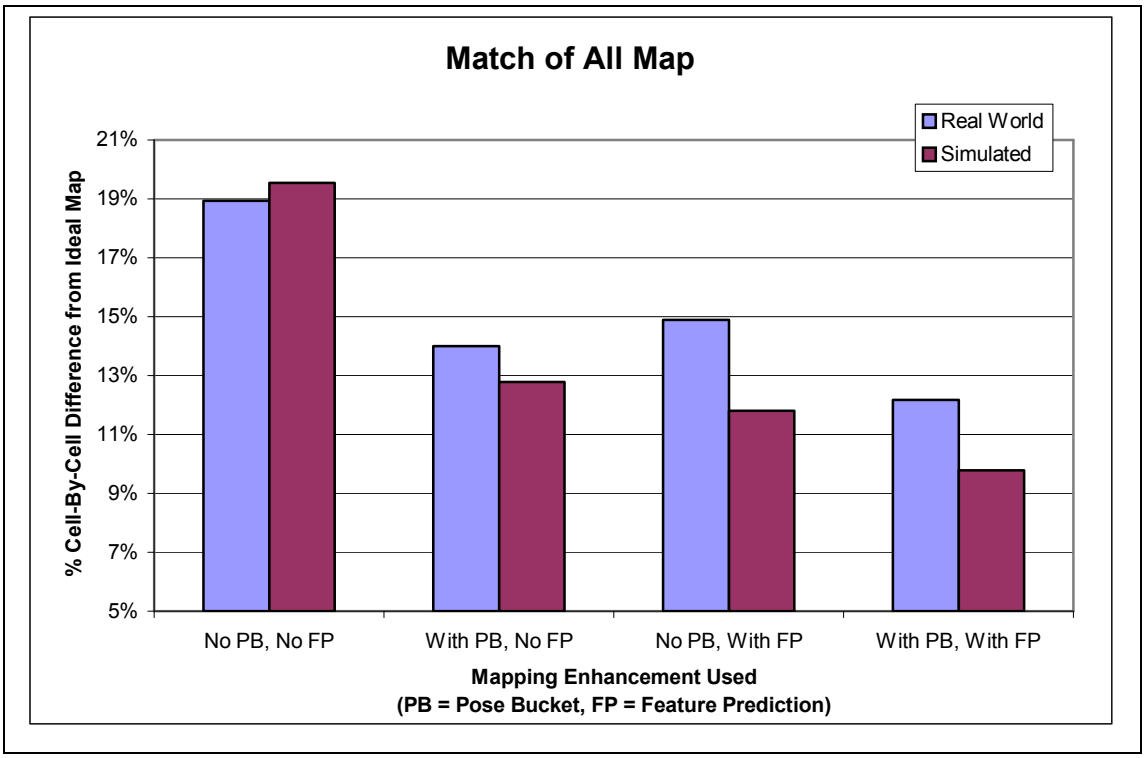


Fig 6.23 Comparison of simulated and real-world results for the percentage cell-by-cell difference between the generated map and the ideal map of the *eSTAR* environment.

The simulated and real-world results follow very similar patterns, with both the use of Pose Buckets and Feature Prediction improving the quality of the maps (albeit by slightly different amounts), and using both together giving the best map of all, as with the Correlation result prior to this.

6.10.3 Match of Occupied Cells Between Generated Maps and the Ideal Map

Simulated Results

Mapping Enhancements	Match Occupied Cells
No PB, No FP	9.66%
With PB, No FP	5.94%
With FP,No PB	5.62%
With FP,With PB	4.38%

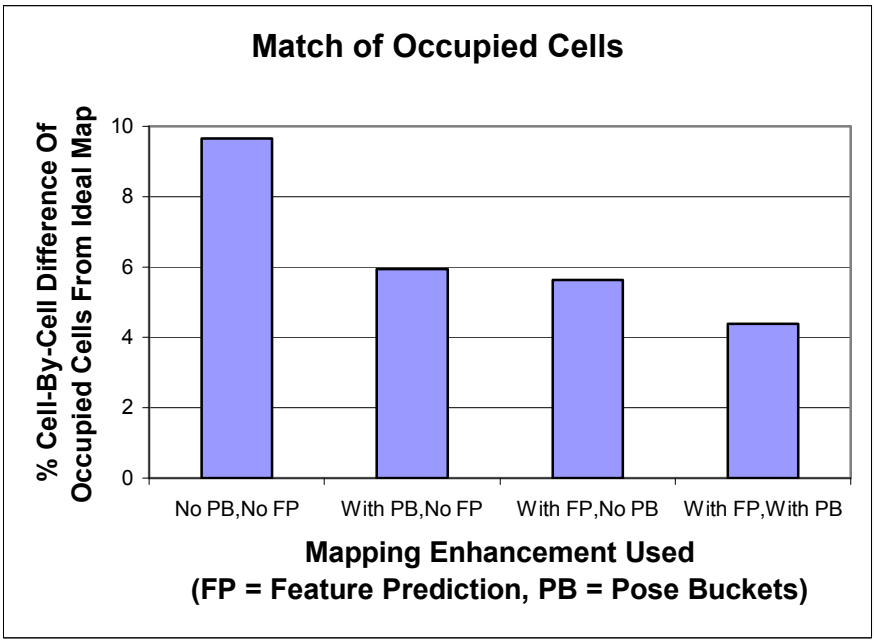


Fig 6.24 Average Match of occupied cells in all maps generated by *ME85mod, ME88mod* and *K97mod*, grouped by their use of feature prediction and pose buckets.

As in the previous two tests, when the mapping systems integrate the sonar information into the map without pose buckets or feature prediction, they miss many of the obstacles in the map and believe that obstacles exist where in fact they don't.

Using pose buckets immediately cuts down the obstacle-detection error by a large margin, almost halving it. From inspection of the maps created just using pose buckets, it can be seen that many more obstacles are correctly identified, with less 'bleeding' of freespace areas into occupied areas. However there are still many 'sonar shadows' behind the walls, which account for much of the 5.94% error.

Feature prediction offers a similar performance gain to pose buckets, though for a different reason. While feature prediction does lead to many more obstacles being correctly identified, it also cuts the number of sonar shadows in the map drastically by discarding or weakening readings that report obstacles farther away than they are. The fact that it doesn't give a significant improvement in this test over pose buckets leads one to believe that it misses more real obstacles than pose buckets.

When used together, FP and PB once again complement each other, correctly identifying twice as many obstacles as the mapping systems would without them.

Real-World Results

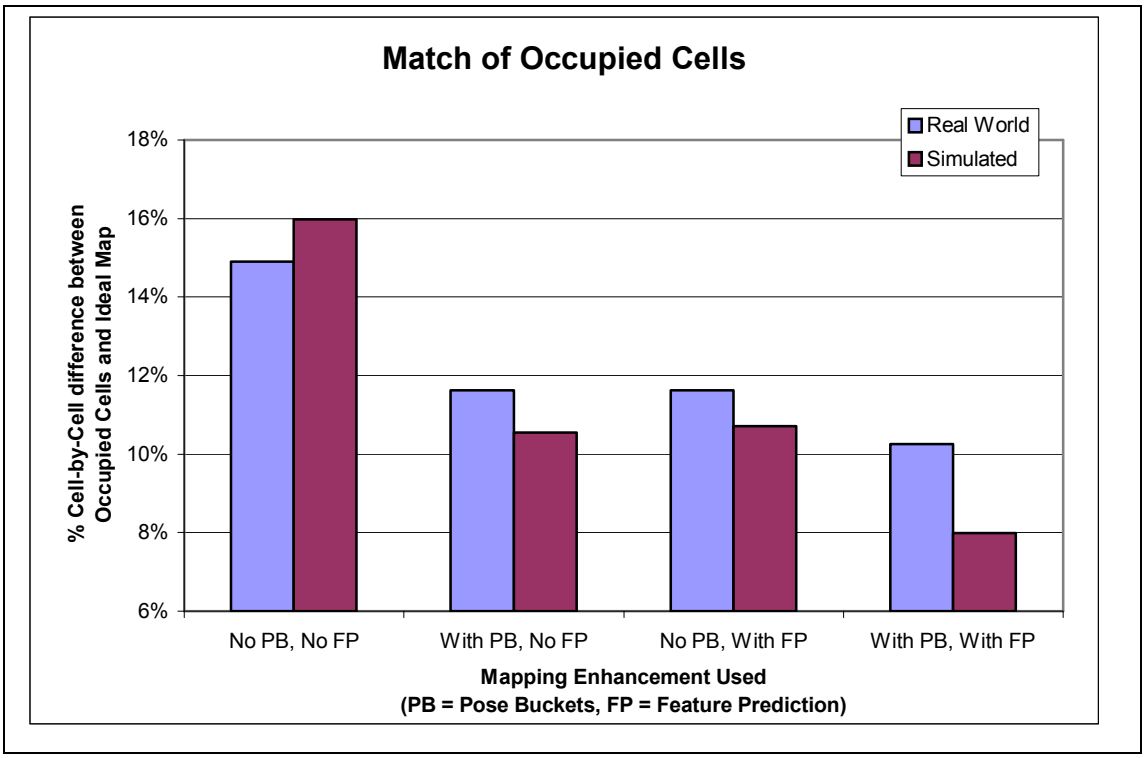


Fig 6.25 Comparison of simulated and real-world results of the percentage cell-by-cell difference between the occupied cells in the generated *Star* map, and the ideal map of the *Star* environment.

The simulated and real-world results mimic each other in Fig 6.25, with both of them showing that both Pose Buckets and Feature prediction reduce the number of mis-marked occupied cells, and that both used together give the best map of all.

6.10.4 Percentage of False Positive Paths in Generated Maps

Simulated Results

Mapping Enhancements	% False Positive Paths
No PB, No FP	66.57%
With PB, No FP	36.61%
With FP, No PB	40.63%
With FP,With PB	14.86%

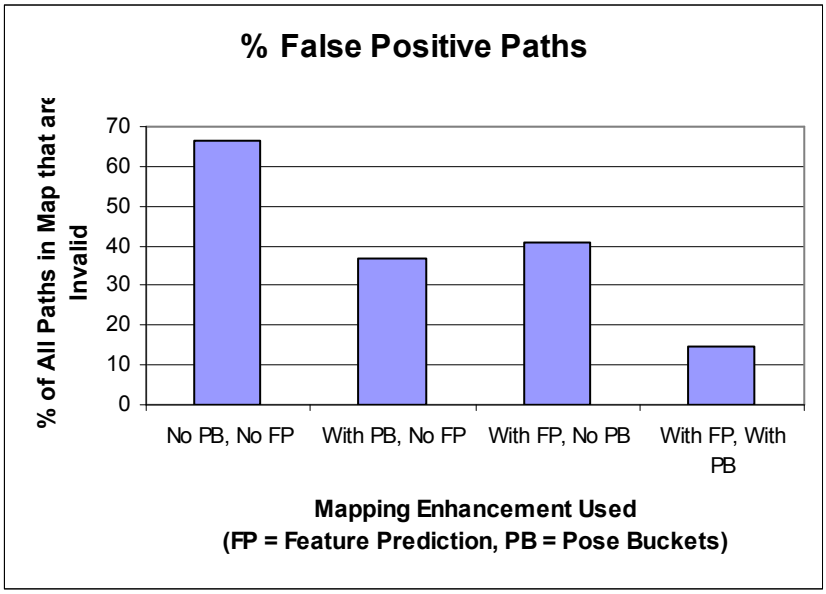


Fig 6.26 Percentage of false positive paths in the generated map. These are paths can be generated in the generated map, but which in the ideal map would cause the robot to collide with an obstacle.

This test clearly shows the weakness of a mapping algorithm – when neither pose buckets nor feature prediction are used to generate the map to the map, 66% of all paths generated on the map will cause the robot to crash. While low-level behaviours can be used to avoid an actual collision with an object, the robot will be required to compute an alternative path. All of this will cause a significant performance hit, with the robot taking far longer to complete the task at hand.

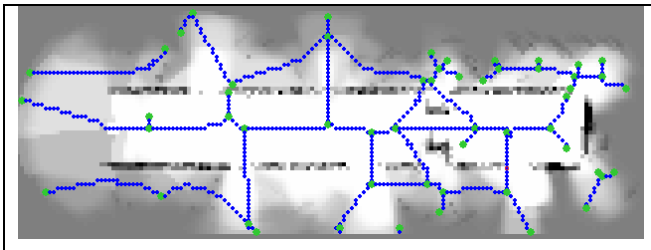


Fig 6.27 (a) Voronoi graph in map of *eCORRsim* generated by *ME85mod* without feature prediction or pose buckets. Note the many paths are plotted through either unknown areas or obstacles.

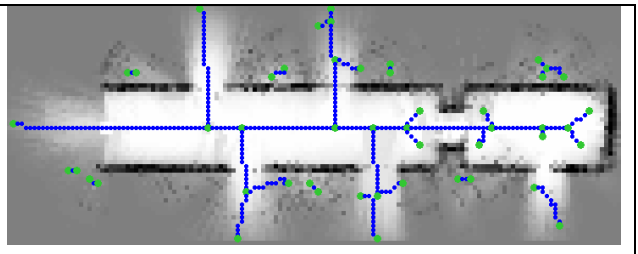


Fig 6.27 (b) Voronoi graph in map of *eCORRsim* generated by *ME85mod* using both feature prediction and pose buckets. The majority of paths are confined to actual freespace areas.

Either pose buckets or feature prediction, used in isolation, reduce the number of invalid paths in the map by almost 50%. However the best results are once again

achieved by using the two filtering method together, which results in almost five times fewer false positive paths in the generated map. This can be seen in Fig 6.27, where the introduction of feature prediction and pose buckets results in far fewer paths being plotted through unknown or obstacles in the ideal map.

Real-World Results

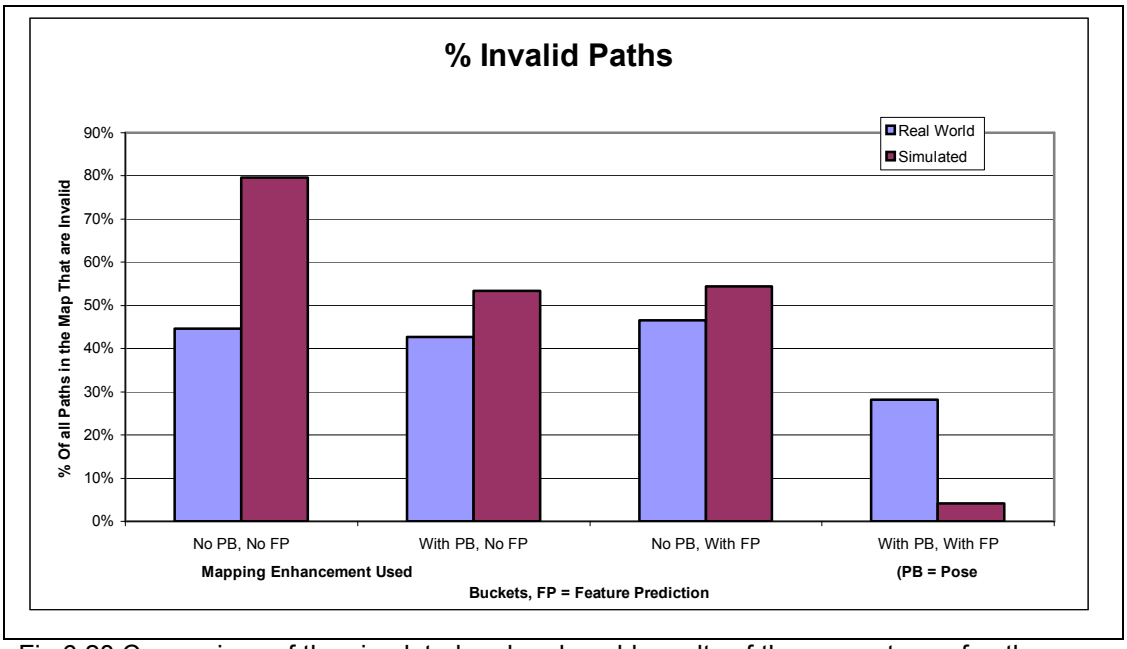


Fig 6.28 Comparison of the simulated and real-world results of the percentage of paths created in the generated map that would cause a collision in the real-world.

The real world results resemble the simulated results very closely in Fig 6.28, with the exception of the map generated using neither Pose Buckets nor Feature Prediction, which performed much better than in the simulated run. This can be explained by the fact that, in the *Star* environment, there are fewer specular reflections that in the simulated environment because of the roughness of the materials used in its construction. Both Pose Buckets and Feature Prediction, on their own, seem to perform more or less the same as each other based on these results. However, visual inspection of the maps created show that the maps generated just using Pose Buckets slightly overestimate the number of occupied cells in the map, and the maps generated with Feature Prediction slightly overestimate the number of freespace cells. When the two are used in tandem they tend to cancel out the weaknesses of the other to some degree, giving the best overall map.

6.10.5 Percentage of Paths from Ideal Maps that Could Not be Completed In Generated Maps

Simulated Results

Mapping Enhancements	%False Negative Paths
No PB, No FP	0.925%
With PB, No FP	1.819%
With FP, No PB	1.029%
With FP, With PB	1.235%

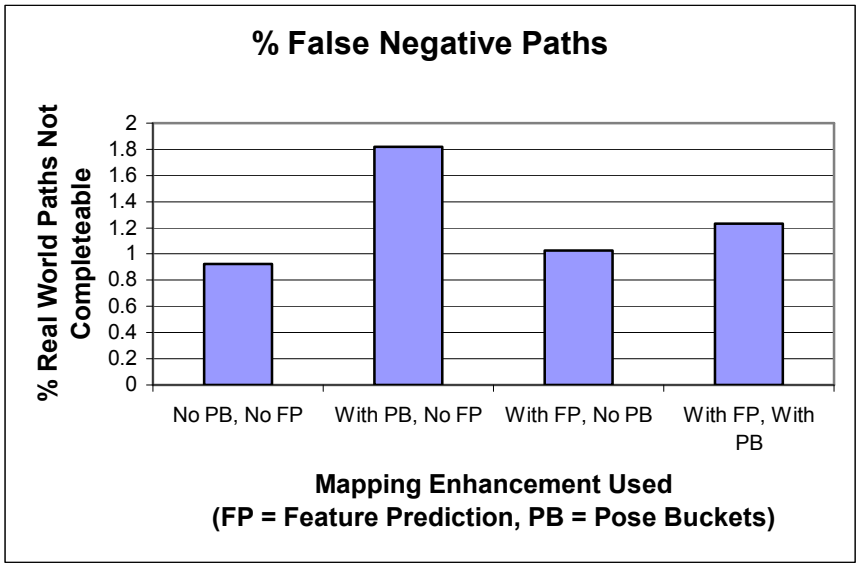


Fig 6.29 Percentage of false negative paths in all maps generated by *ME85mod, ME88mod* or *K97mod*, grouped by their use of feature prediction and pose buckets. These are paths that can be plotted in the ideal map, but cannot be completed in the generated map.

This test is designed to test the degree to which a mapping system marks a freespace cell as being occupied. It does this by measuring the number of paths that can be completed in the real world, but could not be completed in the generated map. The reason percentages are so low, all less than 2%, is because all paths in a Voronoi graph maximise the distance from all obstacles, so the only time a path will come close to an obstacle is in a very confined space such as a narrow corridor.

This is the only test that the mapping systems which use neither pose buckets or feature prediction perform best. Because these systems treat all readings equally and there are generally many more freespace readings than obstacle readings, they tend to overestimate the freespace areas rather than place obstacles in the freespace areas.

Pose buckets, used on their own, perform worst of all the methods with respect to false negative paths. This is because, while they believe both freespace and surface readings equally, they ensure that there is at most a difference of 1 reading between the number of freespace and surface readings from any given position. This prevents

the freespace readings from ‘drowning out’ the incorrect surface readings, unlike the systems that don’t use pose buckets.

Feature prediction, used on its own, performed similarly to the mapping methods that used neither it nor pose buckets. This is because feature prediction only filters out readings that appear to pass through walls, and has little or no effect on readings that are too short, or that appear to be in freespace areas without passing through an obstacle. For these reasons, the results in this test are more or less the same as if feature prediction were not used at all.

When pose buckets are combined with feature prediction, it improves the performance in comparison with the pose buckets in isolation, but is worse than the performance of the feature prediction algorithm on its own. The reason for the improvement in performance over PB is that some of the erroneous readings accepted by PB, which then cause it to discard later correct readings, have been filtered out by FP. The degradation of the maps in comparison with FP is the flip side of that coin – FP doesn’t identify all of the incorrect readings that are too short, as explained above, so PB still accepts some incorrect readings and discards later valuable readings, whereas feature prediction allows them all to be added to the map, enabling the good readings to somewhat counterbalance the bad readings.

Real-World Results

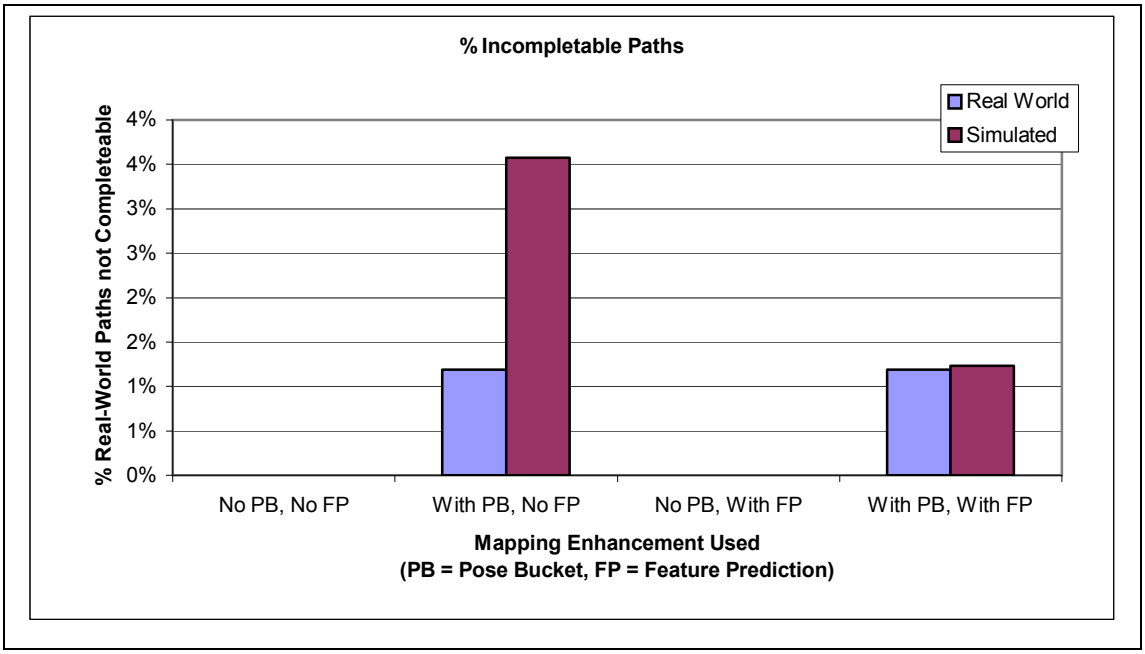


Fig 6.30 Comparison of the simulated and real-world results of the percentage of paths in the real world that could not be completed in the generated map.

Fig 6.30 shows that the real-world data follows the same pattern as the simulated data, albeit with Pose Buckets on their own performing significantly better.

6.10.6 Evaluation of Results 2

The tests performed in this section have shown conclusively that both pose buckets and feature prediction significantly add to the accuracy and usefulness of a map. When used individually both caused a large improvement in the generated map's fidelity to the ideal map. When used together, they yielded even better results, proving that they are complimentary methods of filtering noisy sonar readings, and that while they overlap somewhat in the readings that they discard, where one fails to identify an erroneous reading, the other will often succeed.

The real-world results mirror the simulated results very often, and the few times there are differences, this can be attributed to the differences in the surface texture of the materials in the environment. While this validates the larger body of simulated data, it also shows that the Pioneer simulator used is a very well build simulator

Chapter 7: Conclusions

7.1 Theoretical Evaluation of Mapping with Mobile Robots using Sonar Sensors

There are a number of areas of interest in this thesis. Firstly, theoretical evaluations of the following six bodies of work are presented in Chapter 2.

- Probabilistic Occupancy Grid theory put forward by Moravec and Elfes [46].
- Bayesian based Occupancy Grid methods by Matthies and Elfes [42].
- Konolige's MURIEL [33] method for eliminating incorrect sonar readings.
- Crowley's hybrid method [15] of extracting features from a local grid map.
- Elfes' Inference Grids [20].
- Thrun's automatic learning of sensor models [57, 58] and confidence values using neural nets.

The above theoretical discussions paint an in-depth picture of the current state of map building with mobile robots, and a multitude of paradigms put forward to solve the many problems associated with it such as sensor noise, positional uncertainty and mathematical models.

While the system designed by Moravec and Elfes [46] was prone to gross errors, with no compensation for noise and a simple mathematical model, it laid the foundations for others to improve upon. At the same time as Moravec and Elfes published their seminal paper, Crowley [15] published an alternative method of mapping that used a hybrid of grid-based maps and maps built of line segments. This method, while integrating nicely with localisation routines because the orientation of obstacles is known, suffered from the fact that it did not deal effectively with noisy sonars. Neither did it represent the world probabilistically, which led to problems when conflicting sensory input was received.

Matthies and Elfes soon improved upon the mathematical model of [46], making it more mathematically robust using probability theory. While this was a welcome addition, and led to more intricate mathematical theories being applied to map

building, the Bayesian update procedure in itself did not improve the quality of the maps produced using it.
Konolige developed an improved sensor model as well as developing a more intricate mathematical probability update procedure. He also made an attempt at dealing with the large amount of noise in sonar readings. This attempt was only partly successful, as will be explained along with all the other experimentation results.

Thrun took an alternate approach to dealing with erroneous sonar readings by training a *confidence* network to identify readings that seem to be incorrect. He also trained a neural network to automatically learn a sonar model, rather than developing his own mathematical formula, á la Konolige. The quality of the maps creating using Thrun's approach can not be directly determined as he did not present qualitative analysis of his results. Therefore it is difficult to know whether the following two potential pitfalls have been accounted for:

1. Using neural nets for map building can be inaccurate if the training data is not well chosen, as with many neural net applications.
2. When the robot is introduced to a new environment it must be retrained, which means that it does not generalise well to all environments.

7.2 Feature Prediction – An Algorithm For Detecting Specular Sonar Readings

A method for filtering noisy sonar readings called Feature Prediction was presented in Chapter 3. The results presented in Chapter 6 show that Feature Prediction, when combined with any of the tested mapping systems, improved the quality of the maps by a large degree. For example, in section 6.10.2, the percentage difference between the generated map and the ideal map without feature prediction is 12.73%, whereas with feature prediction it is 6.68%. Therefore the difference between the two maps has been halved using Feature Prediction. While it is not perfect, it is successful in discarding the vast majority of erroneous sonar readings and only very rarely discards correct readings. Feature Prediction therefore goes a long way towards solving one of the major difficulties of mapping with sonars, that of filtering out the noisy readings.

Feature Prediction makes some assumptions about the environment in order to make predictions about it – namely it assumes that the world is made up of straight line segments. It will therefore work best in environments that come close to matching those assumptions. Feature Prediction performs less well in highly cluttered environments with many small objects, causing more erroneous readings to be believed than in simpler worlds. Even in cluttered environments however, using the Feature Prediction algorithm in conjunction with a mapping system such as *ME88* or *K97* leads to a much better map being produced than when it is not used.

7.3 Empirical Evaluation of Mapping Algorithms

The experiments performed aimed to answer a number of questions.

1. Is there any advantage to using a Bayesian mathematical formula to integrate new readings into the map over using the simple update formula presented in [46]?
2. Do the changes to the update procedures and biases introduced in *ME85mod* and *ME88mod* offer any improvements over the systems upon which they are based, *ME85* and *ME88* respectively.
3. Does Konolige's sonar model [33] create better maps than the more simplistic two dimensional gaussian sonar model from [46]?
4. Does the Feature Prediction noise filtering method significantly improve the quality of the map when used in conjunction with a map building algorithm?
5. Does using Pose Buckets to discard duplicate readings improve the quality of a map?

7.3.1 Question 1 – Do Bayesian Update Formulas Improve A Map?

To answer question one, experiments have shown that, when an identical sonar model is used (mapping systems *ME85* and *ME88*), mapping systems that use a Bayesian probability update procedure perform either the same or worse than mapping systems that use the simple mathematical update proposed in [46]. This is largely due to the fact that the Bayesian update can cause a single reading to change the map to a large degree. For example, a single specular reading that reports an area to be free of obstacles when it is not can cause the probability of some cells to go from an initial value of 0.5 (unknown) to less than 0.1 (almost definitely unoccupied). A more

incremental update procedure that changes the map's cell values in smaller increments is required.

However, the strength of the increment is determined by the sonar model. For example, the sonar model decides that a particular cell should be updated with the number 0.2. This figure is then integrated with the current value of the cell either with the Bayesian formula or the update procedure from [46]. As Moravec and Elfes' update procedure [46] updates the cell probability in smaller increments than the Bayesian formula, it usually results in a better map being generated. This result does therefore not prove that one mathematical update method is *better* than the other. Rather, given that the two dimensional gaussian sonar model used in *ME85* and *ME88* can cause large alterations to occur with just a single reading, it is better to use a mathematical update procedure that integrates that value more weakly, i.e. the formulae from [46]. When a variant of the Bayesian update procedure is used with a different sonar model [33] based on Konolige's work, it performs much better because the sonar model changes the map in much smaller increments.

7.3.2 Question 2 – Have Modifications On Original Theories Improves Performance?

The second question posed was whether or not the alterations made to *ME85* and *ME88,* which resulted in *ME85mod* and *ME88mod* respectively, lead to better maps being generated. These alterations including adding an enhanced version of Konolige's Dynamic Mixture Model to both *ME85mod* and *ME88mod*. In the case of *ME85mod* it also involved removing the bias towards believing freespace readings over surface readings which Moravec and Elfes had created as part of their mathematical update procedure and instead having no bias whatsoever, neither towards freespace nor surface readings.

The results in Chapter 6 prove that *ME85mod* consistently performs better than *ME85* and that *ME88mod* performs better than *ME88*. This shows that the when the enhanced Dynamic Mixture Model is used the map generated is much more accurate than without it.

7.3.3 Question 3 – Is Konelige's Sonar Model More Effective Than A Simple 2D Sonar Model

The third question answered by the series of experiments is whether or not using the sonar model proposed by Konolige in [33] resulted in a more accurate map being generated than the more simplistic 2D gaussian sonar model developed by Moravec and Elfes in [46]. In almost all the benchmarking tests performed on the maps generate by the various mapping algorithms, *K97* and *K97mod*, the two mapping systems that used Konolige's sonar model, performed better than the other four mapping systems that used the 2D gaussian model. The only exception was with *K97*, which has a tendency to overestimate the occupancy values of cells, so in environments very prone to specular readings, it can perform badly when evaluated with the fifth benchmark, which tests the robot's ability to plan paths to real-world positions using the generated map.

Part of the reason Konolige's sonar model performs better than the 2D gaussian model is that it updates the map in very small increments, which means that a single incorrect reading does not affect the map to any considerable degree. Konolige's sonar model is therefore much more tolerant of errors and is thus better able to recover from them than systems that use a 2D gaussian sonar model. Perhaps the main reason Konolige's sonar model performs better than the 2D gaussian however, is that is simply a better approximation of the actual operational characteristics of the sonar beam.

7.3.4 Question 4 – Does Feature Prediction Improve A Map's Quality?

The fourth question answered by the experiments was whether or not using the Feature Prediction algorithm to filter out noisy sonar readings results in a better map being generated. As discussed in the previous section, whenever Feature Prediction was integrated with a mapping system, the quality of the maps produced drastically improved.

7.3.5 Question 5 – Does Using Pose Buckets Improve A Map's Quality?

The fifth question answered by the experiments related to whether or not using Pose Buckets to discard duplicate readings increased the quality of the map. Experiments show that the use of Pose Buckets does indeed result in more accurate maps being generated. This can be attributed to the fact that there are usually a greater number of

noisy readings than useful readings, and by ensuring that only one reading from a given position can affect any particular cell, multiple noisy readings are unable to reinforce each other. In this way, the effect of noisy readings is reduced.

In addition to proving that Pose Buckets increase the generated map's fidelity to the real-world environment, it was also discovered that both Pose Buckets and Feature Prediction are highly complimentary filtering methods. In almost every test, the result when both methods were combined to filter sonar readings before they are integrated with the map was better than using either of the methods on its own. While neither method is perfect and some incorrect readings do slip through (more so with Pose Buckets than with Feature Prediction), where Pose Buckets accept a noisy reading as being correct, Feature Prediction will discard it, and vice versa.

Another welcome discovery is that, while Pose Buckets can use a significant amount of memory, a problem mostly alleviated by the use of quad trees to store the Pose Bucket's data, neither Pose Buckets nor Feature Prediction require a great deal of processing power, and were not found to affect the speed of operation of the mapping systems using them.

7.4 Future Research

There are a number of interesting research areas that require further work in mobile robot navigation. Firstly there is the issue of localisation. The combination of localisation and map building through simultaneous localisation and map building (*SLAM*) techniques is a vital component of successful mobile robot navigation architectures. A large amount of work is currently under way in a variety of localisation methodologies, using map-matching techniques, Markov based sensor matching, and visual methods. One of the most promising fields is that of localising a robot within a three-dimensional map using stereovision to detect the position and colour of objects [47].

Three dimensional map building is a field that has only recently become a feasible research area, due to the computational resources it requires. However computers are now powerful enough to both store and process the huge volumes of data in three-

dimensional maps. Building such maps using sonars, lasers and stereovision is an open research area that shows much promise.

The automatic learning of sensor models and confidence measures is another open research area, with numerous possible approaches that could be taken. Whereas Thrun used neural nets to learn sonar models and confidence networks, it is also possible any number of genetic algorithm (GA) approaches could be successful, as well as different types of neural nets.

A research area often ignored by map building researchers is that of map building in dynamic environments. Simple occurrences such as an open door being closed, or a chair being moved can be very difficult to identify and model using a map. This is very much an open issue, and one that must be solved before robots can be said to have a truly accurate map of a dynamic environment.

7.5 Completion of Research Objectives

The contribution of this thesis, as described in chapter one was as follows.

7.5.1 Provides an empirical evaluation of a number of map building methods

The experimentation results in chapter six present a detailed examination of a variety of sonar models and mathematical update methods, as well a number of methods for identifying specular readings. Empirical evaluations are performed for each possible configuration of these various mapping methodologies to form a comprehensive and meticulous representation of the merits of each mapping method.

7.5.2 The Feature Prediction algorithm has been developed to enhance map building algorithms, specifically to identify noisy specular readings

The Feature Prediction algorithm was presented in detail in chapter three as a method for identifying specular sonar readings and assigning a confidence value to each sonar reading. The algorithm was implemented in the *SpecularEstimator* object as detailed in chapter four, which is portable to any architecture and to any robot configuration. The results presented in chapter six show clearly that Feature Prediction improves the quality of the maps generated to a considerable degree.

7.5.3 Development of a suite of benchmarking techniques for map building algorithms

A comprehensive suite of five benchmarking techniques for evaluating the fitness of a map was presented in chapter five. These benchmarks used techniques from the field of image recognition, as well as using methods specific to metric grid maps. Benchmarks were also developed that test a map in the manner in which a robot would use it, i.e. by generating paths in it using a path planning algorithm, and then calculating the value of those paths.

7.5.4 Design and implementation of platform-independent robot control architecture, with support for threading, multiple clients and callbacks, and can be run on a single machine or distributed over a network.

A framework for robotic control was presented in chapter four. A standard interface for robotic control services is presented to which clients can register, send information, and be notified by the service of any actions that must be taken. The architecture contains no dependencies on any robot or protocol, and is therefore portable to any robot or simulator. It is also full distributable, and has been successfully distributed over multiple machines using CORBA to execute multiple services simultaneously over a network, each receiving data from multiple robots operating in the same environment.

And at this point, there is only one thing left to be said:

```
Tá sé criochnadh, buoichas le dia!
```

Bibliography

1. Baron, R. J. 1981. "Mechanisms of human facial recognition." International Journal of Man Machine Studies 15: 137-178.
2. Boden, M. A. 1988. Computer Models of the Mind: Computational Approaches in Theoretical Psychology. Cambridge, Cambridge University Press.
3. Boden, M. A. 1990. The Philosophy of Artificial Intelligence, Oxford University Press.
4. Borenstein, J., Everett, B., Feng, L. 1996. “Navigating Mobile Robots: Systems and Techniques.” A.K. Peters, Ltd, Wellesley, MA.
5. Brooks, R. A. 1986. “A Robust Layered Control System For A Mobile Robot.” IEEE Journal of Robotics and Automation 2(1): 14-23.
6. Brooks, R. A. 1986. “Achieving Artificial intelligence through building robots”, AI-Laboratory, Massachusetts Institute of Technology, Cambridge, MA, AI-Memo 899, 1986. Cambridge, MA, AI Laboratory, MIT.
7. Brooks, R. A., Connell, J.H., Ning, P. 1988. “Herbert: A Second Generation Mobile Robot.” Memo 1016. MA, USA, MIT AI Lab.
8. Brooks, R. A. 1989. “A Robot That Walks: Emergent Behaviors from a Carefully Evolved Network.” MA, USA, MIT AI Laboratory.
9. Brooks, R. A. 1990. “Elephants Don't Play Chess.” MA, USA, MIT AI Laboratory.
10. Brooks, R. A. 1991. “Intelligence Without Reason.” MA, USA, MIT AI laboratory.
11. Brooks, R. A. 1991. "Intelligence Without Representation." Artificial Intelligence Journal(47): 139-159.
12. Brunelli, R., Poggio, T. 1993. "Face Recognition: Features versus Templates." Pattern Analysis and Machine Intelligence, IEEE Transactions on **15**(10): 1042-1052.
13. Burgard, W., Fox, D., Hennig, D., Schmidt, T 1996. “Estimating the Absolute Position of a Mobile Robot Using Position Probability Grids.” Proceedings of the Thirteenth National Conference on Artificial Intelligence, 1996.
14. Collins, J. J., O’Sullivan, S., Mansfield, M., Eaton, M., Haskett, D. 2004. “Developing an extensible benchmarking framework for map building

paradigms.”. In Proceedings Ninth International Symposium on Artificial Life and Robots, Oita, Japan, January 2004.

15. Connell, J. 1989. “A Colony Architecture for an Artificial Creature.” MA, USA, MIT AI Lab.
16. Crowley, J. L. 1985. "Navigation for an Intelligent Mobile Robot." IEEE Journal of Robotics and Automation RA-1(1): 31-41.
17. Crowley, J. L. 1989. “World Modelling and Position Estimation for a Mobile Robot Using Ultrasonic Ranging.” Proceedings 1989 IEEE International Conference on Robotics and Automation, Scottsdale, Arizona, USA.
18. Devore, J. L. 1991. Probability and Statistics for Engineering and the Sciences, 3rd Ed. Belmont, CA, USA, Wadsworth Inc.
19. Drumheller, M. 1987. "Mobile Robot Localization Using Sonar." IEEE Transactions on Pattern Analysis and Machine Intelligence PAMI-9: 325-332.
20. Elfes, A. 1992. Dynamic Control of Robot Perception Using Multi-Property Inference Grids. Proceedings, 1992 IEEE International Conference on Robotics and Automation, 1992..
21. Evans, A. 2001. “Four Tricks for Fast Blurring in Software and Hardware” [online]. http://www.gamasutra.com/features/20010209/evans_01.htm
22. Fodor, J. A. 1987. “Modules, frames, fridgeons, sleeping dogs and the music of the spheres. The robot's dilemma: The frame problem in artificial intelligence”. Z. Pylyshyn. Norwood, NJ.
23. Fox, D., Burgard, W., Thrun, S. 1999. "Markov Localization for Mobile Robots in Dynamic Environments." Journal of Artificial Intelligence Research 11: 391-427.
24. Gat, E. 1997. “On Three-Layer Architectures.” in Artificial Intelligence and Mobile Robots. D. Kortenkamp, Bonnasso, R.P., Murphy, R., AAAI Press.
25. Ghosh, J. 1992. “Evidence Combination techniques for robust Classification of Short Duration Oceanic Signals.” SPIE Conference on Adaptive and Learning Systems, Orlando, FL.
26. Griffith, K. A. 1974. "A Comparison and Evaluation of Three Machine Learning Procedures as Applied to the Game of Checkers." Artificial Intelligence Journal 5: 137-148.
27. Group, P. U. 1992. “Ultrasonic ranging system manual.” Technical report, Polaroid Corporation, Atlanta, GA, USA,
28. Harnad, S. 1993. “Problems, Problems: the Frame Problem as a Symptom of the

Symbol Grounding Problem." Psycoloquy: 4,#34 Frame Problem (11). 2003.
29. Heckerman, D. 1986. "Probabilistic interpretation for MYCIN's uncertainty factors." Uncertainty in Artificial Intelligence. L. N. L. Kanal, J.F. North-Holland**:** 197-196.
30. Hwang, Y. K., Ahuja, N. 1992. "A Potential Field Approach to Path Planning." IEEE Transactions on Robotics and Automation 8(1): 23-32.
31. Kaelbling, L. P. 1996. "Reinforcement Learning: A Survey." Brown University, RI, USA,
32. Konolige, K., Myers, K. 1996. "The Saphira Architecture for Autonomous Mobile Robots." in AI-based Mobile Robots: Case studies of successful robot systems. D. a. B. Kortenkamp, R. Peter and Murphy, Robin, MIT Press.
33. Konolige, K. 1997. "Improved Occupancy Grids for Map Building." Autonomous Robots 4(4): 351-367.
34. Kortenkamp, D., Bonasso, R.P., Murphy, R. 1998. Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems. Cambridge, MA, USA, MIT/AAAI Press.
35. Kose, H., Akyn, H.L. 2000. "Towards a Robust Cognitive Architecture for Small Autonomous Mobile Robots." ISCIS XV, The Fifteenth International Symposium on Computer and Information Sciences: 447-455.
36. Lee, D. 1996. "The Map-Building and Exploration Strategies of a Simple Sonar-Equipped Mobile Robot; an Experimental Quantitative Evaluation." University College London, London, UK,
37. Lee, D. 1997. "Quantitative Evaluation of the Exploration Strategies of a Mobile Robot." International Journal of Robotics Research 16(4): 413-447.
38. MacDorman, K. F. 1999. "Grounding symbols through sensorimotor integration." Journal of the Robotics Society of Japan(17): 20-24.
39. Mansfield, M., Collins, J.J, Eaton, M., O'Sullivan, S., Haskett, D. "Developing a statistical baseline for robot pursuit and evasion using a real world control architecture." In Proceedings Ninth International Symposium on Artificial Life and Robots, Oita, Japan, January 2004.
40. Marr, D. 1971. "Simple memory: a theory for archicortex." Phil. Trans. Royal Soc. London 262: 23-81.
41. Martin, M. C., Moravec, H.P. 1996. "Robot Evidence Grids." Robotics Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania

42. Matthies, L., Elfes, A. 1988. “Integration of Sonar and Stereo Range Data Using a Grid-Based Representation.” Proceedings of the 1988 IEEE International Conference on Robotics and Automation, Philadelphia, Pennsylvania, USA.
43. McCulloch, W. C., Pitts, W 1943. A Logical Calculus of the Ideas of Immanent Nervous Activity, MIT Press.
44. Moore, A. W., Atkeson, C.G. 1993. "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Real Time." Machine Learning 13: 103-130.
45. Moravec, H. P. 1983. "The Stanford Cart and The CMU Rover." Proceedings of the IEEE 71(7): 872-884.
46. Moravec, H. P., Elfes, A. 1985. “High Resolution Maps from Wide Angle Sonar.” Proceedings of the 1985 IEEE International Conference on Robotics and Automation.
47. Moravec, H. P. 1999. “Robust Navigation by Probabilistic Volumetric Sensing.” DARPA ITO solicitation. PA, USA, Carnegie Mellon University.
48. Newell, A., Shaw, J.C., Simon, H. A. 1958. “Chess-Playing Programs and the Problem of Complexity.” in Computers and Thought. E. Feigenbaum, Feldman, J. New York, McGraw Hill.
49. Nilsson, N. J. 1984. “Shakey the Robot,” Technical Report 223, AI Centre, SRI International.
50. O'Keefe, J., Nadel, L. 1978. The Hippocampus as a Cognitive Map, Oxford University Press.
51. O’Sullivan, S., Collins, J. J., Mansfield, M., Eaton, M., Haskett, D. 2004. “A Quantitive evaluation of sonar models and mathematical update methods for Map Building with mobile robots.” In Proceedings Ninth International Symposium on Artificial Life and Robots, Oita, Japan, January 2004.
52. O’Sullivan, S., Collins, J. J., Mansfield, M., Eaton, M., Haskett, D. 2004. “Linear Feature Prediction for Confidence Estimation of Sonar Readings in Map Building”. In Proceedings Ninth International Symposium on Artificial Life and Robots, Oita, Japan, January 2004.
53. Prestes, E., Idiart, M.A.P., Engel, P.M., Trevisan, M. 2001. “Exploration technique using potential fields calculated from relaxation methods.” Proceedings. IEEE/RSJ International Conference on.Intelligent Robots and Systems, 2001.
54. Quirk, G., Muller, R., and Kubie, J. 1990. "The firing of hippocampal place cells in the dark depends on the rat's recent experience." Journal of Neuroscience **10**(6):

2008-2017.

55. Samuel, A. L. 1959. “Some Studies in Machine Learning Using the Game of Checkers.” Computation and Intelligence: Collected Readings. G. F. Luger, Menlo Park. CA/Cambridge, MA/London, AAAI Press/The MIT Press.
56. Steels, L. 1994. “Emergent Functionality in Robotic Agents through on-line evolution.” Artificial Life IV. Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, Cambridge, MA, MIT Press.
57. Thrun, S., A. Bücken. 1993. “Exploration and model building in mobile robot domains.” IEEE International Conference on Neural Networks, 1993., San Francisco, CA,USA.
58. Thrun, S., A. Bücken. 1997. “Learning Maps for Indoor Mobile Robot Navigation.” AI Magazine. 99**:** 21-71.
59. Thrun, S., Bucken, A., Burgard, W., Fox, D., Fröhlinghaus, T., Henning, D., Hofmann, T., Krell, M., Schmidt, T. 1998. “Map learning and high-speed navigation in RHINO.” in AI-based Mobile Robots: Case Studies of Successful Robot Systems. D. Kortenkamp, Bonasso, R.P., Murphy, R., MIT Press.
60. Turing, A. M. 1950. "Computing Machinery and Intelligence." Mind 49: 433-460.
61. Weiss, M. A. 1999. Data Structures and Algorithm Analysis in C++ (Second Edition), Addison Wesley Longman, Inc.
62. Xiao, J., Michaelewicz, Z., Zhang, L., 1997. “Evolutionary Planner/Navigator: operator performance and self-tuning.” Proceedings of IEEE Transactions on Evolutionary Computation 1997.

Appendices

A1 Design Diagrams of Robot Control Architecture

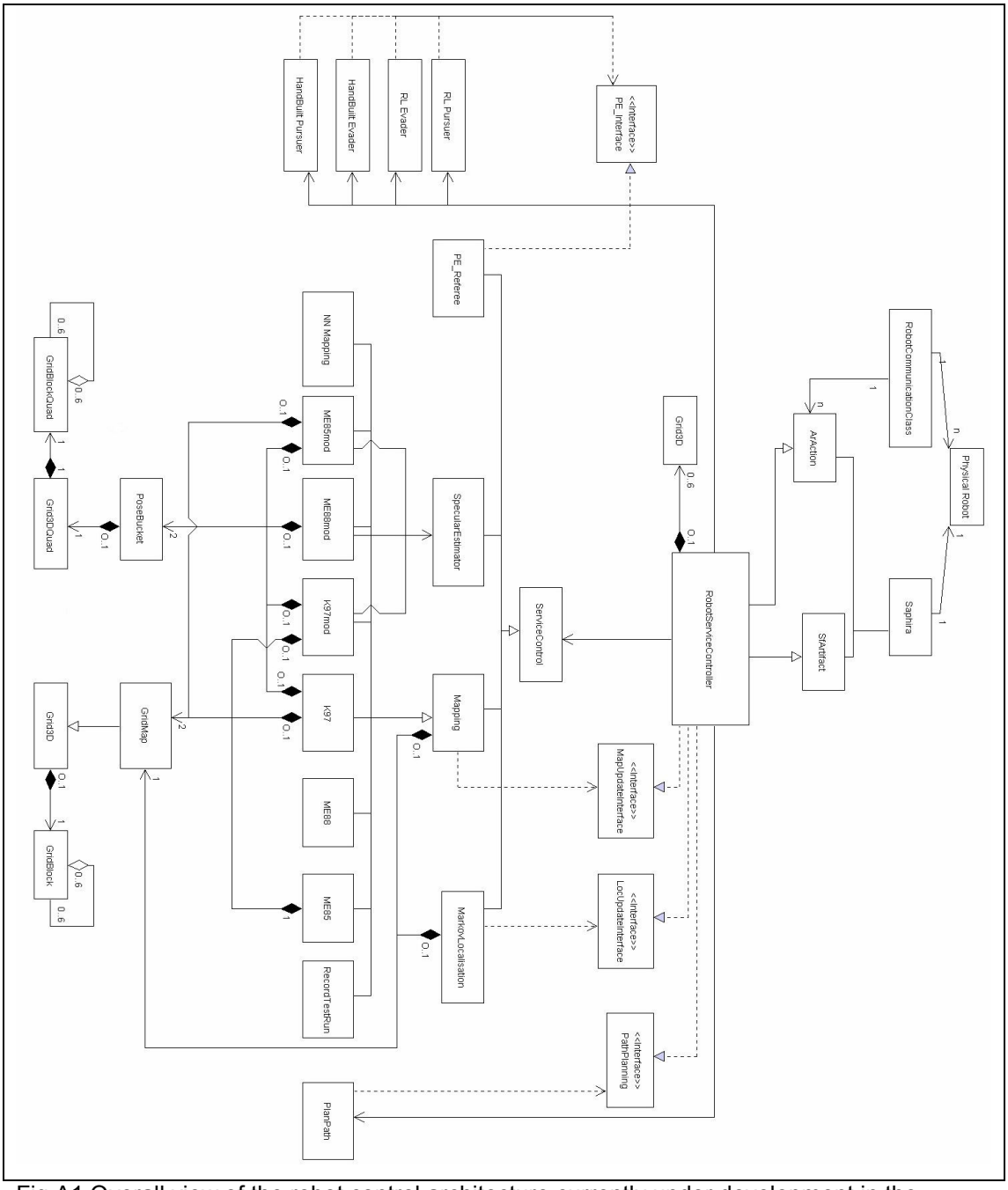


Fig A1 Overall view of the robot control architecture currently under development in the University of Limerick robotics group. A subsection of these modules were used in experimentation. The Pursuit and Evasion (*PE*), Localisation and Neural Net mapping (*NN Mapping)* modules are being developed by other members of the robotics group. All other modules were created for this thesis.

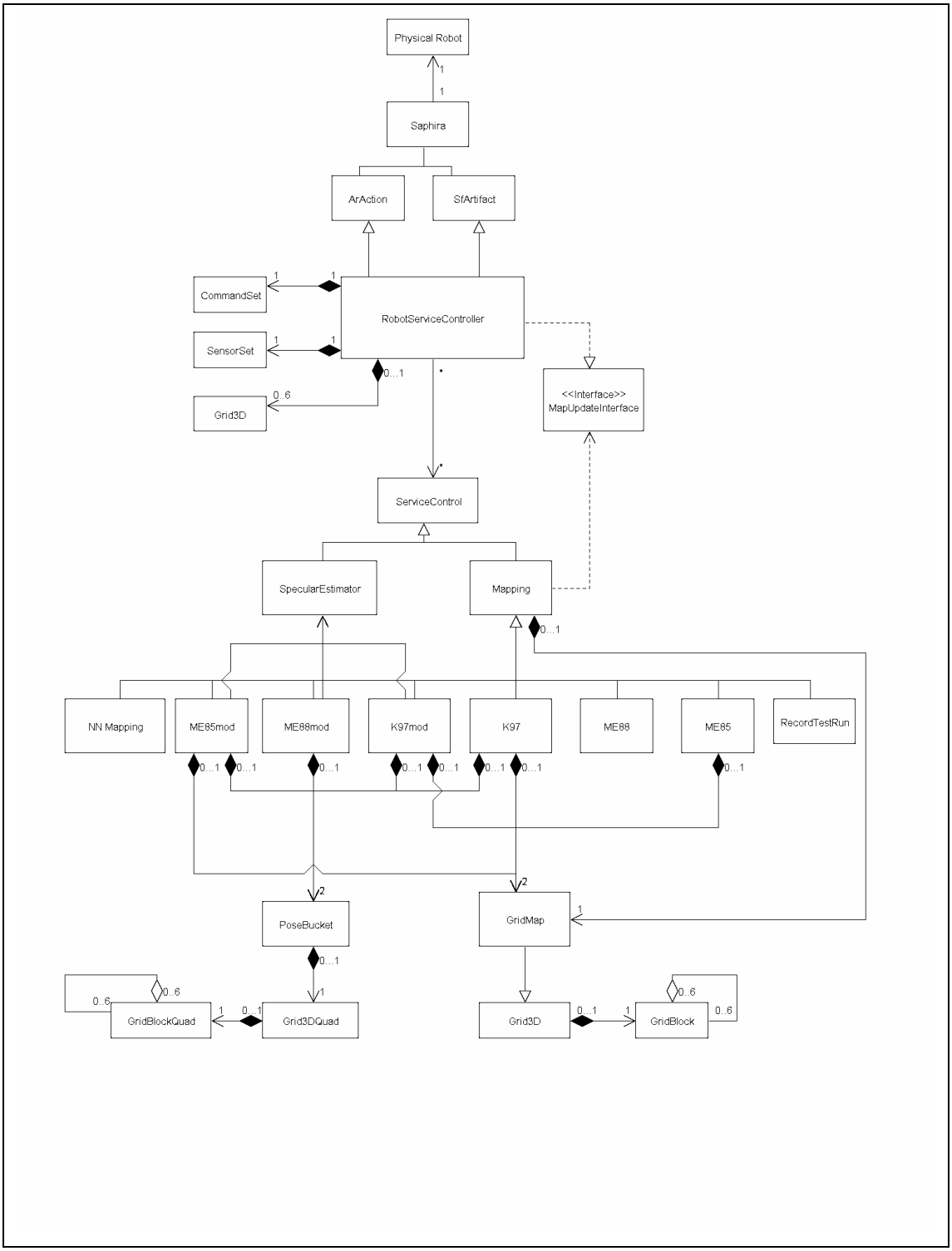


Fig A2 Architecture of modules used in experimentation. This is a subsection of the complete robot control architecture under development in the University of Limerick robotics group.

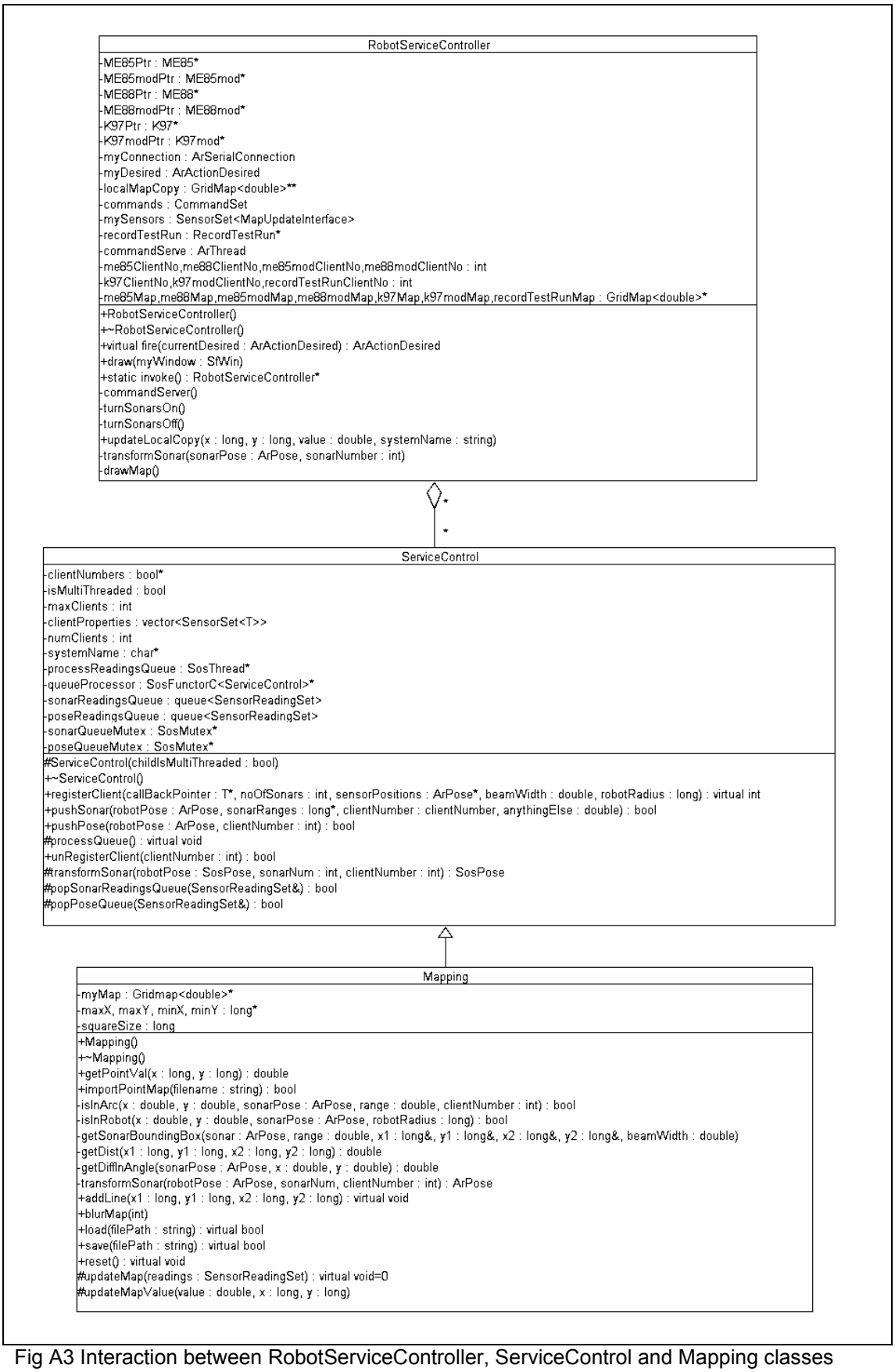


Fig A3 Interaction between RobotServiceController, ServiceControl and Mapping classes

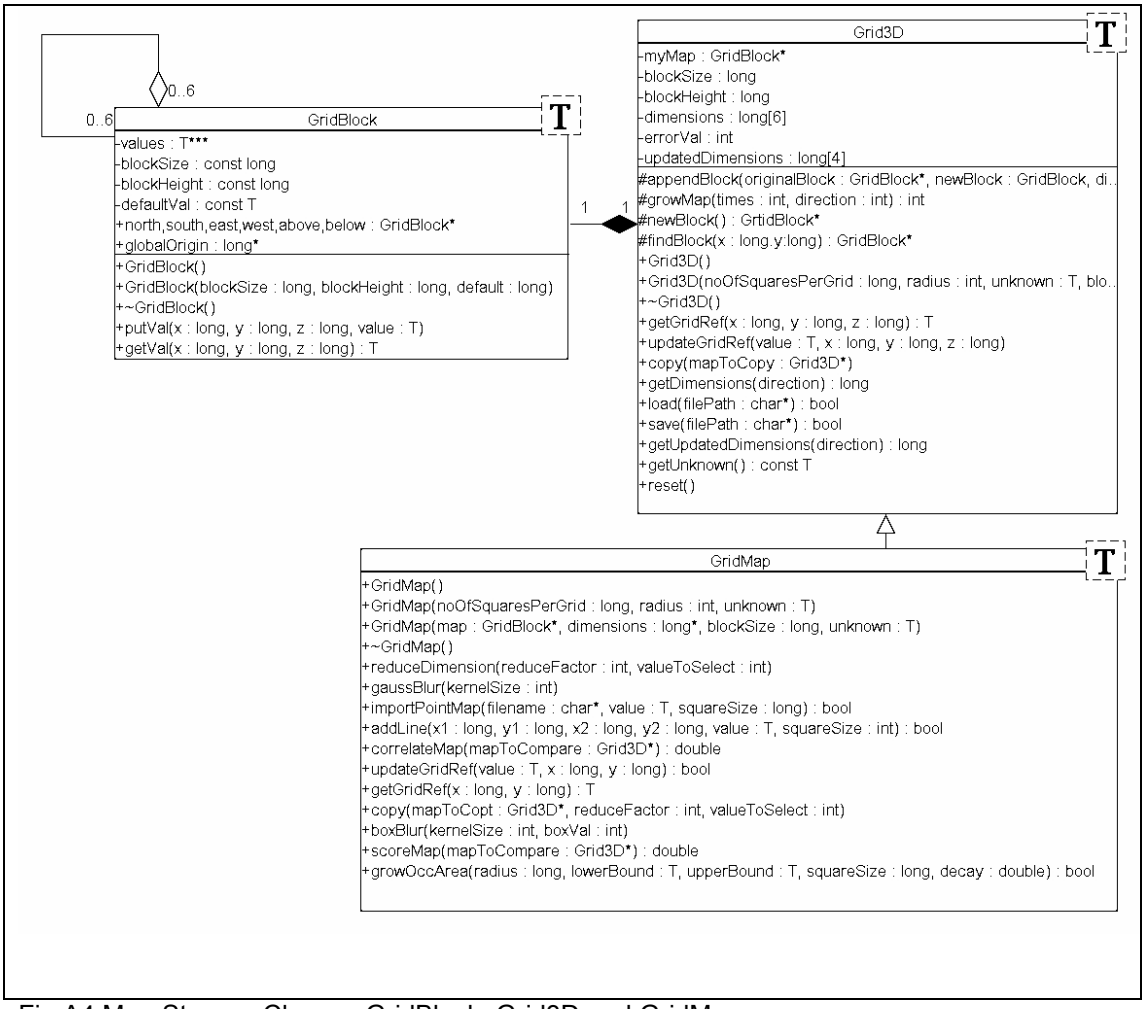


Fig A4 Map Storage Classes GridBlock, Grid3D and GridMap.

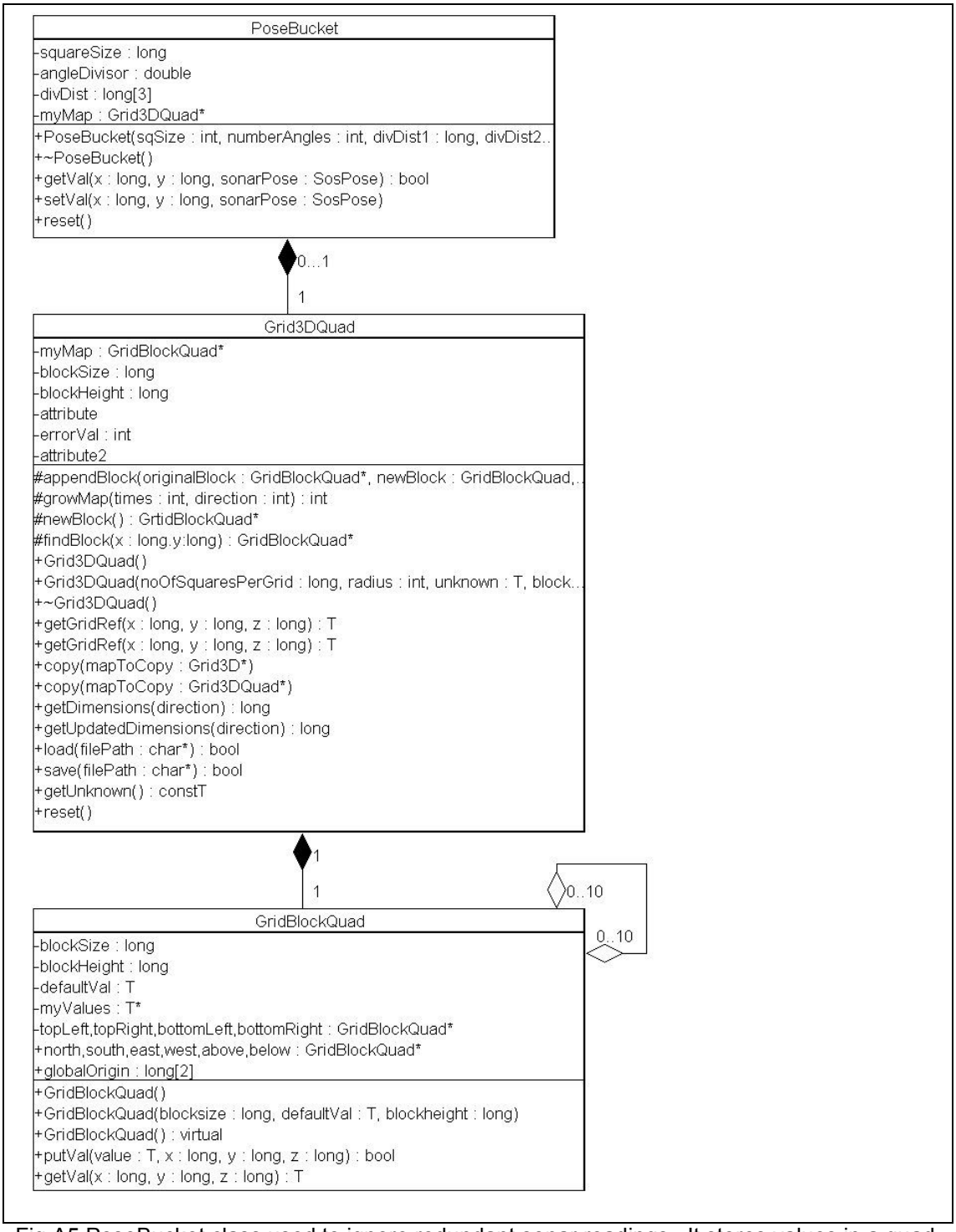


Fig A5 PoseBucket class used to ignore redundant sonar readings. It stores values in a quad tree representation to offset the otherwise prohibitive amount of memory necessary. The quad tree storage is implemented by the GridBlockQuad and Grid3DQuad classes.

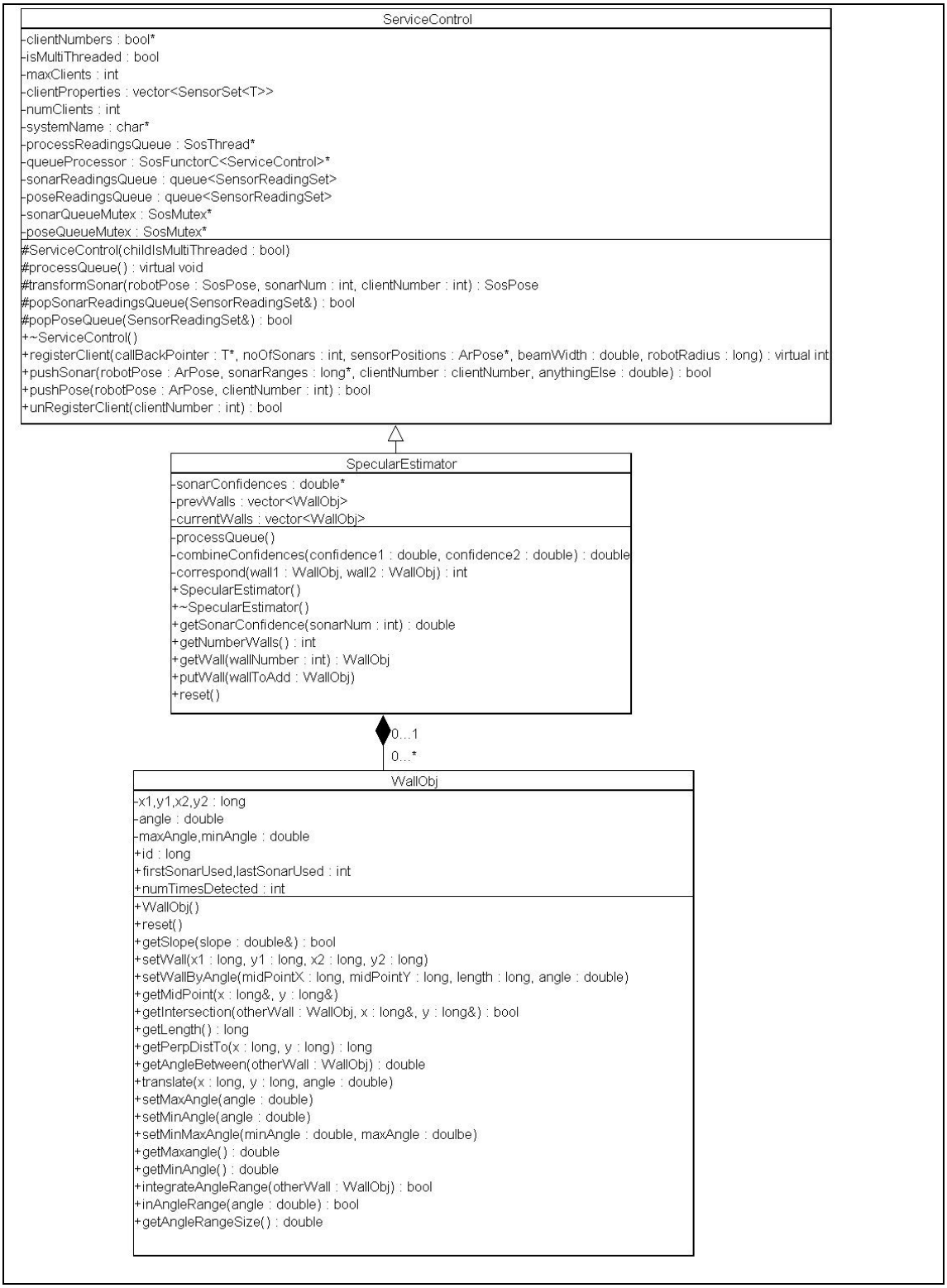


Fig A6 The *SpecularEstimator* class which performs feature prediction to filter noisy sonar readings. It is a child of the *ServiceControl* class, but unlike the *Mapping* class, which is also a child of *ServiceControl*, it operates in a single thread i.e. synchronously. The *WallObj* class is used to store all necessary information for each predicted feature in the environment.

A2 Experimentation Results

Chapter 6 presented an averaged version of the experimentation results in order to display trends and extract meaning from the relatively large body of data. In this section the results in their entirety are displayed, grouped by the environment the experiment was performed in.

A2.1 AIC Simulated Environment Experimentation Results

Mapping Method	Pose Buckets	Feature Prediction	Match All %	Match Occ %	Correlation With Ideal	% Invalid Paths	% False Positives	% Not Completed Paths	% Of Ideal Path Cost
ME85mod	Yes	Yes	2.96 %	2.30 %	88.75%	15.99 %	0.00%	0.00%	100.12 %
ME88mod	Yes	Yes	3.16 %	2.90 %	87.81%	24.11 %	0.00%	0.00%	100.07 %
K97mod	Yes	Yes	3.18 %	1.99 %	88.23%	14.73 %	0.00%	0.00%	100.13 %
ME85mod	Yes	No	4.03 %	3.55 %	84.13%	27.70 %	0.00%	0.00%	100.08 %
K97	Yes	No	5.63 %	2.53 %	81.04%	0.00%	0.00%	5.00%	100.38 %
ME88mod	Yes	No	4.52 %	4.24 %	82.23%	34.44 %	0.00%	0.00%	100.17 %
K97mod	Yes	No	3.19 %	2.38 %	87.72%	24.45 %	0.00%	0.00%	100.09 %
K97mod	No	Yes	2.78 %	2.38 %	89.34%	27.78 %	0.00%	0.00%	100.08 %
ME85mod	No	Yes	3.47 %	3.30 %	86.70%	37.42 %	0.00%	0.00%	100.05 %
ME88mod	No	Yes	4.03 %	3.81 %	84.37%	31.99 %	0.00%	1.25%	100.05 %
ME88	No	No	8.95 %	7.99 %	67.70%	64.51 %	3.75%	0.00%	100.00 %
K97	No	No	4.49 %	3.31 %	82.47%	16.39 %	1.25%	2.50%	100.58 %

Mapping Method	Pose Buckets	Feature Prediction	Match All %	Match Occ %	Correlation With Ideal	% Invalid Paths	% False Positives	% Not Completed Paths	% Of Ideal Path Cost
K97mod	No	No	6.12 %	5.57 %	76.59%	61.00 %	1.25%	0.00%	100.08 %
ME88mod	No	No	7.30 %	6.66 %	72.49%	59.03 %	2.50%	0.00%	100.00 %
ME85	No	No	8.76 %	7.00 %	68.86%	65.76 %	2.50%	0.00%	100.00 %
ME85mod	No	No	6.93 %	6.14 %	74.34%	58.99 %	1.25%	0.00%	100.07 %

A2.2 Corridor Simulated Environment Experimentation Results

Mapping Method	Pose Buckets	Feature Prediction	Match All %	Match Occ %	Correlation With Ideal	% Invalid Paths	% False Positives	% Not Completed Paths	% Of Ideal Path Cost
ME85mod	Yes	Yes	5.68%	3.44%	82.22%	28.66%	0.00%	0.00%	100.00%
ME88mod	Yes	Yes	6.05%	3.67%	81.51%	30.28%	0.00%	0.00%	100.00%
K97mod	Yes	Yes	6.42%	3.33%	80.47%	12.30%	0.00%	0.00%	100.00%
K97mod	Yes	No	6.19%	3.66%	80.55%	19.43%	0.00%	0.00%	100.00%
K97	Yes	No	9.44%	4.41%	74.06%	0.00%	0.00%	0.00%	100.00%
ME85mod	Yes	No	7.38%	4.61%	76.66%	31.98%	0.00%	0.00%	100.00%
ME88Mod	Yes	No	7.72%	4.81%	76.21%	42.01%	0.00%	0.00%	100.00%
ME85mod	No	Yes	7.81%	4.49%	75.72%	46.52%	0.00%	0.00%	100.00%
ME88mod	No	Yes	9.02%	5.24%	72.72%	49.55%	0.00%	0.00%	100.00%
K97mod	No	Yes	6.34%	3.73%	80.15%	38.29%	0.00%	0.00%	100.00%
ME88mod	No	No	13.21%	6.99%	63.61%	56.40%	0.00%	0.00%	100.00%
ME88	No	No	17.74%	9.24%	53.23%	74.68%	0.00%	0.00%	100.00%
ME85	No	No	18.34%	8.24%	54.62%	57.73%	0.00%	0.00%	100.00%
K97mod	No	No	11.87%	6.71%	66.46%	67.96%	0.00%	0.00%	100.00%
ME85mod	No	No	14.85%	7.44%	59.80%	51.50%	0.00%	0.00%	100.00%
K97	No	No	8.25%	5.37	74.46%	15.68	0.00%	0.00%	100.00

Mapping Method	Pose Buckets	Feature Prediction	Match All %	Match Occ %	Correlation With Ideal	% Invalid Paths	% False Positives	% Not Completed Paths	% Of Ideal Path Cost
				%		%			%

A2.3 CSIS Building 1st Floor Simulated Environment Experimentation Results

Mapping Method	Pose Buckets	Feature Prediction	Match All %	Match Occ %	Correlation With Ideal	% Invalid Paths	% False Positives	% Not Completed Paths	% Of Ideal Path Cost
ME85mod	Yes	Yes	3.50%	3.77%	83.91%	15.52%	0.00%	3.70%	100.05%
K97mod	Yes	Yes	3.33%	3.40%	85.00%	1.95%	0.00%	3.70%	100.08%
ME88mod	Yes	Yes	3.52%	3.82%	83.90%	22.22%	0.00%	3.70%	100.07%
K97	Yes	No	4.38%	3.50%	81.22%	3.97%	0.00%	11.11%	100.08%
K97mod	Yes	No	3.56%	3.81%	83.71%	13.47%	0.00%	3.70%	100.12%
ME85mod	Yes	No	6.08%	5.72%	72.91%	43.09%	3.70%	3.70%	100.01%
ME88mod	Yes	No	6.69%	6.83%	70.29%	42.65%	3.70%	3.70%	100.00%
ME85mod	No	Yes	3.94%	4.35%	81.83%	39.50%	0.00%	3.70%	100.05%
ME88mod	No	Yes	3.99%	4.34%	81.81%	38.41%	0.00%	3.70%	100.07%
K97mod	No	Yes	3.33%	3.66%	84.70%	14.91%	0.00%	3.70%	100.09%
K97mod	No	No	9.69%	8.76%	59.81%	65.64%	3.70%	3.70%	100.00%
ME88mod	No	No	12.12%	10.79%	52.55%	65.31%	3.70%	3.70%	100.02%
K97	No	No	4.58%	4.76%	79.05%	9.55%	0.00%	7.41%	100.26%
ME85	No	No	13.31%	9.57%	51.57%	75.88%	7.41%	3.70%	100.00%
ME88	No	No	14.30%	12.49%	46.81%	67.38%	3.70%	3.70%	100.02%
ME85mo	No	No	12.02	8.86%	54.60%	74.03	7.41%	3.70%	100.00

Mapping Method	Pose Buckets	Feature Prediction	Match All %	Match Occ %	Correlation With Ideal	% Invalid Paths	% False Positives	% Not Completed Paths	% Of Ideal Path Cost
d			%			%			%

A2.4 Star Simulated Environment Experimentation Results

Mapping Method	Pose Buckets	Feature Prediction	Match All %	Match Occ %	Correlation With Ideal	% Invalid Paths	% False Positives	% Not Completed Paths	% Of Ideal Path Cost
ME85mod	Yes	Yes	8.61%	6.70%	76.07%	0.00%	0.00%	0.00%	100.50%
K97mod	Yes	Yes	8.79%	6.77%	75.95%	0.00%	0.00%	3.70%	100.00%
ME88mod	Yes	Yes	11.97%	10.51%	65.97%	12.50%	0.00%	0.00%	100.00%
ME88mod	Yes	No	16.63%	14.25%	56.07%	60.66%	0.00%	0.00%	100.00%
ME85mod	Yes	No	11.58%	9.02%	68.29%	1.74%	0.00%	10.71%	100.03%
K97	Yes	No	15.02%	10.74%	62.91%	0.00%	0.00%	39.29%	101.41%
K97mod	Yes	No	10.11%	8.36%	72.30%	97.67%	0.00%	0.00%	100.00%
ME85mod	No	Yes	10.83%	10.07%	70.87%	33.92%	0.00%	0.00%	100.00%
ME88mod	No	Yes	13.44%	11.88%	64.04%	38.12%	0.00%	0.00%	100.00%
K97mod	No	Yes	11.16%	10.19%	69.10%	91.21%	0.00%	0.00%	100.00%
ME85	No	No	26.70%	18.49%	43.03%	90.17%	0.00%	0.00%	100.00%
ME88mod	No	No	19.89%	16.14%	52.62%	79.40%	0.00%	0.00%	100.00%
K97	No	No	24.09%	20.48%	45.94%	0.00%	0.00%	64.28%	107.13%
K97mod	No	No	20.75%	17.77%	50.27%	97.78%	0.00%	0.00%	100.00%
ME85mod	No	No	18.03%	14.04%	57.52%	61.77%	0.00%	0.00%	100.00%
ME88	No	No	26.10%	20.41%	43.54%	81.68%	0.00%	0.00%	100.32%

A2.5 Star Real-World Environment Experimentation Results

Mapping Method	Pose Buckets	Feature Prediction	Match All %	Match Occ %	Correlation With Ideal	% Invalid Paths	% False Positives	% Not Completed Paths	% Path Cost
K97	No	No	585.55	454.96	48.52%	26.71%	7.14%	14.28%	102.04%
K97	Yes	No	344.26	257.63	67.08%	9.23%	0.00%	10.71%	100.05%
K97mod	No	No	704.47	461.34	49.77%	66.26%	0.00%	0.00%	100.00%
K97mod	No	Yes	441.22	366.55	61.61%	59.22%	0.00%	0.00%	100.00%
K97mod	Yes	No	339.62	291.94	67.91%	54.11%	0.00%	0.00%	100.00%
K97mod	Yes	Yes	296.59	243.13	71.17%	19.55%	0.00%	0.00%	100.00%
ME85	No	No	889.3	499.06	43.08%	60.00%	0.00%	0.00%	100.10%
ME85mod	No	No	445.3	395.94	59.53%	29.59%	0.00%	0.00%	100.10%
ME85mod	No	Yes	393.57	344.14	62.14%	37.47%	0.00%	0.00%	100.10%
ME85mod	Yes	No	395.05	314.36	62.61%	24.39%	0.00%	3.57%	100.39%
ME85mod	Yes	Yes	332.48	270.57	67.16%	24.01%	0.00%	3.57%	103.43%
ME88	No	No	806.71	471.2	43.18%	72.29%	0.00%	0.00%	102.04%
ME88mod	No	No	471.14	419.62	57.62%	37.87%	0.00%	0.00%	100.00%
ME88mod	No	Yes	440.55	392.68	54.88%	42.94%	0.00%	0.00%	100.00%
ME88mod	Yes	No	464.8	389.07	59.51%	49.74%	0.00%	0.00%	100.00%
ME88mod	Yes	Yes	414.38	364.36	61.67%	40.77%	0.00%	0.00%	100.00%

A3 Sample Maps Generated In Simulated Experimentation

The following six maps were generated in a simulated run around the CSIS building in the University of Limerick All six of the map building systems tested in experimentation are used, without Feature Prediction and without Pose Buckets.

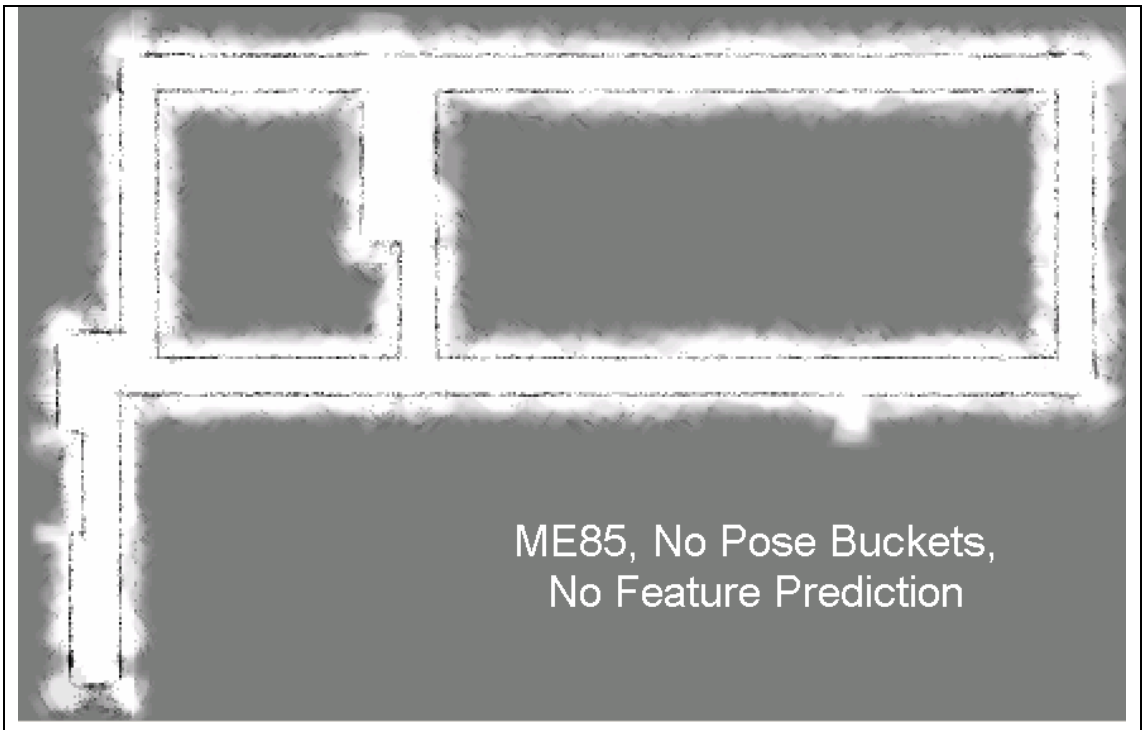


Fig A7 Map generated of the *eCSBsim* environment by the *ME85* map building system, which uses neither pose bucket nor feature prediction.

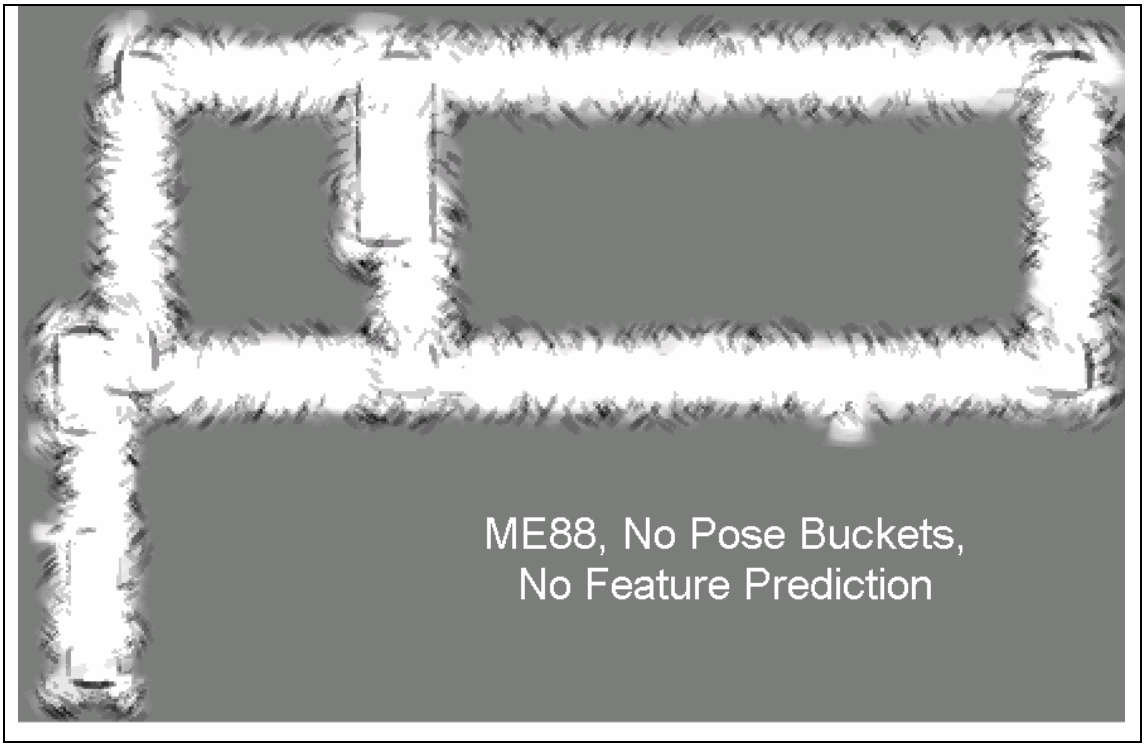


Fig A8 Map generated of the *eCSBsim* environment by the *ME88* map building system, which uses neither pose bucket nor feature prediction.

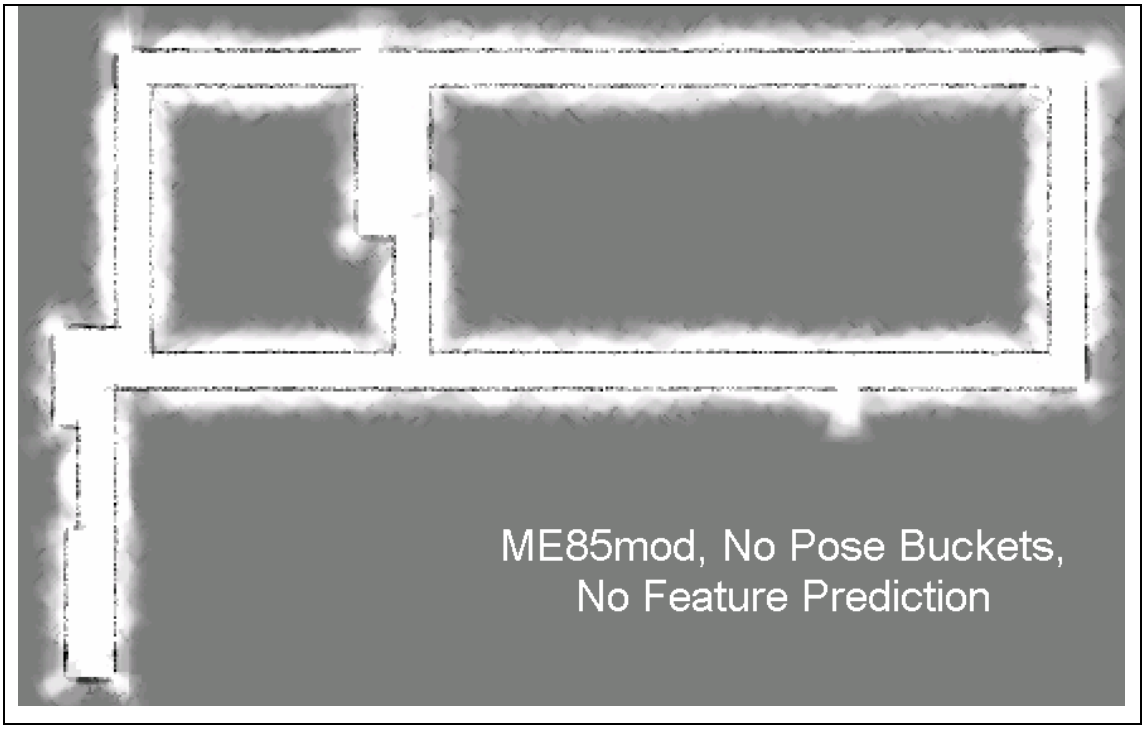


Fig A9 Map generated of the *eCSBsim* environment by the *ME85mod* map building system, with feature prediction and pose buckets disabled.

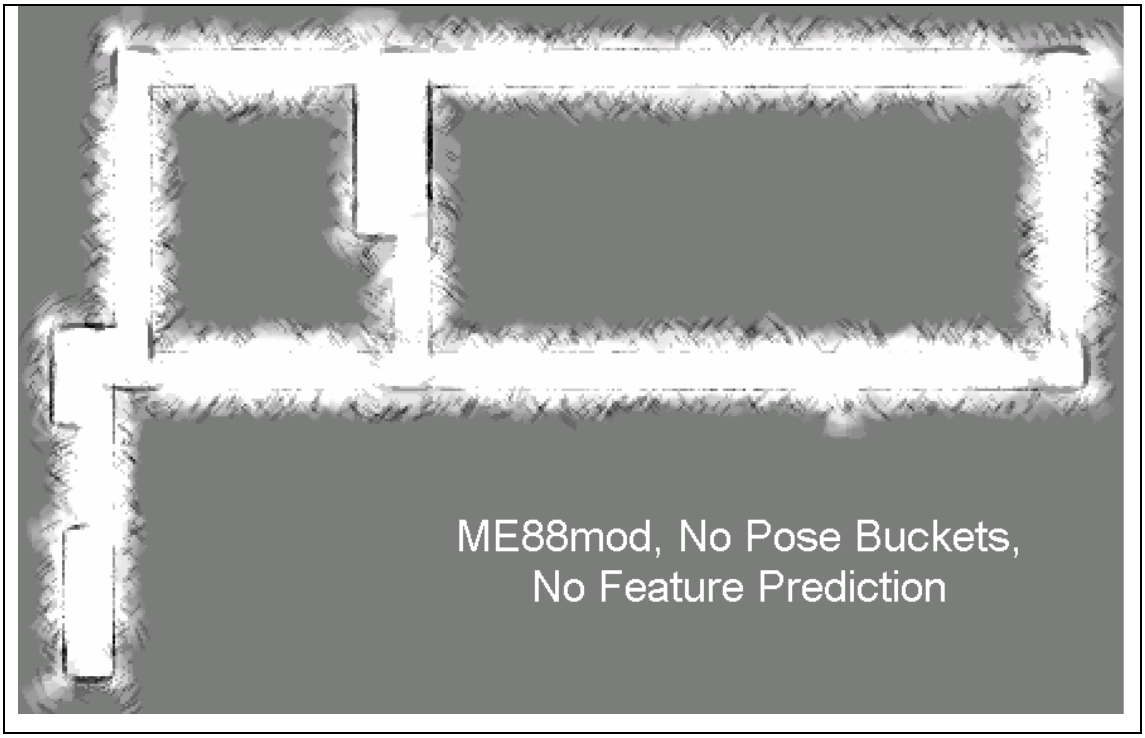


Fig A10 Map generated of the *eCSBsim* environment by the *ME88mod* map building system, with feature prediction and pose buckets disabled.

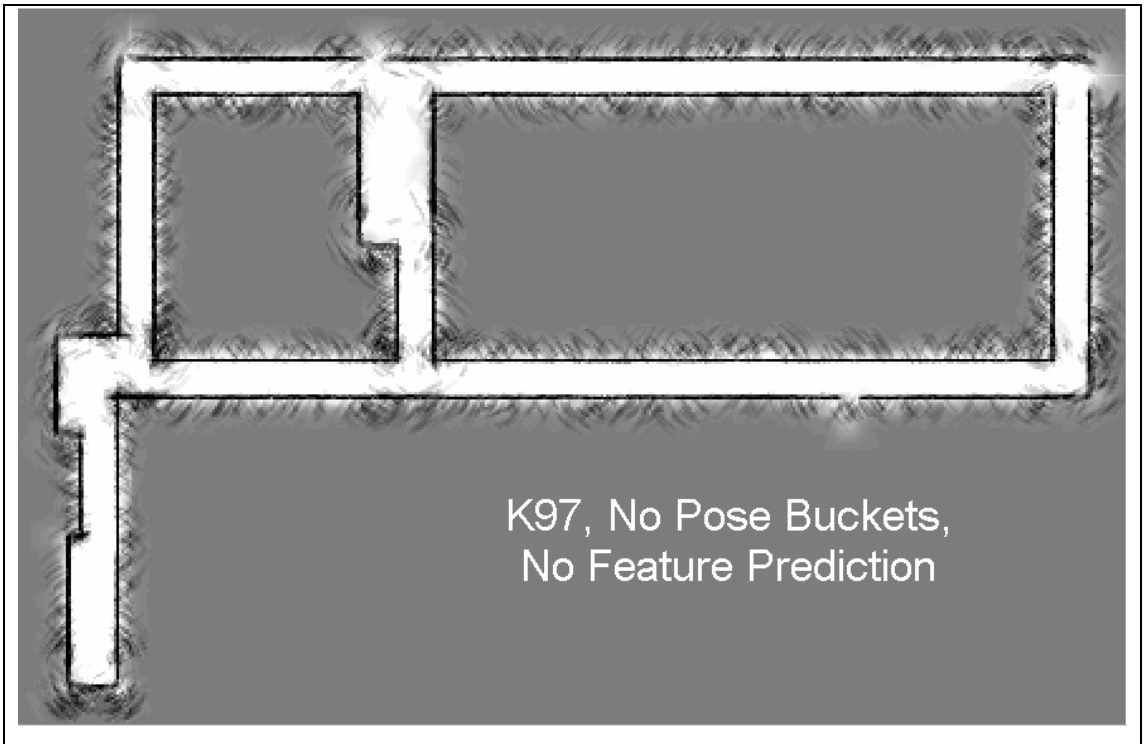


Fig A11 Map generated of the *eCSBsim* environment by the *K97* map building system, with pose buckets disabled. *K97* does not use feature prediction.

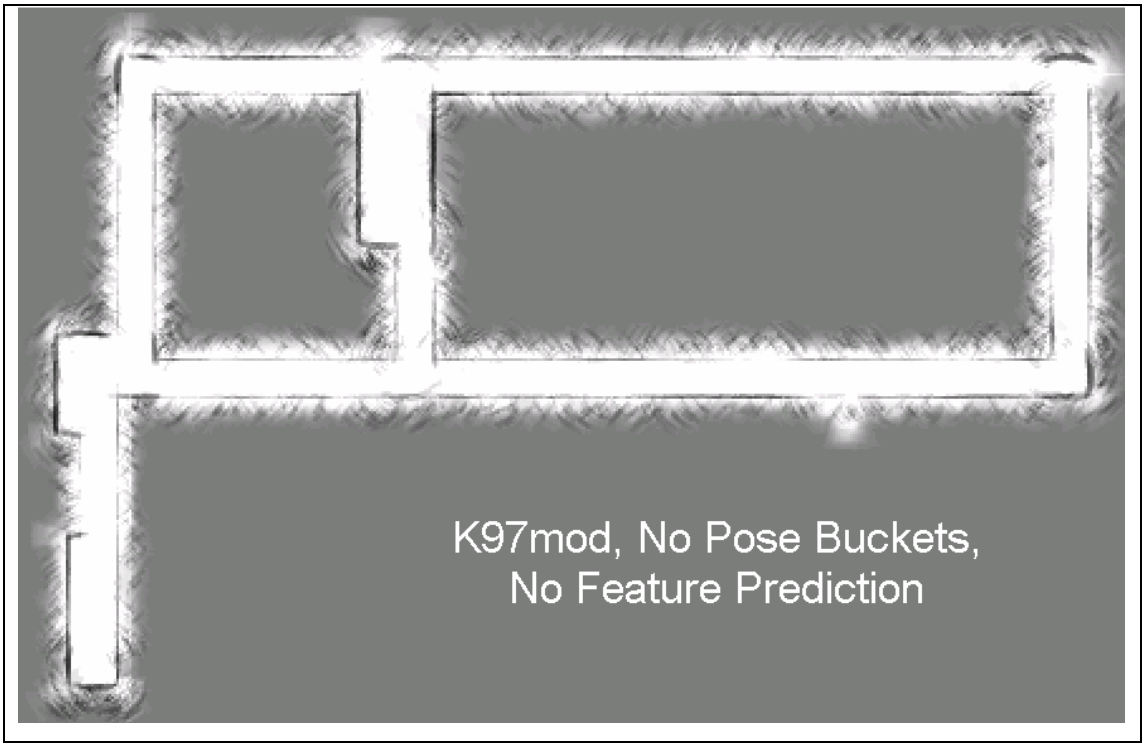


Fig A12 Map generated of the *eCSBsim* environment by the *K97mod* map building system, with feature prediction and pose buckets disabled.

These final four maps are designed to show the effect of Pose Buckets and Feature Prediction on the quality of the map generated. For this example, the *ME85mod* map building system was used to generate a map of the *eCSBsim* environment.

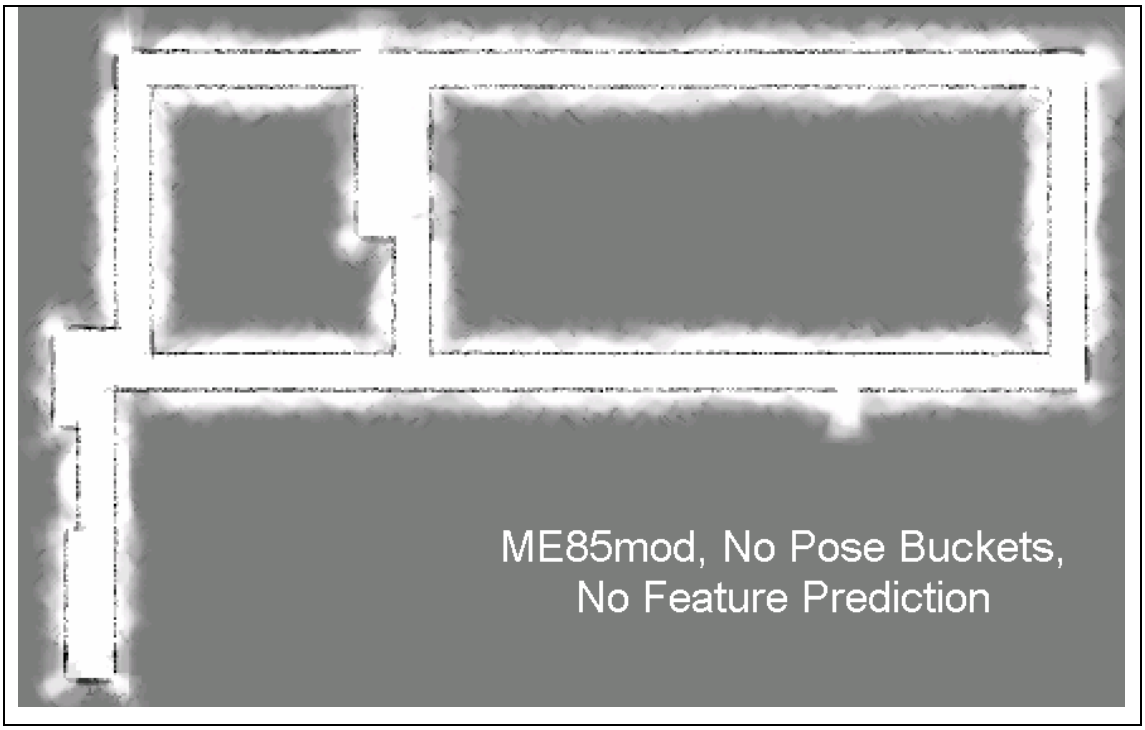


Fig A13 Map generated of the *eCSBsim* environment by the *ME85mod* map building system, with feature prediction and pose buckets disabled.

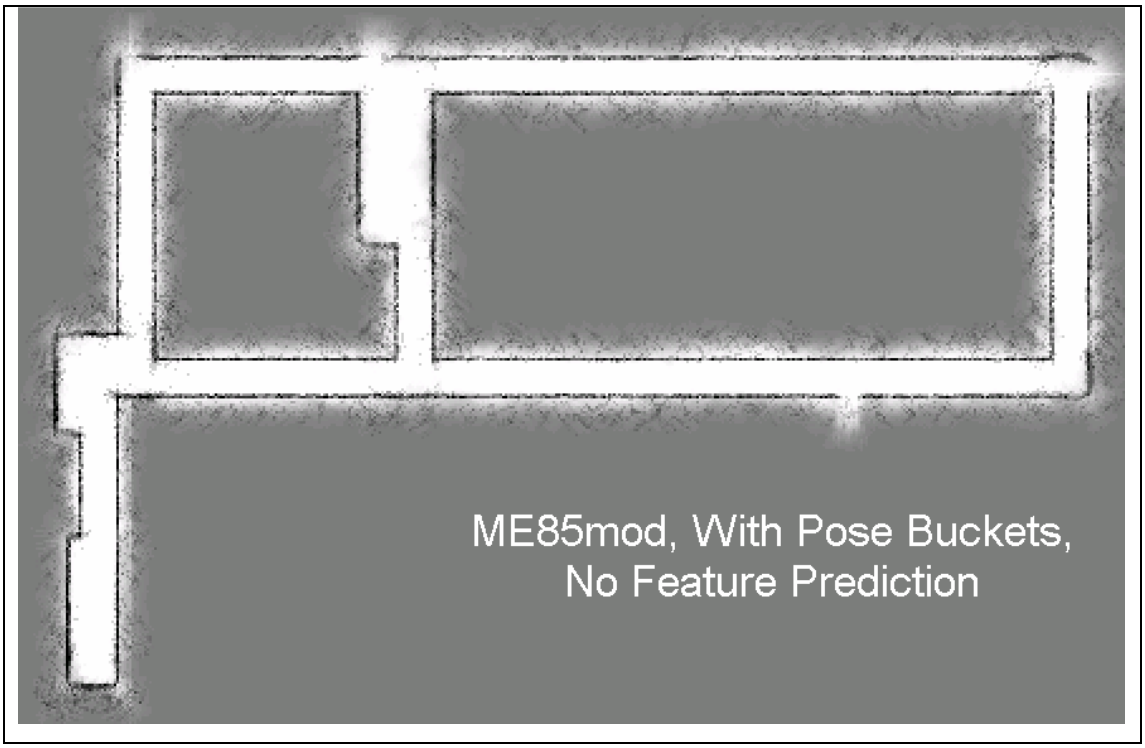


Fig A14 Map generated of the *eCSBsim* environment by the *ME85mod* map building system, with feature prediction disabled and pose buckets enabled.

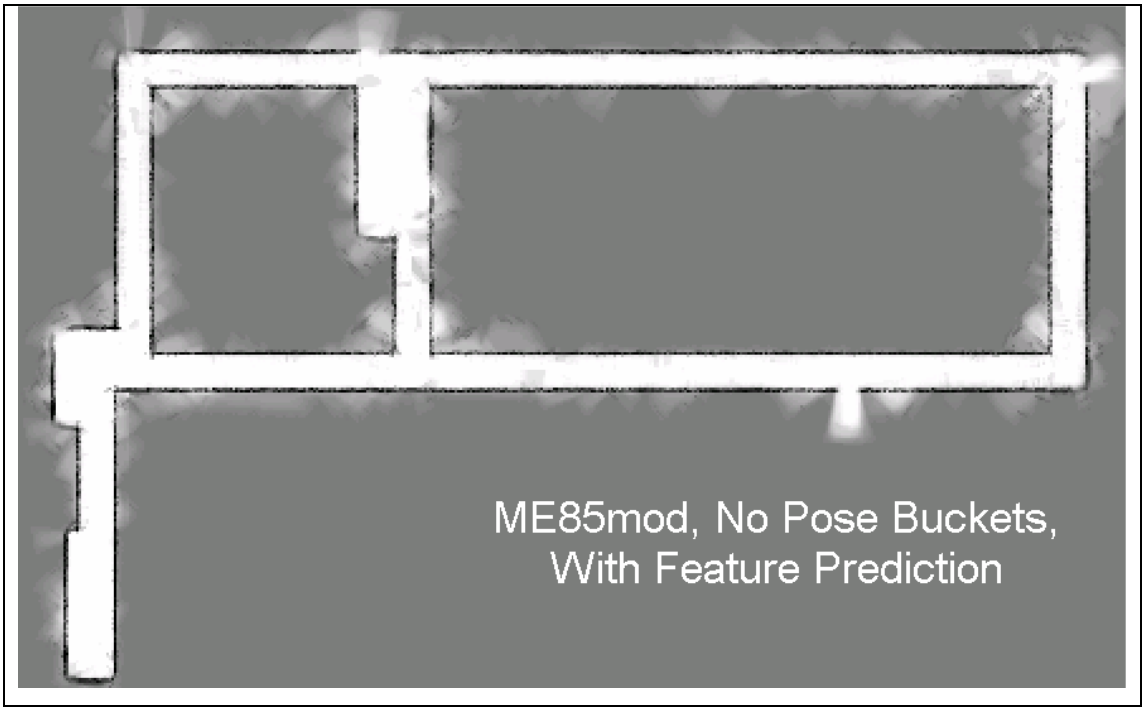


Fig A15 Map generated of the *eCSBsim* environment by the *ME85mod* map building system, with feature prediction enabled and pose buckets disabled.

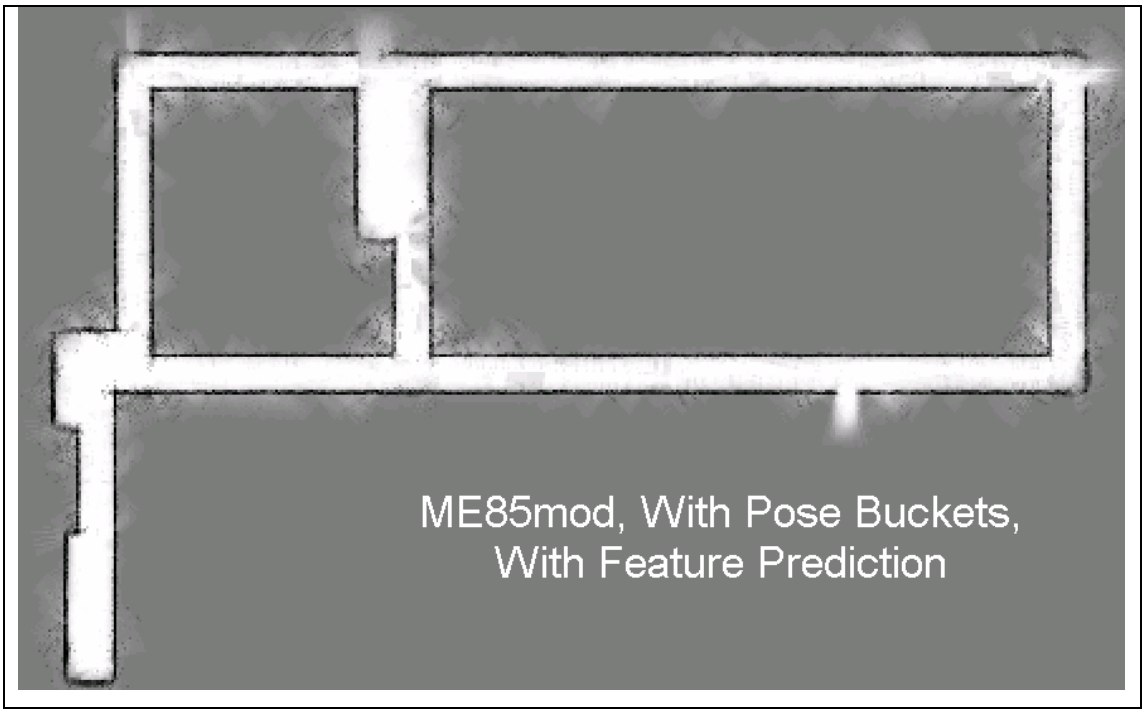


Fig A15 Map generated of the *eCSBsim* environment by the *ME85mod* map building system, with feature prediction and pose buckets enabled.

A4 Introduction to Probability

An experiment is any action or process that generates observations, and a sample space of an experiment is the set of all possible outcomes of that experiment. An event is any collection or subset of outcomes contained in the sample space. While an experiment that is performed just a few times may return some very erratic results, as the number of test runs, *n*, gets arbitrarily large, the results smooth out and approach what is called the *limiting relative frequency*. Probabilities can then be assigned to events based of their limiting relative frequency. However, since the probability of an event relies on its l.r.f, its applicability is limited to experiments that are repeatable.

If an event is certain to happen, it is said to have a probability of 1. If it is certain not to happen, it has a probability of 0. And if any particular event has a probability, P(A), the probability of it not occurring is 1-P(A).

$$P(A) = 1 - P(A')$$

The probability of two non-mutually exclusive events occurring is:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

Mutually exclusive means that A and B have no outcomes in common, therefore:

$$P(A \cap B) = 0$$

If all outcomes are equally likely, determining the probability of some event is reduced to simple counting.

$$P(A) = N(A) / N$$

Where N(A) is the number of outcomes contained in event A.

Conditional Probability

The fact that one event, B, has occurred can sometimes affect the probability of another event, A, occurring. This is represented with P(A | B), which means the conditional probability of A given that the event B has occurred.

$$P(A \mid B) = P(A \cap B) / P(B)$$

And it follows that $P(A \cap B) = P(A \mid B) * P(B)$.

Bayes Theorem

The law of total probability states that for any event B, and events $A_1, \ldots A_n$ which are mutually exclusive and exhaustive,

$$P(B) = \sum P(B \mid A_i)\, P(A_i).$$

Bayes uses this to prove that for A_1, …. A_n, a collection of *n* mutually exclusive events with $P(A_i) > 0$ for i = 1, … n, for any other event B for which P(B) > 0:

$$P(A_k \mid B) = \frac{P(B \mid A_k)P(A_k)}{\sum P(B \mid A_k) * P(A_i)}$$

Independence

Two events are said to be independent if P(A | B) = P(A), and are dependent otherwise. If two events are mutually exclusive, they cannot be independent, since when A and B are mutually exclusive, the information that A has occurred says something about B (it cannot have occurred), so independence is impossible. A and B are independent only if:

$P(A \cap B) = P(A) * P(B)$

The Expected Value of X

The expected, or mean (average) value of a random variable X, denoted by E(X) is

$$\sum_{x \in D} x * p(y)$$

The Expected Value of h(X)

Often we will be interested in the expected value of some function *h*(X), rather than X itself. If the random variable X has a set of possible values *D* and a p.m.f. p(x), then the expected value of any function *h*(X) is computed by:

$$E[h(X)] = \sum_{D} h(x) * p(x)$$

But if this function is a linear function, such as

aX + b

then E(aX + b) = a * E(X) + b

The Variance of X

While the expected value of X determines where the probability distribution is centred, the variance measures how great the spread of the distribution is, or how dispersed it is.

The variance of X, $V(X) = \sum_{D} (x - \mu)^2 * p(x) = E\left[(X - \mu)^2\right]$

For the random variable X with a probability mass function p(x) and expected value μ. The variance can also be denoted by $\sigma^{2.}$

The standard deviation $\sigma = \sqrt{\sigma^2}$

A quicker way to calculate the variance, V(X), is

$$V(X) = E(X^2) - [E(X)]^2$$

Which reduces the number of calculations considerably.

The Normal Distribution

A continuous random variable *X* is said to have a **normal distribution** with parameters μ and σ where $-\infty < \mu < \infty$ and $0 < \sigma$, if the probability density function of *X* is

$$f(x;\mu;\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/(2\sigma^2)} \quad -\infty < x < \infty$$

The statement that X is normally distributed is often abbreviated to $X \sim N(\mu,\sigma^2)$. A normal curve is bell shaped, and therefore symmetrical. The mean, μ, is at the peak of the curve, and the standard deviation from the mean, σ, is the distance from μ to the inflection points of the curve.

The Standard Normal Distribution

As it can become quite complex to compute a normal curve, we use what is called the Standard Normal Distribution, which has $\mu = 0$ and $\sigma = 1$. A random variable, *z*, that has a standard normal distribution is called s standard normal random variable.

$$f(z;0,1) = \frac{1}{\sqrt{2\pi}} e^{-z^2/2} \quad -\infty < z < \infty$$

The cumulative distribution function of *Z* is

$P(Z \le z) = \int_{-\infty}^{z} f(y;0,1)dy$, which we denote as $\Phi(z)$.

Percentiles of the Standard Normal Distribution

For any number, *p*, between zero and one, the standard normal distribution tables can be used to find the (100*p*)th percentile of the standard normal distribution. For

example, find which point on the curve has 90% of the area under the curve to its left, and 10% to the right.

z_α Notation

We will use z_α to denote the value on the horizontal axis which has α of the area under the curve to the right of z_α. z_α is often referred to as a z critical value, and can be found in standard normal distribution tables.

Non-standard Normal Distributions

When a normal distribution is not standard, it can be computed by standardising its values. Subtracting the mean, μ from the random variable X shifts the mean to zero, and dividing the result by σ scales the variable so that the standard deviation is 1 instead of σ. Therefore, if X is a random variable with a normal distribution,

$$Z = (X - \mu)/\sigma$$

is a standard normal random variable.

In order to calculate the (100*p*)th percentile of a normal distribution, first calculate the (100*p*)th percentile for the standard normal distribution, multiply it by the standard deviation of the distribution, σ, and add the mean.

(100*p*)th percentile for normal (μ,σ)=μ + [(100*p*)th percentile for standard normal]* σ

Approximating Discrete Populations with the Normal Distribution

The normal distribution can be used to approximate discrete populations. Discrete populations can be modelled using a histogram, with the rectangles centred at integers. To approximate the area greater than a particular integer, e.g. 5, it is necessary to measure the area to the right of 4.5 i.e. $P(X \geq 4.5)$.

Joint Probability Distributions and Random Samples

It is often the case that more than one random variable will be of interest in an experiment. It follows that we must model the joint probability distribution of these random variables.

Covariance

When two random variables are dependent, it is often of interest to what degree they are related to one another. The **covariance** between two random variables X and Y is

$$Cov(X,Y) = E\left[(X-\mu_x)(Y-\mu_y)\right] = \begin{cases} \sum_x \sum_y (x-\mu_x)(y-\mu_y)p(x,y) & X,Y \text{ discrete} \\ \int_{-\infty}^{\infty} (x-\mu_x)(y-\mu_y)f(x,y)\,dx\,dy & X,Y \text{ continuous} \end{cases}$$

The problem with covariance is that its result depends strongly on the units of measurement used in the experiment. To remedy this, we use the **correlation coefficient**.

Correlation

The correlation coefficient of X and Y, denoted Corr(X, Y), $\rho_{x,y}$ or ρ, is

$$\rho_{x,y} = \frac{Cov(X,Y)}{\sigma_X * \sigma_y}$$

Some important properties of correlation are that

1. The correlation coefficient is not affected by a linear change in the units of measurement, i.e. Corr($aX + b$, $cY + d$) = Corr(X, Y)

For and two random variables X and Y, $-1 \le \text{Corr}(X, Y) \le 1$

A5 Robot Simulators

Here follows a selection of the mobile robot simulators currently available.

1. ARC – Autonomous Robot Controller
    - May be connected to an external controller programmed by the user
    - Can control multiple robots interacting in the environment.
    - Geometric, physical and dynamic configuration of each robot.
    - Graphic visualisation of the virtual world, robots and database.
    - Permits insertion of various types of static and rotary sensors into the robot.
    - Objects and obstacles with geometry and physical characteristics.
    - Robots are limited to movement in 2 dimensions.
    - The robots must be ones with 2 independent motorised wheels.
    - Supports Win 98/2000, NT.
    - Freeware

2. BugWorks – 2D Robot Simulator and eTutor
    - Drag and drop interface.
    - Integrated tutor
    - Designed with students with no programming experience in mind.
    - Used language, 'BugScript'.
    - Available for Java 1 and 2, runs as an applet.

3. EyeSim – Eyebot simulator.
    - Allows testing and debugging of EyeBot applications on a Unix platform.
    - Allows robot to drive in simulated 2D environment, with all its sensors and actuators.
    - Multiple robot simulator, simulated sensors – bumper, PSD, IR-Proxy.
    - Simulated actuators: v- omega differential driving.
    - Parameter settings can be changed via menu for errors in velocity, sensor ranges, robot size.
    - Eyebot is a controller for robots with wheels, walking or flying robots.

4. WebBots 3
    - 3D robot simulator which simulates any robot using two-wheeled differential steering.
    - Model robots and world using VRML-style language.
    - Design behaviour of robot in C++.
    - Distance sensors: infrared and sonars.
    - Emitters: infrared and radio.
    - Receivers: infrared and radio.
    - Supports Linux and PC.
    - Design supervisor program to monitor simulations in C++.
    - Simulation cannot be transferred to a real robot.

5. Millbots Simulator
    - Written in Java – multi-platform support.
    - Dynamically configurable control and interface – drag and drop robots during operation, add obstacles during operation.
    - Multiple robots with different configurations.
    - Robot can be local or distributed.
    - Support for hierarchical teams.
    - Supports timed events.

6. Flat
    - Simulates an RWI robot – a circular, sonar-guided robot – in a flat 2D world, with simulated sonar and laser rangefinder sensors.
    - Contains an accurate physical simulator and accurate sonar and laser rangefinder simulators.
    - The simulated robot has internal odometry, one or two range finders, and twelve sonar sensors.
    - Can design own 2D floor plan to simulate a building or space.
    - Define the robot environment, including multi-level worlds.
    - Sensors include optional noise models.
    - LISP program Flat Client provides all low-level command to control the robot.
    - Supports Sun and x86 Linux machines.

7. MobotSim – Mobile Robot Simulator
    - Supports Win 9x/Me/NT/2000
    - 2D simulation of differential drive mobile robots
    - Unlimited number of Mobots and obstacles.
    - Flexible configuration of Mobots – platform diameter, wheels diameter, number of sensors, and angle between sensors.
    - Configuration of ranging sensors – radiation cone, range, misreading percentage.
    - Easy integration of 3rd party ActiveX controls and .dll files.
    - Can add specific tools to the BASIC editor to make use of Fuzzy Logic, Genetic Algorithms, Neural Networks etc.
    - Motor errors and wheel slippage not simulated.
    - Angle of incidence and surface features of objects not considered.

8. MOBS – Mobile Robot Simulator
    - Fully 3D simulation system.
    - Can be connected to a robot application program even without re-compilation of the application.
    - Sensors modelled are: odometry, bumpers, sonar, camera view.
    - Applications include – following road markings, autonomous cars in highway traffic, planning and coordination of multiple autonomous agents in factory environments, convoy driving.

9. Netmaze
    - 3D simulation environment.
    - Client – server architecture.
    - Real-time interactions.
    - Can be used for robot navigation, behaviour, artificial vision, multi-agent systems etc.
    - Environment consists of walls only.
    - 256 sensors placed around robot to indicate collisions with walls or other agents.

- Range finder covers the visual are – indicates distance between objects and the agent.
- Runs on Linux with X11 and OpenGL.

10. Player/Stage – Networked Transducer Interface / Multiple robot simulator
    - Player is a device server that provides an interface to sensors and actuators.
    - Robot control programs can be written in any language.
    - Support for multiple concurrent connections to devices.
    - Supports ActivMedia Pioneer 2-DX robot, SICK LMS-200 laser rangefinder, Sony EVID-30 camera, sound hardware, ACTS colour vision system and the Festival speech synthesis system.
    - Runs on Linux and Unix.
    - Stage is a scaleable multiple robot simulator.
    - 2D bitmapped.
    - It provides sonar, scanning laser rangefinder, pan-tilt-zoom camera and odometry.
    - Little or no changes required to move from simulation to hardware.
    - Free to distribute and modify.

The UL Robotics Group Architecture Manual

August 2003

Shane O'Sullivan

UL Robotics Group Software Architecture Manual

The architecture used by the UL robotics group to control mobile robots is designed to be as flexible as possible, while at the same time standardising an interface to a family of services that can be used to control a mobile robot. The architecture is based upon the use of modules, either in isolation or in conjunction with one or more other modules. Each module is responsible for a single task, be it map building, localisation, pursuit, evasion or whatever. These modules must be coordinated by a central controller. This ensures that modularity is preserved, and that no one module relies upon another in order for it to operate, though it may function better if another module is also in use – for example, a mapping service will work better if a localisation service is constantly confirming its position, but it can work without it.

While the standard configuration will be that the central controller will be the client of the service, it is also possible for services to be clients of other services. An example of this is the *ME88mod* class which, while being a mapping service, is also a client of the *SpecularEstimator* class. The mechanism for achieving this is detailed at the after the description of the *ServiceControl* class.

ServiceControl

The primary object in the architecture is the *ServiceControl* class. Each service in the architecture inherits from the *ServiceControl* class. The only requirement is that the service defines a function called *processQueue* which will use the information sent to the module. The *ServiceControl* class has a number of features:

1. Multiple client support, with each client having an individual profile that guides how the service uses the information provided to it by that client.
2. An information Repository
3. Control of threading of a service.
4. Support of different callback interfaces through the use of templates.

ServiceControl's Multiple Client Support

The *ServiceControl* class offers support for multiple clients of a service through the use of the *registerClient* function. An example of this is multiple robots being used to simultaneously build a single map.

Each client who want to use a service must first register with it using the *registerClient* function. This serves a number of purposes.

Firstly, if the client wants the service to perform callbacks on it can pass a C++ pointer to itself in the first parameter of the function. The other parameters provide information regarding the client's robot, such as the number of sonars, their positions on the robot (in a circular ring), and the robot's radius.

RegisterClient has a return type of **int**. If the registration is unsuccessful, for example if the maximum number of clients has already been reached, then the function returns –1. If the registration is successful the function returns a positive integer which is the clients' C*lient Number*. This number must be passed to the service each time a *push* function is called so that the service can know which client the information belongs to, and can use the correct robot profile as a consequence. This number must also be passed to the *unRegisterClient* function when unregistering the client from the service.

A client can also register with a service as a *listener*. This means that it cannot update the service's sonar readings, but will be notified of any changes through callback, just like a normal client would be. This can be done by passing a pointer to the client as the first parameter to *registerClient*, but setting the number of sonars to zero, as well as the *sonarPositions* pointer. If this is done, any call to *updateSonar* will fail, but a call to *updatePose* can still succeed.

If the service needs the clients information, such as sonar positions or robot radius, it can get it from the *clientProperties* array. This array is of type *SensorSet* which stores all necessary information regarding the client, including the pointer to the client. For

example, if a readings is popped from a queue, and it came from a client with a client number 5, then the robot radius for that client is stored in

```
clientProperties[5].robotRadius
```

See the *commonDefs.h* header file for the current list of parameters stored in the *SensorSet* object. At time of writing, August 2003, *SensorSet* is as follows:

```
template <class T>
struct SensorSet
{
    int numSonars;
    SosPose* sonars;
    long robotRadius;
    double sonarWidth;
    T* clientPointer;
};
```

The ServiceControl Information Repository

The *ServiceControl* class acts as an information repository. All information passed to the module must be send to this class, using the various *push…* functions (currently *pushSonar* and *pushPose*). The information is placed in a queue and can be accessed by the service inheriting from *ServiceControl* using the various *pop* functions (currently *popSonarReadingsQueue* and *popPoseQueue*). When a *pop* function is called, one item is removed from the head of the queue and placed in the object passed to the function, which is of type *SensorReadingSet*(defined in the *commonDefs.h* header file). The *SensorReadingSet* object stores everything that the service needs to know about that reading, the robot position, the sonar readings it received etc.

Because some service will be running with multiple threads, there could possibly be a conflict when pushing a reading onto a queue and popping a reading off in a different thread. Because of this, mutual exclusion is enforced on each queue by the *push* and *pop* functions, which removes the possibility of any conflict.

Each instance of a service, be it mapping, localisation or whatever, inherits from the *ServiceControl* class, and therefore there is one instance of a *ServiceControl* class for each service, with each service having its own queues. It is up to the central control module to decide what information is passed to which service. If the same information, e.g. robot position, is passed to two different services, two separate copies of it will exist, and if one service pops and processes the information, this will have no effect on the other service.

The *SensorReadingSet* object, as it is currently written, is prone to memory leaks. When the service pops the object off a queue, after it is finished with it the service must **delete** the *ranges* array if the object came from the sonar queue.

The ServiceControl Threading Service

ServiceControl handles the single or multi-threading of a service. If a service uses little processing power and can be expected to run in real time it can instruct the *ServiceControl* to run in single threaded mode. This is done in the constructor of the child class by passing the parameter **false** to the constructor of the *ServiceControl* class. For example, the *SpecularEstimator* class' constructor looks as follows:

```
SpecularEstimator::SpecularEstimator(): ServiceControl<int>(false)
{
    strcpy(systemName,"SpecularEstimator");
    maxClients = 1;
    sonarConfidences = 0;
}
```

As can be seen, when the *SpecularEstimator* constructor invokes the *ServiceControl* constructor, it passes the value **false** to it, indicating that the service is to be single threaded. **true** is the default value of the *ServiceControl* constructor, so if no value is passed to it, then it is assumed that the service is multi-threaded. The service's name can also be set here as well as the maximum number of clients allowed.

When a service is single threaded, each time new information is sent to it by the *push* functions, the function defined by the service whose purpose it is to use this information, *processQueue*, is called. If the service is multi-threaded (set by passing **true** to the *ServiceControl* constructor), then *processQueue* is spooled off in its own thread when the service is created, and is never called again after this. The thread is closed when the service is destroyed.

Because the *processQueue* function can be treated in two different ways, either single or multi-threaded, it is necessary to write it in two different ways.

The single-threaded mode is the simplest. Since the function will be called each time new information is received, all it must do is *pop* one piece of information off the queue, use it and terminate. An example of this is the *SpecularEstimator* class' *processQueue* function:

```
void SpecularEstimator::processQueue()
{
  SensorReadingSet reading;

  if(popSonarReadingsQueue(reading))
  {
      //if there is a reading on the queue
      //do whatever work needs to be done
  }

}
```

Here, an object of type *SensorReadingSet* is declared and passed to the *popSonarReadingsQueue* function. If the function returns **true** then a readings has been popped and placed in the *reading* variable. If not then the queue is empty.

The multi-threaded version of *processQueue* is slightly more complex. Since it is only invoked at creation time, it must run in an infinite loop, constantly checking the queue to see if there is anything new on it.

```
void A_Multi_Threaded_Service::processQueue()
{
    SensorReadingSet reading;

    while(1) //keep going in an infinite loop
    {
        while(popSonarReadingsQueue(reading))
        {
            //while there are still readings on the queue
            //pop them off and do whatever needs to be done with them
        }

        //if there are no readings, sleep for 100ms rather than take
        //up processor time constantly looping
        SosUtil::sleep(100);
    }
}
```

In the example above, when the function is executed it begins an infinite loop. Inside this loop it continually pops off readings form the queue and uses them in whatever way is appropriate for that service – map building, localisation or whatever. If there are no readings on the queue (i.e. when the *pop* function returns false) it is a good idea to make the thread sleep for a short time. This saves it from using processor time constantly looping, waiting for a reading to come in.

If the programmer does not want the service to sleep when there are no readings on the queue, and would rather simply keep checking for new readings, an alternate version of the *processQueue* function is:

```
void A_Multi_Threaded_Service::processQueue()
{
    SensorReadingSet reading;

    while(1) //keep going in an infinite loop
    {
        if(popSonarReadingsQueue(reading))
        {
            //if there is a reading on the queue
            //pop it off and do whatever needs to be done with it
        }
    }
}
```

ServiceControl's support for Interface-Based Callbacks

As mentioned earlier, *ServiceControl* supports client callbacks. This means that while the client of the service can update the service using the *push* functions, the service can also update the client by calling a specified function specific to that service. For example, mapping services update the client using the function *updateLocalCopy* to update the clients' local copy of a map, whereas localisation service (currently not implemented) can update the client using a function called *updateRobotPosition*.

Because of the way types and function calls on objects work in C++, this is implemented using *interfaces*. This means that if a service attempts to make a function call on an object, that object must have defined that function first. While this is not intended as a tutorial on object oriented (OO) interfaces, a short explanation of this area might be necessary.

An interface lists the public functions of a class. In C++ an interface is a class with no implementation, and one or more functions defined as **pure virtual**. Any class that inherits from that interface *must* implement those functions declared in the interface, or it will not be compiled. Because the object inherits from the interface (or '*implements the interface*' in OO terminology), it can be referred to using a pointer to that interface type, though by doing this only the functions listed in that interface can be called. If the object has other public functions that are not listed in the interface they cannot be referred to by using a pointer to the interface, a pointer to the object type itself is required.

Since different services will need to use different callback functions with different parameters, the *ServiceControl* class has to support calls any kind of interface, even interfaces that have not yet been implemented. It has to store pointers to the clients who have *implemented* these interfaces (from the *registerClient* function's 1st parameter). It is up to the service itself if and when a callback is executed.

To achieve this, the *ServiceControl* class is a **template** class. When a service inherits from *ServiceControl* it must specify the interface it makes callbacks to in its constructor. An example of this is the Mapping service's constructor:

```
SFEXPORT Mapping::Mapping():ServiceControl<MapUpdateInterface>(true)
{
```

Here the Mapping service tells *ServiceControl* that it makes callbacks to the *MapUpdateInterface* interface, and that it is multi-threaded (passing the **true** parameter). The *MapUpdateInterface* class looks like below:

```
class MapUpdateInterface
{
 public:
   virtual void updateLocalCopy(double value,long x,long y,string systemName)=0;
} ;
```

The word **virtual** before the function prototype and the **=0** after it specifies that the object is an abstract base class, and since there are no non-pure virtual functions in the class it is an interface.

If a service does not need to make callbacks it inherits from *ServiceControl* using the template **int** as was seen in the example of *SpecularEstimator* earlier.

Unfortunately, because of the problems C++ linkers currently have with templates (a long and boring subject I won't go into here), for each template version of *ServiceControl* required in the framework, it is necessary to declare this at the beginning of the ServiceControl.cpp file that defines the class. At time of writing there are two templates defined, which is done like below:

```
template class ServiceControl<MapUpdateInterface>;
template class ServiceControl<int>;
```

This creates two version of the *ServiceControl* object during compilation and linking. If, for example, a new interface called *LocalisationUpdateInterface* were to be created, and a localisation service was required to make callbacks to that interface, the two lines above would become:

```
template class ServiceControl<MapUpdateInterface>;
template class ServiceControl<int>;
template class ServiceControl<LocalisationUpdateInterface>;
```

For an example of a simple class inheriting from and using *ServiceControl* with callbacks and multi-threading look at the *RecordTestRun* class. For an example of a class working in single-threaded mode with no callbacks, look at the *SpecularEstimator* class. Below is the class definition of *ServiceControl* as of August 2003.

```
template <class T>
class ServiceControl
{
  public:
    virtual ~ServiceControl(); //class destructor

    //the client of this class is returned a client number which they use
    //each time they update the class
    virtual int registerClient(T* functionPointer, int noOfSonars,
                SosPose* sonarPositions, double beamWidth, long robotRadius);
    bool pushSonar(SosPose robotPose, long* sonarRanges, int clientNumber,
                   double anythingElse = 1);
    bool pushPose(SosPose robotPose, int clientNumber);
    bool unRegisterClient(int clientNumber);

  protected:
    int maxClients;
    vector<SensorSet<T> > clientProperties; //stores all info about each clients

    //if the child of this class wants to operate in a single threaded
    //way, so that whenever updateSonar() is called processQueue() is also called,
    //they can simply set isMultiThreaded to 'false' in their constructor, and
    //processQueue() will be called at the end of updateSonar() the default is
    //that the child will operate in a multi-threaded fashion - asynchronously.
    const bool isMultiThreaded;

    bool clientNumbers[MAX_CLIENTS]; //this array stores which client numbers
                                     //are already taken
    int numOfClients;  //use this to remember how many clients we have ->incremented
                       //in registerClient, decremented in unRegisterClient()
    char systemName[50];

    SosThread* processReadingsQueue; //a pointer to the processQueue thread in
                                     //a multi-threaded service
    SosFunctorC<ServiceControl> *queueProcessor;//a pointer to processQueue function

    //the constructor is Protected so that only a child class can call it, it
    //can never be used to create an instance of the ServiceControl class on its own
    ServiceControl(bool childIsMultiThreaded = true);//class constructor

    virtual void processQueue() = 0;

    SosPose transformSonar(SosPose robotPose,int sonarNum, int clientNumber);

    //pop the top reading off the queue of sonar range readings
    bool popSonarReadingsQueue(SensorReadingSet&);

    //pop the top reading off the queue of robot poses
    bool popPoseQueue(SensorReadingSet&);

 private:
    queue<SensorReadingSet> sonarReadingsQueue;//queues up the sonar readings
    queue<SensorReadingSet> poseQueue; //queues up the robot's poses

    SosMutex *sonarQueueMutex; //mutually exclusive locks used on the queues to
    SosMutex *poseQueueMutex;  //preventdifferent threads clashing when reading
                               //or writing to the sonar queue
};
```

Extending the *ServiceControl* class

It is of course possible to extend the *ServiceControl* class (though matching the original authors, how shall we put it….. brilliance will take a bit more effort ☺). Below is the procedure for adding a new queue to the *ServiceControl* class. Other means of extending the class can not be anticipated, and are therefore not explained. If any more information is required on the architecture, either about extending it or just general questions, email shane.osullivan@don't-even-think-about-it.com

To add a new queue to the object, a number of things must be done.

1. Declare the queue in the class header file, as with *sonarReadingQueue* and *poseQueue*.
2. Define an object of type *SosMutex* object for the queue (like *sonarQueueMutex*), which will be used by the *push* and *pop* functions to avoid errors reading and writing from and to the queue with multiple threads.
3. Define a *push* function for the queue. This function must accept a client number, like the current two *push* functions do. It must also validate the client's client number. All client numbers are ≥ 0, and less than the variable *maxClients* (default value is 50 if not set by the service). This can be checked as follows:

```
if(clientNumber < 0 || clientNumber > maxClients)
{
    return false;
}
```

   If a client exists with the number provided the function, then the entry with that number in the *clientNumbers* Boolean array will be set to **true**. This is done as follows:

```
if(clientNumbers[clientNumber] == false)
{
    return false;
}
```

   Now a object of type *SensorReadingSet* must be declared, and all the necessary information copied into it – for example in *pushSonar* the robot's position, orientation and sonar ranges are copied into it.

   Before pushing the reading onto the queue, it must first be checked if the service is running in multi-threaded mode, and if it is it is necessary to lock the mutual exclusion variable for that queue, as follows:

```
if(isMultiThreaded)
{
    poseQueueMutex->lock();
}
```

Next, push the *SensorReadingSet* object onto the queue, and unlock the mutex variable.

Finally, if the class that has inherited from *ServiceControl* is operating in single-threaded mode the *processQueue* function must be called. This is done as follows:

```
if(!isMultiThreaded)
{
    processQueue();
}
else
{
    SosThread::yieldProcessor();
}
```

The *yieldProcessor* function call is optional. It takes control of the CPU from this thread and allows another thread to take over. This speeds up processing a little on services that are heavy on CPU usage since the processing thread doesn't have to wait as long to gain control of the CPU.

4. Define a *pop* function for the queue. This function should look similar to the two already defined, return a Boolean true if it succeeded and false if the queue is empty. It should take one parameter as an argument, an object of type *SensorReadingSet* passed by reference. It should also check if the service is operating in multi-threaded mode, and if it is, place a mutex lock and unlock around the popping of the queue, as shown below:

```
template <class T>
bool ServiceControl<T>::popSonarReadingsQueue(SensorReadingSet& reading)
{
    if(sonarReadingsQueue.empty())
    {
        return false;
    }

    if(isMultiThreaded)
    {
        sonarQueueMutex->lock();

        reading = sonarReadingsQueue.front();
        sonarReadingsQueue.pop();

        sonarQueueMutex->unlock();
    }
    else
    {
        reading = sonarReadingsQueue.front();
        sonarReadingsQueue.pop();
    }

    return true;

}
```

Services becoming clients of other services

Client-server communication is generally quite simple in this architecture – the client creates the server, sends it information, receives callbacks or polls the service for information, and finally deletes it. However, when a child of the *ServiceControl* class is a client of another child of the *ServiceControl* object, a slight change must be made. An example of this can be seen in the *ME85mod, ME88mod* and *K97mod* mapping classed. Each of these is a client of the *SpecularEstimator* service. Every time a new client registers with *ME88mod*, it must also register this client with the *SpecularEstimator* service. (As a side note, *ME88mod* both creates and deletes the instance of *SpecularEstimator*, and so can be said to 'fully own' it. It is of course possible for the second service to exist outside the client service, with perhaps a pointer to the second service being passed to the client service in the constructor, though this has not been implemented yet.)

ME88mod ensures that each new client is registered with a *SpecularEstimator* class by redefining the *registerClient* function from *ServiceControl*, as follows:

```
//This function is a redefinition of the parent class' registerClient function, so it can
// register itself with another service every time a new client registers with it
int ME88mod::registerClient(MapUpdateInterface* clientPointer, int noOfSonars,
                            SosPose* sonarPositions, double beamWidth, long robotRadius)
{
  int clientNumber = -1;
  clientNumber = Mapping::registerClient(clientPointer,noOfSonars,sonarPositions,beamWidth,robotRadius);
  if(clientNumber != -1)
  {
    specEst[clientNumber] = new SpecularEstimator;
    specEst[clientNumber]->registerClient(0, clientProperties[clientNumber].numSonars,
                      clientProperties[clientNumber].sonars,
                      clientProperties[clientNumber].sonarWidth,
                      clientProperties[clientNumber].robotRadius);
  }
  return clientNumber;
}
```

Since *registerClient* is defined as **virtual** in *ServiceControl*, when it is called by the client, the *registerClient* function from *ME88mod* is executed, and not the *registerClient* from *ServiceControl*. In the function above we can see that *ME88mod* calls the *registerClient* function from its parent class to take of all the usual things the function does. It then creates a new service of type *SpecularEstimator*, stores a reference to it in an array of pointers to the *SpecularEstimator* type, and registers its new client with that service. All of the objects of type *SpecularEstimator* are destroyed either when the client unregisters or when in the class destructor. As mentioned above, a class does not necessarily have to create the other service, as is done here. It could also be given a reference to the other service it wishes to use, rather than creating and deleting it.

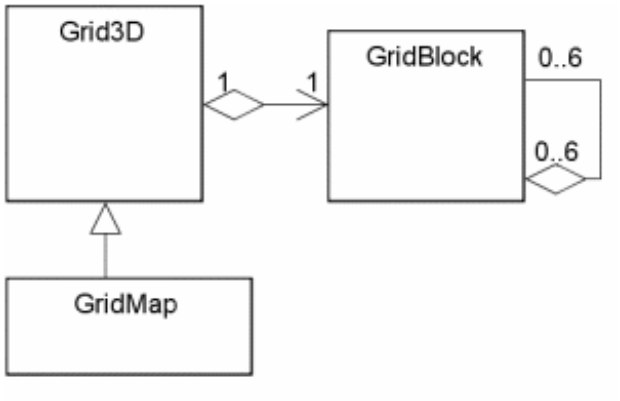
In the example above, the *ME88mod* service is a client of the *SpecularEstimator* service, which does not perform callbacks. If a service was to be the client of a second service which *did* implement callbacks, it would of course have to implement the interface that lists the callback function(s) for that service, just like any other client.

A second way that one service could become the client of another service, rather than it making itself a client, would be for a third party to register it with another service. For example, if the central control class created service A and service B, it could register service A as a client of service B by calling B.*registerClient* and passing a pointer to service A (as long as service A implements service B's callback interface). While service A would not make any calls on service B, it would receive all callbacks that service B sends out.

An example of the above situation would be for the central control class to create a mapping service and a localisation service, and to register the localisation service as a client of the mapping service. While the localisation service would initially have to work off a small map that's not very well defined (the robot is in a new, previously unexplored environment), as the mapping service created a better map it would send updates to the localisation service, which would then be able to make more accurate estimations as to the robot's position because it had more information regarding the environment.

Map Storage Classes

There are three classes used to store grids and maps. These are G*ridBlock, Grid3D and GridMap*.



The *GridBlock* class is a fixed size grid that stores the values. *Grid3D* arranges many *GridBlock* objects to form a dynamically sized map, and *GridMap*, a child of the *Grid3D* class, provides a number of additional operations to be performed upon a map.

GridBlock Class

The *GridBlock* class is very simple. It is essentially a 2D or 3D array which, once initialised, is of fixed size. The object can be initialised with the parameters:

- *blockSize* – the number of cells wide and high the block will be. All *GridBlock*s are square.
- *defaultVal* – the value to initialise all cells to.
- *blockHeight* – How tall the block is. If this is set to 1 then the block is essentially a 2D array. If it is any number greater than one then it is a 3D array.

```
template <class T>
class GridBlock
{
  public:
    GridBlock();
    GridBlock(long blocksize, T defaultval,long blockheight=1);
    virtual ~GridBlock();
    bool putVal(T value, long x,long y, long z = 0);
    T getVal(long x, long y, long z=0);

    GridBlock *north, *south, *east, *west, *above, *below;
    long globOrigin[2];

  private:
    const long blockSize;
    const long blockHeight;
    const T defaultVal;

    T*** values;
};
```

However, simply having a map of fixed size is not enough. It is necessary for a map to be able to grow as big as needed, but also to only be as big as needed and no more. For this reason, multiple *GridBlock* objects can be organised into a quad-linked list by another class, the *Grid3D* class. The *GridBlock* class contains six pointers, *north, south, east, west, above* and *below*. The *Grid3D* class will use these to link each *GridBlock* object to other *GridBlock* objects in order to form a complete, dynamic map that can vary in size.

The *globOrigin* parameter stores the block's position in the overall map with the X position in the 1st element of the array and the Y position in the 2nd element of the array.

Grid3D class

The *Grid3D* class arranges one or more *GridBlock* objects in a lattice to create a dynamically sized map that grows as the user needs it. It can be initialised with the parameters:

- *blockSize* – The number of cells big a single *GridBlock* is. The bigger this number the faster the search for a value will be, but the more wasteful of memory it is.
- *radius* – the initial size (in *GridBlock*s) of the map.
- *unknown* – the default value of the map. If the value of a cell has not been set and the user asks for its value, then this *unknown* value is set.
- *blockHeight* – The height of the map. If this is set to 1 then it is a 2D grid.

```
template <class T>
class Grid3D
{
public:
    Grid3D();
    Grid3D(int blocksize, int radius, T Unknown, int blockheight = 1);
    virtual ~Grid3D();

    T getGridRef(long x, long y, long z = 0);
    bool updateGridRef( T value, long x, long y, long z = 0);

    //copy directly another map
    void copy(Grid3D<T>* mapToCopy);
    //save the map to a file
    bool save(char* filename);
    //load a map from a file
    bool load(char* filename);
    //return the size of the map in either NORTH, SOUTH, EAST or WEST
    long getDimensions(int direction);

    //return the size of the map in either NORTH, SOUTH, EAST or WEST
    //long getDimensions(int direction);
    long getUpdatedDimensions(int direction);
    const T getUnknown() const{return unknown;};
    void reset();

protected:
    void appendBlock(GridBlock<T>* original_block,GridBlock<T>* new_block,int direction);
    int growMap(int times, int direction=0);
    GridBlock<T> * newBlock();
    GridBlock<T>* findBlock(long x, long y);

    GridBlock<T>* myMap;
    int errorVal;
    long blockSize;
    long blockHeight;
    long dimensions[6];
    long updatedDimensions[4];
    T unknown;
};
```

To set the value of a cell in the map, use the *updateGridRef* function. The first parameter is the value to set, and the last three values are the (X,Y,Z) grid position of the cell. If the map is not big enough at the current time, so that this grid reference does not exist in memory, then the map is grown in the direction of the cell until it is big enough. For example, if the required cell is to the north of the map, a new row of *GridBlock*s is added to the northern end of the map until it a *GridBlock* that contains the required cell.

To retrieve the value of a cell, use the *getGridRef* function, once again with the (X,Y,Z) position of the cell. If the value of the cell has not been set, then the value that *unknown* was set to in the constructor (0 is the default value) is returned. If the cell does not exist in memory (i.e. if it is outside the map) then *unknown* is also returned.

The map can be saved and loaded with the *save* and *load* functions.

To find the size of the map, including the complete area covered by the *GridBlock*s in memory, whether they have been updated by the user or not, use the *getDimensions* function. This takes as its parameter one of six #defined variables, NORTH, SOUTH, EAST, WEST, ABOVE, and BELOW. When NORTH is passed to the function, it gives the Y position of the cell that is furthest north. The same goes for the others.

To find the size of the map that has actually been updated by the user (which is often smaller than the total cells actually held in memory), use the function *getUpdatedDimensions*. This takes one of the four constants NORTH, SOUTH, EAST or WEST as parameters.

GridMap class

The *GridMap* class is a child of the *Grid3D* class, and can therefore do everything that the *Grid3D* class can do, except for the fact it can only be a 2D map, and not a 3D map. It also has a number of additional abilities.

```
template <class T>
class GridMap : public Grid3D<T>
{
public:
    GridMap();
    GridMap(int blocksize, int radius, T Unknown);
    GridMap(GridBlock<T>*mapToCopy, long* dimensions, long blocksize, T defaultValue);
    ~GridMap();

    bool updateGridRef(T value, long x, long y){return Grid3D<T>::updateGridRef(value, x,y,0);}
    T getGridRef(long x, long y){return Grid3D<T>::getGridRef(x,y,0);}
    void copy(Grid3D<T>* mapToCopy,int reduceFactor = 1,int valueToSelect = LARGEST_VALUE);
    void reduceDimension(int reduceFactor = 4, int valueToSelect = LARGEST_VALUE);
    void boxBlur(int kernelSize=3, double boxVal=1);
    void gaussBlur(int kernelSize=3);
    bool importPointMap(char* fileName, T value=1, long squareSize=100);
    bool addLine(long x1, long y1, long x2, long y2, T value, long squareSize);
    double correlateMap(GridMap<T>* mapToCompare);
    double scoreMap(GridMap<T>* mapToCompare);
    bool growOccArea(long radius, T lowerBound, T upperBound, long squaresize = 100);
};
```

- It can shrink the map using the *reduceDimension* function. It also has an enhanced version of the *copy* function from the *Grid3D* class that can be used to copy a shrunk version of the map being copied rather than simply copying it exactly. The degree to which it is to be reduced is set using the *reduceFactor* variable. A value of 1 (the default) means that no reduction is performed. The only values supported are 1, 4 and 9.
- Blurring – the map can be blurred either in a simple, box-blur fashion using the *boxBlur* function, or in a gaussian-blur way using the *gaussBlur* function. In both of these, the higher the *kernelSize* variable the more the maps are blurred. In *boxBlur*, the overall 'brightness' of the map can be increased using the *boxVal* variable. The higher it is, the more the value of the cells will be raised. The default value of 1 means that the map is just blurred, with no brightness alteration being performed.
- Importing/Converting Saphira's .wld files - If the *GridMap* class is being used on Linux, it can import the world files in use by Saphira, and convert the line-based maps into grid cells using the *importPointMap* function. The *value* variable is the value to set the cells that a line passes through. The *squareSize* variable tells the function what the scale of the map is. For example, if it is set to 100 (the usual value), it means that each represents a square of 100mm * 100mm to a side in the real world.

- Adding lines to the map – the *addLine* function can be used to add a line to the map. You must give it the start (x,y) position and the end (x,y) position of the line. Note that these values are in mm, and the real position of the lines in the environment, rather than cell coordinates. For this reason, the *squareSize* parameter must also be set, as in the *importPointMap* function above.
- Correlation – One *GridMap* can be compared to another to get a correlation value between them using the *correlate* function.
- Score Map – One *GridMap* can be compared to another to get a score, which is the squared difference between them by using the *scoreMap* function.
- Configuration Space Generation – a map can be converted to its 'configuration space' equivalent using the *growOccAreas* function. This essentially expands each obstacle by half the width of the robot so that it shows places the robot can and cannot go, rather than just the map. The *radius* of the robot must be provided to it in millimetres, as well as the *lowerBound* and *upperBound* of what is considered to be an occupied area. For example, we may only want to expand cells with values between 0.8 and 1.0. Finally, it must be supplied with the *squareSize* of each cell, with the default being 100mm to a side.

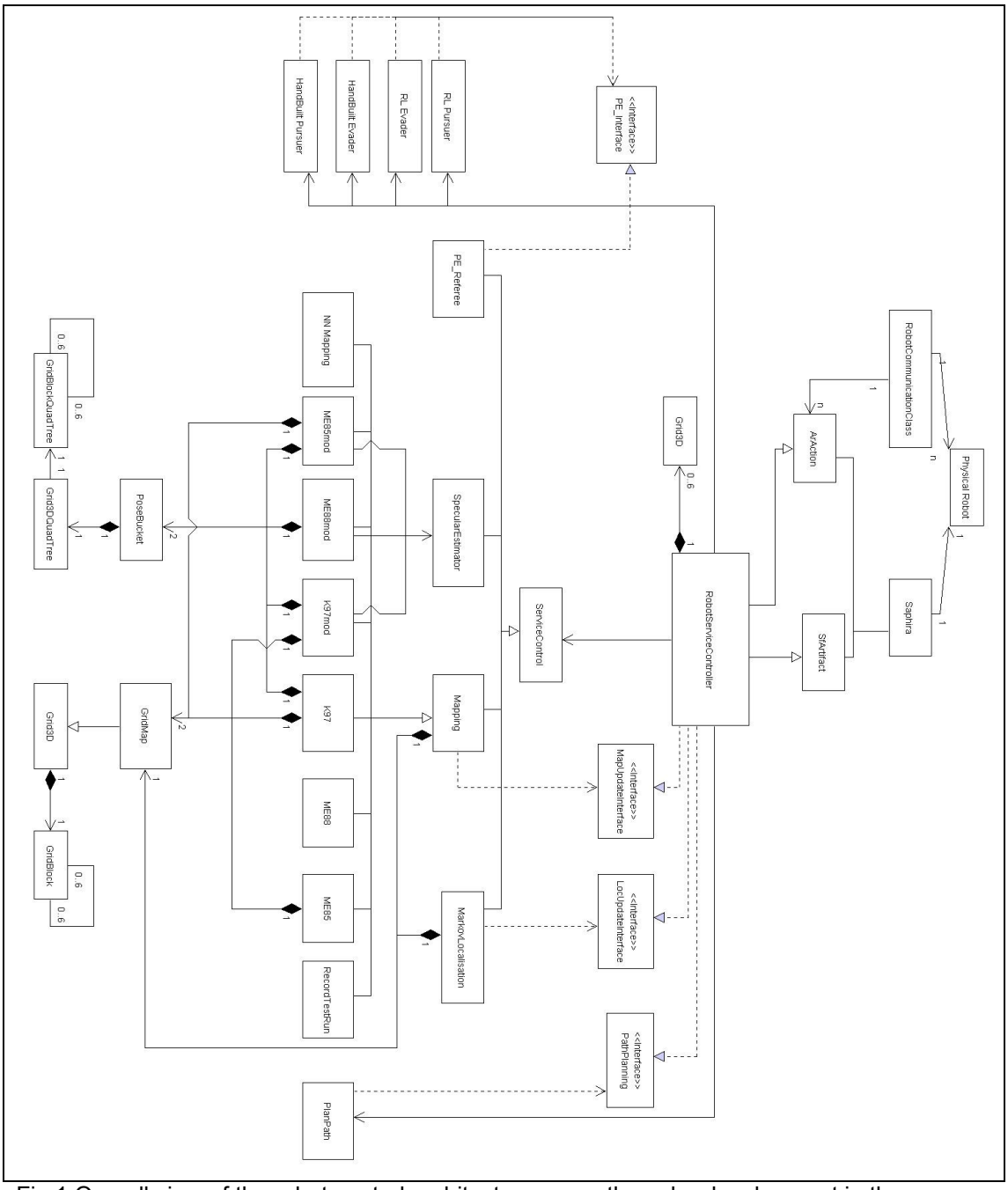


Fig 1 Overall view of the robot control architecture currently under development in the University of Limerick robotics group.