Choosing Where To Go: Mobile Robot Exploration

Robbie Shade

New College

Supervisor:

Professor Paul Newman

Robotics Research Group

Department of Engineering Science

University of Oxford

August 2011

Robbie Shade
New College

Doctor of Philosophy
August 2011

Choosing Where To Go: Mobile Robot Exploration

Abstract

For a mobile robot to engage in exploration of a-priori unknown environments it must be able to identify locations which will yield new information when visited. This thesis presents two novel algorithms which attempt to answer the question of choosing where a robot should go next in a partially explored workspace.

To begin we describe the process of acquiring highly accurate dense 3D data from a stereo camera. This approach combines techniques from a number of existing implementations and is demonstrated to be more accurate than a range of commercial offerings. Combined with state of the art visual odometry based pose estimation we can use these point clouds to drive exploration.

The first exploration algorithm we present is an attempt to represent the three dimensional world as a continuous two dimensional surface. The surface is maintained as a planar graph structure in which vertices correspond to points in space as seen by the stereo camera. Edges connect vertices which have been seen as adjacent pixels in a stereo image pair, and have a weight equal to the Euclidean distance between the end points. Discontinuities in the input stereo data manifest as areas of the graph with high average edge weight, and by moving the camera to view such areas and merging the new scan with the existing graph, we push back the boundary of the explored workspace.

Motivated by scaling and precision problems with the graph-based method, we present a second exploration algorithm based on continuum methods. We show that by solving Laplace's equation over the freespace of the partially explored environment, we can guide exploration by following streamlines in the resulting vector field. Choosing appropriate boundary conditions ensures that these streamlines run parallel to obstacles and are guaranteed to lead to a frontier – a boundary between explored and unexplored space. Results are shown which demonstrate this method fully exploring three dimensional environments and outperforming oft-used information gain based approaches. We show how analysis of the potential field solution can be used to identify volumes of the workspace which have been fully explored, thus reducing future computation.

Statement of Authorship

This thesis is submitted to the Department of Engineering Science, University of Oxford, in fulfilment of the requirements for the degree of Doctor of Philosophy. This thesis is entirely my own work, and except where otherwise stated, describes my own research.

Robbie Shade, New College

Funding

The work described in this thesis was funded partly by OC Robotics, and partly by the Engineering and Physical Sciences Research Council (EPSRC).

Acknowledgements

First and foremost I'd like to thank my supervisor, Paul Newman, for his unwavering support, invaluable research insights and ideas, and his good humour in times of crisis. It has been hugely rewarding being a member of his Mobile Robotics Group, not least because of the fantastic team he has put together: thanks everyone for the discussions, arguments, and adventures we've had together!

My girlfriend Kai deserves special thanks: I'm not sure I'd have made it this far without you, and you've made my time here many times more enjoyable. Finally I'd like to thank my family who have been there to support me from the beginning, always welcoming me when I needed to escape from Oxford, and for the much needed biscuit supply drops.

Contents

List of Figures

0.1 Nomenclature

$\mathbb{R}^3$ - Three dimensional Euclidean space

$\mathbf{c}_i = [x, y, z, \theta_r, \theta_p, \theta_q]^T$ - A camera pose at time i with size degrees of freedom

$\mathcal{I}^L, \mathcal{I}^R$ - Left and right images from a stereo camera

$\mathbf{p} = [p_x, p_y, p_z]^T$ - A 3D point lying in $\mathbb{R}^3$

$G = (V, E)$ - a graph data structure

$u \in V$ - a vertex in a graph

$(u, v) \in E$ - an edge in a graph with vertices u and v

$w : E \to \mathbb{R}$ - weighting function mapping a real-valued weight to each edge $e \in E$

ϕ - Scalar field which satisfies Laplace's equation

Ω - Domain over which a partial differential equation is solved

$\partial\Omega$ - Boundary of a partial differential equation's domain

Chapter 1

Introduction

1.1 Choosing Where To Go

This thesis is about autonomous exploration of a-priori unknown environments with mobile robots – it is about choosing where to go.

After switching a mobile robot on, where should it go? In this work we say it should try and look at, and thus map, every reachable surface. Initially only a fraction of the workspace is visible and plans must be made about where the robot should move to further increase the extent of its map. Ideally we would decide upon a smooth path to a new sensor pose which looks into a "maximally promising" area. But how do we define promising? It is evidently sensible to try and extend the boundary between explored and unexplored and in particular, to plan a view which is in some sense perpendicular to the boundary, to look directly into the unseen regions of the workspace. We cannot expect a single boundary to exist and so we must entertain the possibility that there could be a multitude of view points which would provide information rich views – which one to choose? Some could be far away and so costly to reach but resting on an extensive free space boundary, some could be close but small. How should we balance these aspects and capture our intuition of a good exploration strategy given above?

Figure 1.1: OC Robotics snake-arm robot. A highly redundant snake-arm robot which can access enclosed or confined spaces by adjusting the joint angles along its length.

1.2 Exploration in 3D

We are specifically interested in exploration techniques which are not limited to robots constrained to the ground plane. The snake-arm robot shown in Fig. 1.1, designed and built at OC Robotics [92] is an example of a highly flexible robot which operates in three dimensions. These snake-arms are suited to a wide range of tasks – from applying sealant to Airbus aircraft interiors [43] to repairing cooling systems in nuclear power plants [91].

Currently snake-arm control is performed by a human operator – using a joystick to drive the tip of the arm, with the control software calculating the necessary joint angles. This form of control is acceptable in some situations such as repetitive bolt inspection, but consider the daunting task of searching for explosives in the interior of every car in a car-park, or searching for survivors in the ruined buildings of a city struck by an earthquake.

If an autonomous search could construct a 3D representation of the workspace as it progressed, then this would provide a valuable resource for later review. The most up to date 3D model of a nuclear reactor's piping could be compared with a model from a year ago, and potentially worrying discrepancies could be flagged up.

With recent developments in personal robotics [69] in which robots are demonstrated performing complex manipulations of objects such as making a cup of tea or playing pool, and the impressive acrobatics being displayed by aerial quadcopters [11], it seems that that the problems of exploration and path planning in 3D are well worth considering.

The aim of this thesis is to develop and test exploration algorithms which:

- explore in three dimensions
- require no a-priori knowledge of the environment
- operate with no infrastructure which might provide pose estimation (such as fixed camera networks, radio beacons, or pre-placed markers in the environment)
- be scalable to larger environments

1.3 Thesis Structure

1.3.1 How to read this thesis

This thesis is written to tell a coherent story when read as a whole. However, Chapter 3 could be skipped by a reader who is familiar with dense stereo processing. Additionally Chapter 4, which describes the robot systems used, is not essential for understanding the exploration algorithms in the later Chapters. The bulk of the novel work is found in Chapter 5 and Chapter 6.

1.3.2 Structure

We begin, in Chapter 2, with an overview of research done in the fields of pose estimation, mapping, and exploration.

Chapter 3 considers various 3D sensing modalities, and justifies our selection of a high resolution stereo camera. The pipeline and algorithms used to produce accurate and dense 3D point clouds from a pair of stereo images are described in detail. We present results which show our stereo implementation outperforming commercial offerings, and show quantitatively that it performs comparably with the best local window-based methods.

A robot is a complex, layered system with many interlinked hardware and software components. Chapter 4 describes the structure of the Europa robot which is used by our research lab, the Mobile Robotics Group (MRG). We present the software hierarchy from low-level device drivers to high-level exploratory algorithms and user interfaces. The inter-communication between software components is handled by the Mission Oriented Operating System (MOOS) middleware, and we discuss the protocols employed by this system.

We now consider the central question of choosing where to go. Chapter 5 introduces a graph based exploration algorithm in which we answer this question by

identifying 'rifts' in the graph and actuating the camera to view and resolve them. A rift is an area corresponding to a large discontinuity in the disparity data from the stereo camera, and this is manifested in the graph by an area of high local edge weight. Ultimately it is found that this approach places unrealistically high demands on the accuracy of pose estimates and depth data, and relies on a number of arbitrary parameters. This motivates the need for a parameter-free exploration algorithm.

We present in Chapter 6 a parameter-free algorithm based on gradient-descent in a continuous vector field found by solving Laplace's equation over the freespace of the map. It is shown that by setting appropriate boundary conditions and following streamlines in the resulting potential flow field that we are guaranteed to reach a frontier – a boundary between explored and unexplored space. Results are shown which compare our approach to existing algorithms, and we conclude the chapter by demonstrating a method for identifying fully explored volumes which reduces the computational cost of solving Laplace's equation.

Concluding remarks and an outline of future work are presented in Chapter 7.

1.4 Contributions

The main contributions of this thesis are as follows:

- an accurate implementation of an adaptive window stereo matching algorithm which runs in realtime and outperforms existing commercial solutions
- an exploration algorithm based on identifying characteristics of a graph which is constructed from stereo depth data.
- a parameter-free exploration algorithm based on continuum mechanics
- implementation of a 3D environment simulator which interfaces with the Mission Oriented Operating System (MOOS) [73], and which provides depth data from a simulated depth sensor.

1.5 Publications

The description of the dense stereo algorithm given in Chapter 3 was published as part of the "Navigating, Recognizing and Describing Urban Spaces With Vision and Lasers" paper in the International Journal of Robotics Research in 2009 [86].

The graph based exploration algorithm which is described in Chapter 5 was presented as "Choosing Where To Go: Complete Exploration With Stereo" at the IEEE International Conference on Robotics and Automation (ICRA) in Anchorage, Alaska, in May 2010 [109].

The workspace exploration with continuum methods which is described in Chapter 6 was partly described in "Discovering and Mapping Complete Surfaces With Stereo" at the IEEE International Conference on Robotics and Automation (ICRA) in Shanghai, China, in May 2011 [110].

Chapter 2

Background

This chapter presents an overview of research areas which are either prerequisites for the exploration behaviour presented in later chapters, or which present alternative answers to the question we are asking: where should the robot move to next?

2.1 Overview

What prerequisites are there for a robot to perform exploration? In the problem as defined in Chapter 1 a robot will, upon startup, have no knowledge of the environment in which it finds itself. It is ignorant and blind, and to remedy this situation it must begin to interact with the environment in such a way that it can begin to build an internal representation of the physical structure of the world.

A simple bump sensor as found on the Roomba vacuum cleaning robot [48] can detect collisions, but driving into every obstacle to build an environment is not a particularly efficient strategy. The modern robot is spoilt for choice in terms of sensing capabilities and a few of these such as stereo cameras and laser scanners are discussed later in Chapter 3.

The output from these sensors is stored in the robot's internal world map, and

the robot can now begin to move through the environment acquiring new sensor data as it does so. We are now faced with the two problems of figuring out how the robot has moved with respect to its map, and of extending and expanding the map based on new sensor input. Collectively these problems are known as the Simultaneous Localisation and Mapping (SLAM) problem and have been studied extensively in the robotics community [6, 27, 84, 118].

This chapter will cover the following distinct areas:

6DoF pose estimation – what is the sensor's position with respect to a global reference frame, and how is this estimate updated as the robot moves? The SLAM problem.

3D data acquisition – what choices do we have for capturing and processing 3D data? This is covered in more detail in Chapter 3.

Mapping – given a 6DoF pose and 3D data captured at that pose, how do we integrate it into a globally consistent map structure in which we can plan paths and perform exploration?

Exploration – high level decision making: given a map and a current pose, where should we move next?

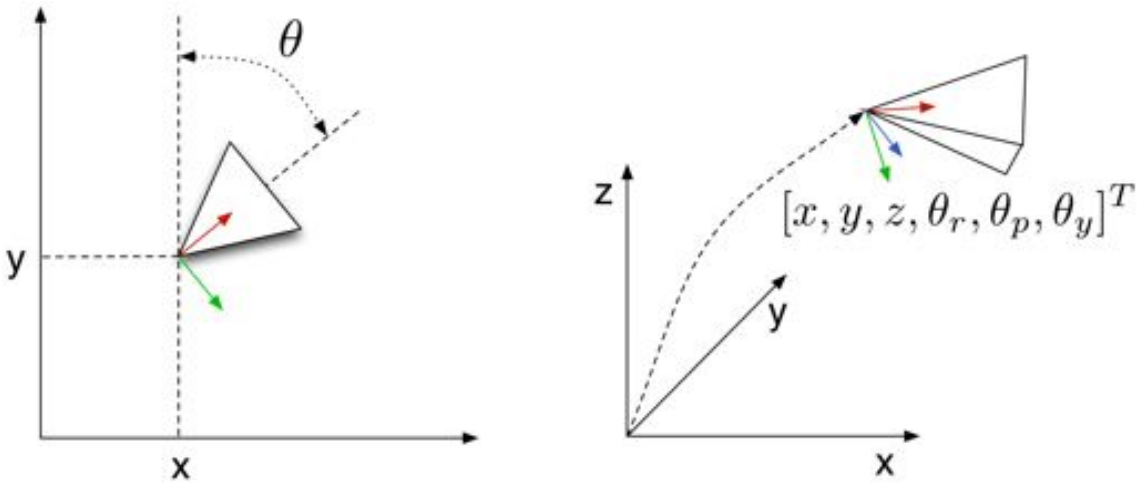


Figure 2.1: Coordinate frames in 2D and 3D. (a) shows the 2D coordinate frame of a robot confined to the ground plane. Its pose is fully described by two translational components and one rotational component. (b) shows a sensor in a 3D coordinate frame: its pose now consists of translations along and rotations around 3 axes.

2.2 Pose estimation

Assuming the robot has made at least an initial scan of its immediate environment, the first question we want to answer is 'where is the robot located with respect to its initial pose?'.

It must maintain some internal estimate of its position and orientation in the world and for a robot confined to the ground plane (such as a typical wheeled robot), this pose estimate will have two translational components and one rotational component: $[x, y, \theta]^T$. We will be considering the more general case of a robot which has freedom in 3 spatial dimensions, and will therefore maintain an estimate which contains 3 translational components, and 3 rotational components:

$$\mathbf{p} = [x, y, z, \theta_r, \theta_p, \theta_y]^T \tag{2.1}$$

where θ_r is the *roll* angle around the x-axis, θ_p is the *pitch* angle around the y-axis, and θ_y is the *yaw* angle around the z-axis. These six degrees of freedom will be referred to as 6DoF from here on. Figure 2.1 shows coordinate frames for robots

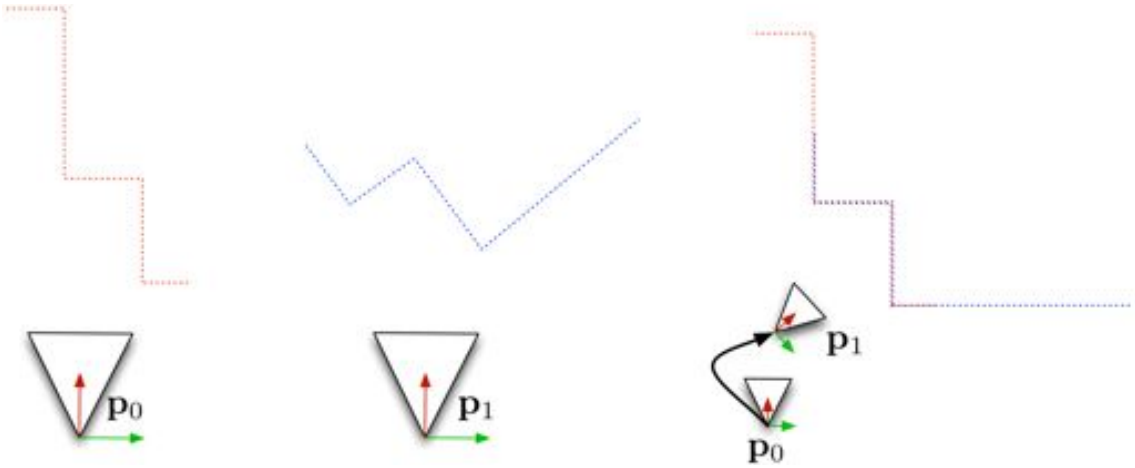


Figure 2.2: Laser scan matching. Two laser scans (shown in red and blue), taken from two different poses $\mathbf{p}_0$ and $\mathbf{p}_1$. The transformation found by minimising the distance between neighbouring points using ICP scan matching is the sensor transformation from $\mathbf{p}_0 \rightarrow \mathbf{p}_1$.

operating in 2D and 3D.

2.2.1 Wheel odometry

A simple way to estimate pose is to record the number of rotations of each wheel of a robot. With knowledge of the wheel diameters these values can be converted to translation and rotation values. This is easy to implement but errors in position quickly accumulate due to wheel slippage (the odometry is then not representative of distance travelled), and due to the open-loop nature there is no way to correct these errors. For this reason basic wheel odometry is rarely used as the only pose estimator in a robot system (it may well be used to provide initial transformation estimates for more sophisticated algorithms however). It is also limited to wheeled robots: legged or aerial robots must find another way to estimate pose.

2.2.2 Laser scan matching

Consider a robot which takes two laser scans $\mathbf{z}_0$ and $\mathbf{z}_1$ at two different poses $\mathbf{p}_0$ and $\mathbf{p}_1$. $\mathbf{z}_0$ and $\mathbf{z}_1$ are simply 1D vectors of range values obtained by measuring

time-of-flight of a laser beam. If there is overlap between the two scans then we can find a transformation which best aligns them and deduce the change in pose which caused them. This is shown in Fig. 2.2.

This idea of scan matching for localisation is due to [65], which uses the Iterative Closest Point (ICP) algorithm of [9]. ICP takes two scans $\mathbf{z}_0$ and $\mathbf{z}_1$, and an initial estimate of the transformation (typically this will come from the unreliable wheel odometry), $\mathbf{p}_0 \rightarrow \mathbf{p}_1$. Each point in $\mathbf{z}_0$ is associated with its nearest spatial neighbour in $\mathbf{z}_1$ and the estimated transformation is iteratively modified such that the sum of neighbour to neighbour distances is reduced.

The result is a transformation (translation and rotation) which results in the two scans being aligned – and this transformation when applied to $\mathbf{p}_0$ will yield the best estimate of $\mathbf{p}_1$.

2.2.3 Visual odometry

Cameras are a pervasive sensor in robotics – they are passive sensors which are cheaper and more lightweight than a typical laser scanner, provide RGB images of the world, and if two or more cameras are viewing a scene then 3D reconstruction can be performed (more detail can be found later in Chapter 3).

If we could track distinctive features through a sequence of images then we could estimate the inter-frame pose transformations (rotation and translation) which best explain the image sequence. This is known as visual odometry (VO) and it is used on a wide-range of robots from Mars rovers [20, 66] to aerial platforms [87].

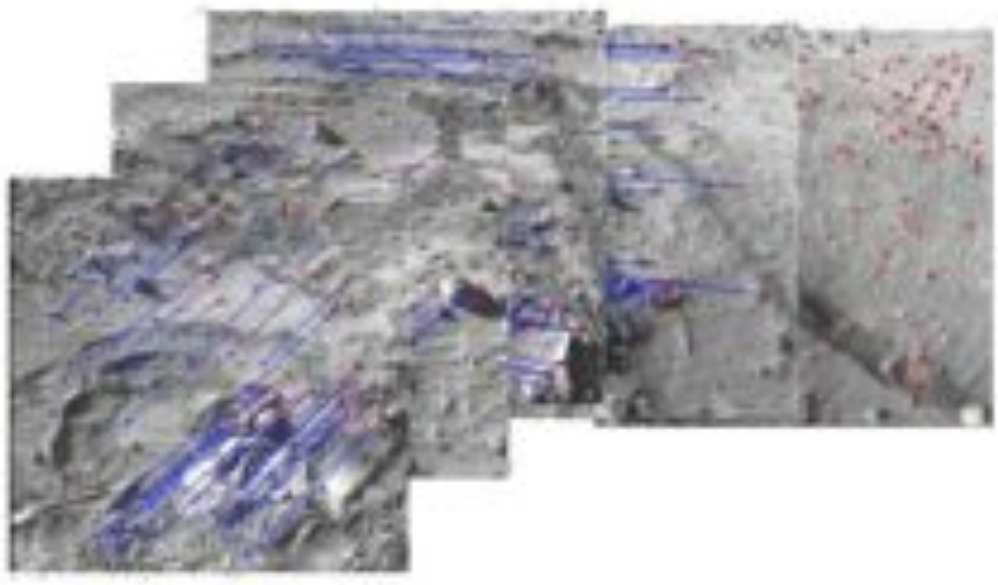
A recent development is the relative bundle adjustment system of [86, 112, 113]. Rather than attempt to calculate pose estimates in a global Euclidean frame, this system keeps track of only inter-pose transformations, as shown in Fig. 2.3(b). The result is that local relative transformations are extremely accurate as errors don't accumulate, but the resulting pose graph is not necessarily globally metrically

accurate. For example the inter-pose transformations will be continuous as the sensor travels into a lift, travels up a floor, and exits, but if we visualise the transformations in $\mathbb{R}^3$ we will see no change in pose height as the input images from the camera give no indication of height change.

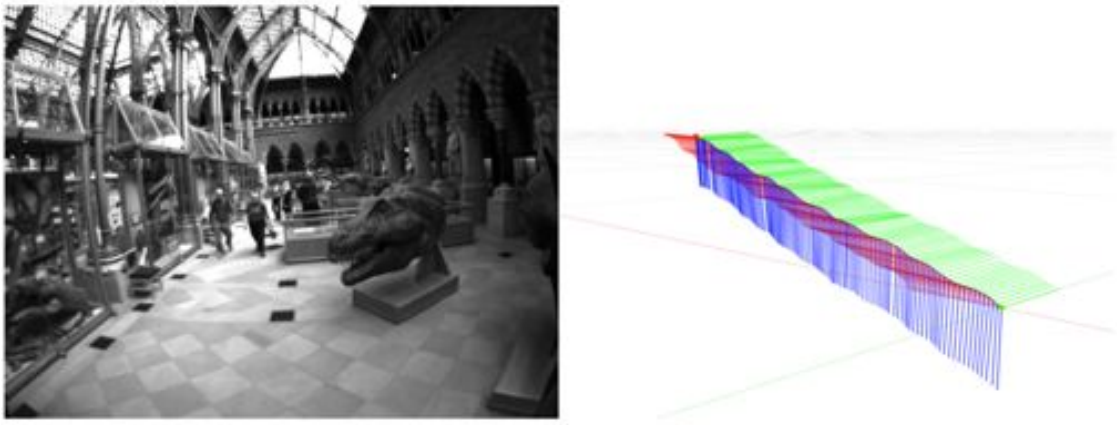
The processing pipeline of the relative bundle adjustment VO system [113] is as follows:

- **Image processing**: Remove lens distortion and filter the images to allow faster matching of features
- **Image alignment**: Estimate an initial 3D rotation using a gradient descent method based on image intensity – this helps with perceptual aliasing
- **Feature match in time**: 3D features are projected into left and right images and matched using a sum-absolute-difference error metric
- **Initialise new features**: Typically 100-150 features are tracked and an even spatial distribution is ensured using a quad-tree

This system has been shown to be robust under difficult conditions (lens flare, motion blur), and to work over large scales. Even though it is a relative representation, metric errors of the order of centimetres over multi-kilometre trajectories are reported in [86]. On the smaller scale environments we are dealing with, we use this system for all stereo camera pose estimates and treat these estimates as error-free.



(a)



(b)

Figure 2.3: Visual odometry. (a) shows sample images from the Spirit rover on Mars. Detected features used for visual odometry are shown in red, and feature correspondences between frames are shown as blue lines. Image from [66]. (b) is example output from the visual odometry system described in [86, 112, 113] and used in this thesis. The camera was helmet mounted and the walker's gait can be seen in the pose estimates on the right hand image. Image from [113].

2.3 3D Data Acquisition

The acquisition of 3D data is covered in detail in Chapter 3, but will be described briefly here to aid understanding of the Mapping and Exploration sections of this chapter.

There are wide variety of sensors which produce 3D measurements of the world: laser scanners, sonars, stereo cameras. The output from any of these is typically a collection of $[x, y, z]^T$ points lying in the sensor's coordinate frame. In Chapter 3 we describe in detail the algorithms which can produce a point cloud from a pair of images captured by a stereo camera.

Now, with an estimate of the sensor position in a global coordinate frame, and 3D data lying in the local coordinate frame of the sensor, we are in a position where we can integrate sensor data into a globally consistent map of the world. It should be noted that this map is *not* the same as the map used by the visual odometry SLAM engine. The SLAM map consists only of sparsely distributed features and does not provide the dense data required for exploration purposes.

2.4 Mapping

A map reflects the structure of the environment at some level of abstraction and broadly speaking can be categorised as either a topological or a metric map.

A purely topological map is concerned with the connections between locations and is typically stored as a graph-like data structure: vertices are locations in the world such as significant features or places, and edges are a connection between two such locations. For example a graph containing vertices labelled 'office', 'doorway', 'corridor' with edges between 'office' and 'doorway', and between 'doorway' and 'corridor', embodies the knowledge that we know of three locations and that if we are at location 'office' we can get to location 'corridor' by first moving along the edge to

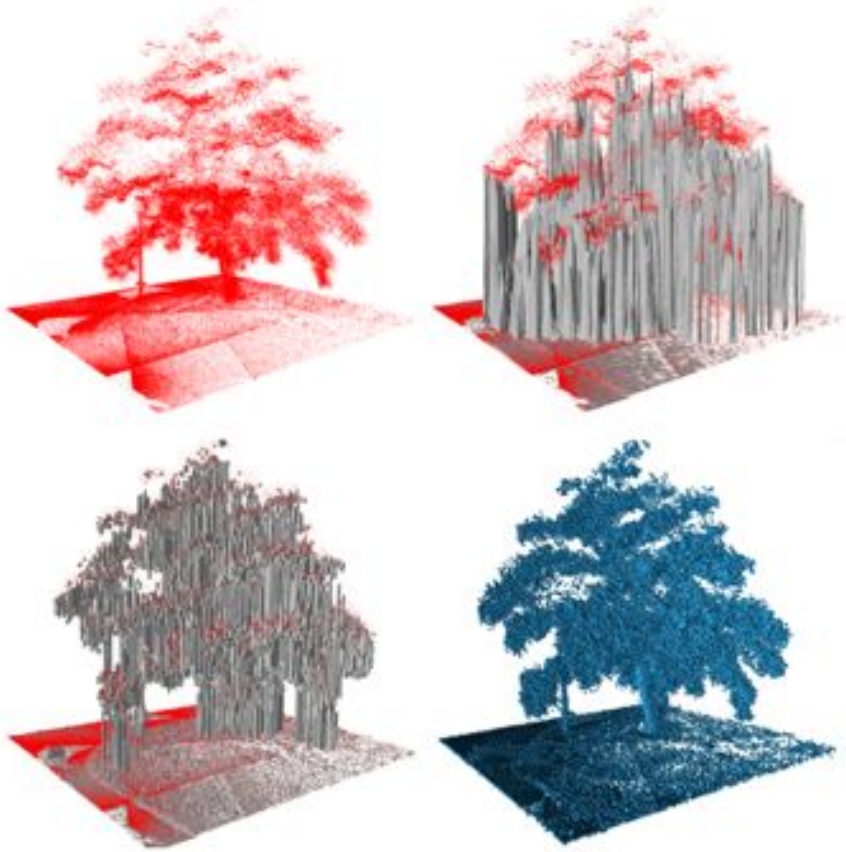


Figure 2.4: Map structures. Comparison of the same tree rendered in various map structures. Top left is a point cloud, top right shows an elevation map, bottom left is a multi-level surface map, and bottom right is an occupancy grid map based on an octree. Image from [135].

'doorway' and then to 'corridor'. Note that this says nothing about the practicalities involved in actually moving along a given edge – this would be the job of a local path planner – it simply says that such a movement is believed to be possible.

A metric map, on the other hand, maintains a model of the environment which corresponds more directly to the real world in a geometric sense. Typical grid based approaches represent the world as a regular, discrete grid in which each cell maintains a probability that that cell is occupied by an obstacle.

2.4.1 Metric map representations

2.4.1.1 Point clouds

Perhaps the simplest map structure is the point cloud [17]. Points detected by a range sensor are transformed into a global coordinate frame and added to an existing collection – a point cloud, as shown in the top left of Fig. 2.4. The simplicity of a point cloud representation comes with a number of disadvantages including the inability to cope with dynamic objects or sensor noise. Concatenating point clouds from multiple scans leads to redundancy in overlapping areas and only occupied space is modelled explicitly – free-space or unseen areas cannot easily be identified.

On a larger scale, the visibility-based fusion of stereo data by [70] quickly combines stereo depth maps (see Chapter 3) from multiple viewpoints. Using consideration of occlusions and free-space violations to find accurate depths for each pixel, they then apply a triangular mesh to the resulting point cloud to produce the final model in real-time. Although visually compelling, a mesh suffers from the same problems as the raw point cloud in terms of free-space identification. Recent scene reconstruction from monocular [82] and structured light cameras [83] present impressive 3D mesh models of smaller scale environments.

2.4.1.2 Occupancy grids

A common approach to explicitly modelling free and occupied (and unknown) space, is to model the world as a discretised grid of 2D squares or 3D cubic volumes, voxels [28, 29, 76, 77]. Grid elements store a probability of occupancy which can be thresholded to classify each voxel as free, occupied, or unknown space. Regular grids are memory intensive, with a footprint of $O(N^3)$ in 3D (where N is the length of the side of the cube of voxels). At high spatial resolutions this can become prohibitive, especially considering that the memory must be allocated beforehand

and no compression of homogeneous regions is performed. They have been used in a wide range of successful robots including the autonomous car Stanley which won the DARPA grand challenge in 2005 [123].

2.4.1.3 2.5D maps

Some of the limitations of a full 3D occupancy grid can be mitigated when dealing with a robot which is constrained to the ground plane. In such a situation a 2.5D or elevation map can be sufficient for navigation and exploration [32, 96]. Rather than using a full 3D grid, an elevation map is a 2D grid where each cell contains an estimate of the surface height at that position.

As such a representation is only able to model a single surface it can only be used when the robot is confined to the ground plane. Obstacles which are higher than the top of the vehicle can be safely ignored. An example of a 2.5D map can be seen in the top right of Fig. 2.4.

The multi-level surface map, introduced by [127], allows multiple surfaces to be represented while still using the 2.5D grid. The limitation of a standard elevation map is that it cannot represent environments which contain multiple levels, such as bridges or underpasses.

A multi-level surface map improves on the basic elevation map by storing in each cell a list of surface patches, rather than a single height value. A surface patch consists of a mean height and a variance, and in this way multi-level surfaces such as underpasses or overhangs can be represented. An example of a multi-level surface map is shown in the bottom left of Fig. 2.4.

A disadvantage of these 2.5D map structures is their inability to encode free-space or unexplored regions volumetrically. This limits their use when trying to perform the exploration task, especially when we are considering full 6DoF movement. For this reason we will focus on the 3D occupancy grid.

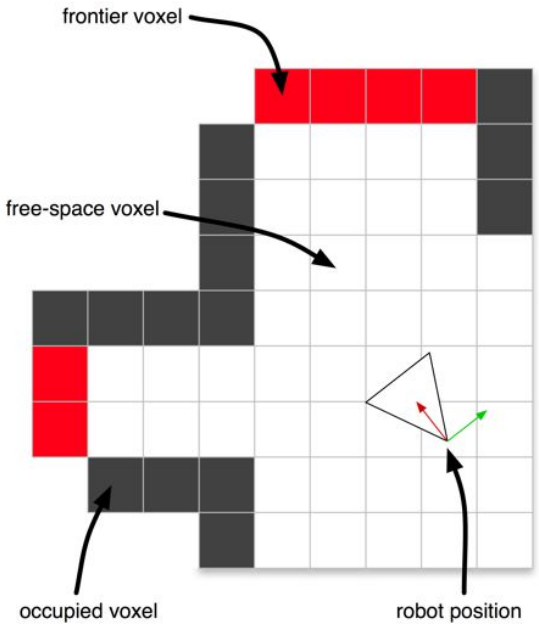


Figure 2.5: Occupancy grid. An occupancy grid map is a discretisation of a workspace into regular volumetric elements, voxels. Each voxel stores a probability of occupancy – an estimate of whether there is an obstacle in that volume of space. These probabilities are thresholded resulting in three classifications: explored voxels, unexplored voxels, and unknown voxels.

2.4.2 Occupancy grid maps

Occupancy grid maps are a method for building consistent maps from noisy sensor data. They rely on known robot poses and so in using them there is an implicit assumption that the Simultaneous Localization and Mapping problem has been solved. This assumption is justified through our use of state of the art SLAM engine, the relative bundle adjustment visual SLAM of Sibley et al. [112]. This system has been shown to give metric errors on the order of 10^{-2} m over loops of many km and so at the scales we are dealing with we make the assumption that the pose estimates are error-free. When performing exploration in a simulated environment we of course have access to the exact pose of the sensor.

It should be mentioned that the Mapping component of SLAM differs from the mapping requirements for exploration in that a typical SLAM system relies on matching sparse distinctive features to come up with pose estimates. These sparse

features are not suitable for producing a densely detailed map for exploration and so we must fill an occupancy grid map using dense sensor data (see Chapter 3) in addition to any map which the SLAM system may produce.

An occupancy grid is a discretisation of the world (in either 2D or 3D) into regularly sized cells – an evenly spaced grid of squares in 2D, or a block of volumetric pixels, in 3D. For consistency we use the term *voxel* to refer to grid cells in both 2D and 3D occupancy grids.

We follow the derivation as given in [122]. Each voxel is referred to by an index – m_i is the voxel with index i. Each voxel contains a binary random variable which maintains the occupancy probability. $p(m_i = 1)$ is the probability that cell m_i is occupied, and conversely $p(m_i = 0)$ is the probability it is empty.

An occupancy grid is a finite collection of these cells:

$$m = \{m_1, m_2, \ldots, m_i\} \tag{2.2}$$

and these will (typically) be arranged in a rectangle or cuboid.

The goal is to calculate a posterior map probability given all sensor and pose measurements recorded so far:

$$p(m \mid z_{1:t}, x_{1:t}) \tag{2.3}$$

where $z_{1:t}$ is the set of all measurements up to the current time t, and $x_{1:t}$ is the corresponding set of robot poses.

Calculating this posterior directly is intractable in any real application. A typical $100m^2$ occupancy grid with voxels of size $10cm \times 10cm$ would have 10000 voxels. As discussed earlier each of these voxels has a binary value – occupied or not – and thus there are 2^{10000} possible maps. Calculating the posterior for a map of this size is not feasible and indeed we will be dealing with maps which are larger and/or higher

resolution, both of which can result in orders of magnitude more voxels.

A simplifying assumption is therefore made to enable map estimation – all voxels are considered to be independent binary random variables. This is not necessarily true (as neighbouring voxels will often be part of the same obstacle for example), but it does allow the posterior calculation to be approximated by the product of the marginals:

$$p(m \mid z_{1:t}, x_{1:t}) = \prod_i p(m_i \mid z_{1:t}, x_{1:t}) \tag{2.4}$$

We are now faced with the (much simpler) problem of estimating the binary occupancy of each cell in the grid, which we do using a binary Bayes filter.

One assumption which should be made explicit is that we assume the map is *static* (no moving obstacles). This assumption means that the occupancy belief is only a function of measurements, and not of poses which simplifies Eq. (2.4)

$$p(m_i \mid z_{1:t}, x_{1:t}) = p(m_i \mid z_{1:t}) \tag{2.5}$$

Applying Bayes' rule to Eq. (2.5), conditioned on the latest measurement z_t, gives us

$$p(m_i \mid z_{1:t}) = \frac{p(z_t \mid m_i, z_{1:t-1}) p(m_i \mid z_{1:t-1})}{p(z_t | z_{1:t-1})} \tag{2.6}$$

The latest measurement, z_t is conditionally independent from all previous poses and measurements: it is only conditional on the map $p(z_t \mid m_i, z_{1:t-1}) = p(z_t \mid m_i)$. Therefore Eq. (2.6) becomes

$$p(m_i \mid z_{1:t}) = \frac{p(z_t \mid m_i) p(m_i \mid z_{1:t-1})}{p(z_t \mid z_{1:t-1})} \tag{2.7}$$

Applying Bayes' rule to the measurement model $p(z_t \mid m_i)$ gives us

$$p(z_t \mid m_i) = \frac{p(m_i \mid z_t)p(z_t)}{p(m_i)} \tag{2.8}$$

which, when substituted into Eq. (2.7) gives us

$$p(m_i \mid z_{1:t}) = \frac{p(m_i \mid z_t)p(z_t)p(m_i \mid z_{1:t-1})}{p(m_i)p(z_t \mid z_{1:t-1})} \tag{2.9}$$

The *odds* of a probability $p(x)$ is defined as

$$odds(x) = \frac{p(x)}{p(\neg x)} = \frac{p(x)}{1 - p(x)} \tag{2.10}$$

Taking the logarithm of this ratio gives us the log-odds representation

$$l(x) = \log \frac{p(x)}{1 - p(x)} \tag{2.11}$$

which results in a computationally elegant Bayes filter update. Additionally, the range of the log-odds function is $(-\infty, \infty)$ and we thus avoid truncation issues for probabilities which are close to 0 or 1. Note that the probability can easily be retrieved from the log-odds:

$$p(x) = 1 - \frac{1}{1 + \exp(l(x))}. \tag{2.12}$$

Taking the odds of Eq. (2.9) gives us

$$\frac{p(m_i \mid z_{1:t})}{1 - p(m_i \mid z_{1:t})} = \frac{p(m_i \mid z_t)}{1 - p(m_i \mid z_t)} \frac{p(m_i \mid z_{1:t-1})}{1 - p(m_i \mid z_{1:t-1})} \frac{1 - p(m_i)}{p(m_i)} \tag{2.13}$$

We therefore get the log-odds representation

$$l_t(m_i) = \log \frac{p(m_i \mid z_t)}{1 - p(m_i \mid z_t)} + l_{t-1}(m_i) - \log \frac{p(m_i)}{1 - p(m_i)} \tag{2.14}$$

where $l_{t-1}(m_i)$ is the previous log-odds value of m_i, $p(m_i)$ is the occupancy prior, and $p(m_i \mid z_t)$ is the inverse sensor model.

The inverse sensor model is a critical part of the update process, and depends on the characteristics of the sensor being used. With each new sensor reading the grid cells which lie in the view frustum of the sensor are updated according to Eq. (2.14). The inverse sensor model is as [135].

The implementation of the inverse sensor model works by performing ray tracing in the grid [1], which is based on the Bresenham line drawing algorithm [14]. A ray is cast from the current sensor position to each point in the point cloud. The voxel in which the ray ends corresponds to the inverse sensor model returning a positive l_{occ} value, while any voxels lying between the sensor and this point are updated with an l_{free} value.

$$\log \frac{p(m_i \mid z_t)}{1 - p(m_i \mid z_t)} = \begin{cases} l_{occ} & \text{if ray reflected in voxel } m_i \\ l_{free} & \text{if ray traversed voxel } m_i \end{cases} \tag{2.15}$$

Log-odds values of $l_{occ} = 0.85$ and $l_{free} = -0.4$ are used as suggested in [135].

2.4.2.1 Implementation

The naïve approach to creating an occupancy grid map in 3D is to create a regular grid of voxels which you hope is large enough to cover fully the region you wish to explore, an approach which scales as $O(N^3)$. Depending on the size of the workspace and the desired resolution of the occupancy grid this can quickly become a problem. A second issue is that once initialised it can be difficult to expand the occupancy grid

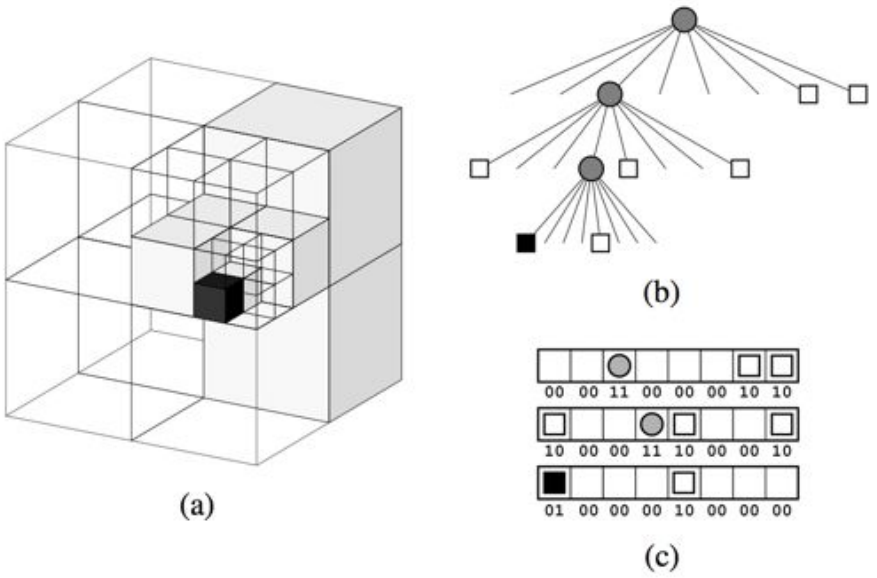


Figure 2.6: Octree data structure. (a) shows the recursive subdivision of a cube into octants. (b) is the corresponding octree data structure. Note that the depth of the tree need not be consistent amongst branches, the only stipulation is that a node must have exactly zero or eight child nodes. (c) efficient storage techniques used in OctoMap mean that the complete octree structure can be stored using only 6 bytes (2 bits per child of a node). Image from [135]

if it is discovered that it is not in fact large enough to contain the entire workspace.

The octree data structure helps to address these concerns and is used in many other systems [30, 63, 78, 89]. An octree is an hierarchical data structure used to partition 3D space by recursive subdivision into eight equally sized octants. Specifically it is a directed acyclic graph where each node has at most one parent node and either zero or eight child nodes. Fig. 2.6 shows the recursion progressing down two levels, the spatial subdivisions on the left and the corresponding octree on the right. Note that the depth of the tree need not be consistent amongst branches, and in fact it is this property which results in the higher efficiency of the octree.

The octree structure has a number of advantages over regular discretised grids: lower memory consumption as large contiguous volumes will be represented by a single leaf node; the extent of the map need not be known at runtime, the octree can expand outwards as needed; it is trivial to obtain subdivisions at different resolutions

by traversing the tree to a given depth.

A freely[1] available occupancy grid implementation based on an underlying octree is the OctoMap library [135]. Developed specifically for robotic applications, it provides an interface to add sensor data to the map and access to the underlying structure as needed.

[1] GNU-GPL licensed

2.5 Exploration

The goal of an exploration algorithm is to increase a robot's knowledge of its workspace by selecting appropriate control actions which will lead it to visit previously unseen areas (or revisit areas of high uncertainty).

Many applications exist, from extra-planetary exploration [58] to finding survivors in the aftermath of a natural disaster [52] to mapping volcano interiors [131]. A robot which can perform such a job autonomously has great value – if it is no longer tied to a human operator a robot can be deployed in environments which could potentially leave it out of contact for long periods of time. Relieving a human of control means that many more robots can be in operation at once, perhaps reporting in once they have exhaustively searched the building they were assigned to.

This problem has been studied from other perspectives as well, such as the question of choosing the next best view when scanning an object [132]. In this they make use of a parametric representation of a known object which is being scanned . This representation is used to determine the next view which will maximise uncertainty reduction of the 3D model. Their scope for exploration is bounded by the known size of the object and the capabilities of the manipulator being used.

2.5.1 Topological methods

Topological exploration describes an exploration method in which a robot constructs a connectivity graph of its environment. For example, [26] describes a robot exploring a graph-like world with limited sensing capability – it cannot make metric measurements of distance or orientation, but can identify when it has reached a graph vertex. A related approach is described by [57] in which distinctive places in the world are identified and allow the robot to differentiate between previously explored and unexplored areas.

2.5.2 Gap Navigation Tree

The Gap Navigation Tree (GNT), introduced by [62, 126], is a data structure designed to maintain a graph of gaps – depth discontinuties with respect to the heading of the robot. The authors show that by chasing down gaps, while maintaining the graph structure by adding and removing gaps as necessary, locally optimal navigation can be achieved. A path planning exploration algorithm based on eliminating these gaps using visibility maps is introduced by [59], and exploration of a simple bounded environment by multiple observers is demonstrated.

The work presented in Chapter 5 has interesting parallels with the Gap Navigation Tree. One could see the graphical approach we adopt as a re-factorisation of this approach and a generalisation to 3D.

2.5.2.1 GNT algorithm

The GNT was created with the aim of achieving exploration while using the minimum amount of information possible. This minimalist approach means that the authors aimed to avoid building costly metric environment models, and require only a single sensor which must be capable of tracking the directions of gaps (discontinuities) in the environment. Assumptions are also made about the environment in which this algorithm will operate: the robot must live inside a two dimensional, simply connected, polygon world.

As the name suggests, the GNT maintains a tree data structure which contains:

- **Root node** The initial position of the robot. To create a GNT a root node is created, and non-primitive nodes for each gap observed from the initial robot location are added as children.

- **Non-primitive nodes** Used to motivate exploration, they correspond to unexplored occluded regions. They result from the observation of a gap, or

from the splitting of an initial node or its non-primitive children.

- **Primitive nodes** A primitive node is added to the tree when a previously visible area becomes occluded, hence causing a gap to appear. Chasing down a primitive node will only result in its disappearence and the re-exploration of previously covered territory.

A robot using the GNT to explore must be capable of rotating to face a gap and to then approach the gap at a constant speed, a process called chasing the gap. After selecting a non-primitive node from the tree the action of chasing the gap can result in four events occurring which affect the non-primitive gap node:

- **Appear** A previously visible part of the world becomes occluded and a new node g is added as a child of the root node.
- **Disappear** A previously hidden section of the environment is revealed and the gap being chased disappears. The non-primitive leaf node is removed from the tree.
- **Split** A gap splits into two distinct gaps due to the environment geometry – g is replaced with the two new gaps g_1 and g_2.
- **Merge** Inverse of the split event: two previously occluded regions are revealed to be part of the same region and the two gaps representing these occlusions are merged.

For a more detailed description of the algorithm implementation details, see [80, 126]. High level pseudocode for the GNT exploration algorithm is given in Algorithm 1.

The GNT is limited to working in a simple 2D polygon world with an idealised sensor, and so the algorithm described in Chapter 5 can be seen as an extension to 3D

in the spirit of the GNT, maintaining the idea of chasing down depth discontinuities during exploration.

Algorithm 1: Gap Navigation Tree Algorithm. From [80].

Input: Set of gaps $G(x)$

Initialize GNT from $G(x)$
while $\exists$ *nonprimitive* $g_i \in GNT$ **do**
    choose any nonprimitive $g_k \in GNT$
    chase(g_k) until *critical-event*
    update(GNT) according to *critical-event*
end
return GNT

2.5.3 Potential methods

Potential methods have a long history in the path-planning literature, see the early work of [18, 19, 51, 61]. Assigning a high potential to the start position, and a low potential to the goal position, a path can be found by gradient descent through the resulting field, assuming appropriate boundary conditions have been set.

Little attention has been given to adapting such methods for exploration, a notable example being [100]. They demonstrate successful exploration of a 2D environment with a sonar sensor. They make use of Dirichlet boundary conditions only, which can result in scalar potential fields with close-to-zero gradient in large-scale environments.

The work presented in Chapter 6 can be seen as an extension of this work to three dimensions, with additional important boundary conditions. Specifically we use Neumann boundary conditions (which specify the first derivative of the field) across obstacle boundaries. We set this first derivative to be equal to 0 perpendicular to obstacles, meaning that no flow can cross a boundary resulting in streamlines which run parallel to walls. More details regarding this type of exploration and the algorithm details can be found in Chapter 6.

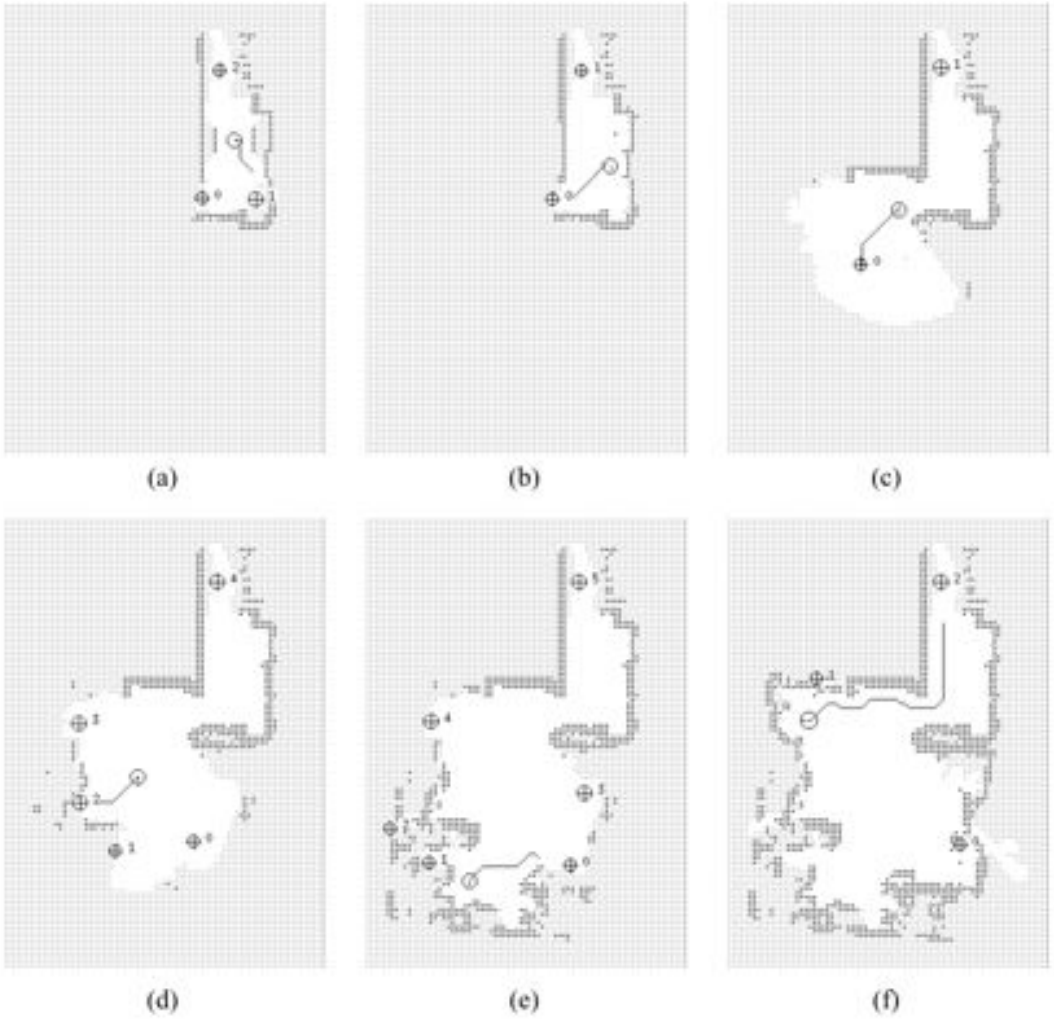


Figure 2.7: Frontier exploration. The original frontier-based exploration method as presented in [136]. The robot has built a 2D occupancy grid map and explores by navigating to the closest frontier. Image from [136].

2.5.4 Frontier methods

The central question of exploration is posed as "Given what you know about the world, where should you move to gain as much new information as possible? [136]". The idea of frontier-based exploration is to move to a boundary between known free-space and unknown space [115, 136] where we can expect a robot will be able to use its sensors to see beyond the boundary, and expand the map. Typically this exploration will be done in an occupancy grid map, as it is easy to identify frontiers,

and free and unexplored space are modelled explicitly. Results using sonar to build a 2D occupancy grid map are shown in Fig. 2.7.

The simplest frontier-based exploration method is to identify the frontier voxel which is closest (in configuration space, or the Euclidean distance in $\mathbb{R}^3$) to the current robot position. The robot is then instructed to move from its current position to this frontier and to perform a sensor scan on reaching it. The process is then repeated until no frontiers remain. This is the method described in [136] and we implement it in Chapter 6.

Closest frontier is a simple and quite effective exploration algorithm but there exist other approaches which take into account how valuable it would be to move the robot to a given frontier. One such algorithm is information gain exploration in which the amount of new information which we might expect to obtain at a given pose is used as the metric for deciding where the robot should move next.

2.5.5 Information gain exploration

In information gain approaches the goal of exploration is twofold – not just to map the environment but to move the robot to maximise the amount of new information obtained with each scan. To do this, new camera poses are randomly hypothesised in the free space of the grid and the number of frontier cells seen from that pose (a measure of expected information) are counted. Such an exploration strategy is employed in [33] in which a voxelised 3D workspace is fully explored with a robot arm. An interesting approach is the particle filter approach of [120] in which exploration of new areas is balanced with improving robot localisation – new poses are selected which maximise expected uncertainty reduction.

We implemented an information gain exploration method for the comparative results in Chapter 6, following the binary gain method of [122]. Although a crude approximation of the expected information gain (voxels are either explored or unex-

plored, there can be no middle ground), it is effective and widely used.

2.5.5.1 Entropy

In information theory the measure of the *information* in a probability distribution $p(x)$ is the entropy $H_p(x)$

$$H_p(x) = -\int p(x) \log p(x) dx \tag{2.16}$$

Intuitively this is a measure of the uncertainty associated with a random variable. $H_p(x)$ is maximal for a uniform distribution and this maps to the intuition that uncertainty is highest when all outcomes are equally likely. As $p(x)$ becomes more peaked $H_p(x)$ decreases, reaching zero when the outcome of a random trial is certain.

2.5.5.2 Information gain

Information gain is defined to be the decrease in entropy between successive measurements. In the context of robotic exploration this means that we measure the entropy of the map at time t and subtract this from the entropy at time $t+1$ to get the information gain. The information gain for a single voxel in an occupancy grid is

$$I(m_i|z_t) = H(p(m_i)) - H(p'(m_i|z_t)) \tag{2.17}$$

where $p'(m_i|z_t)$ is the updated distribution for cell m_i after updating the cell with a new measurement z_t through the sensor model described in Section 2.4.2.

Summing the information gain of every voxel in an occupancy grid gives us a measure of the decrease in entropy of the whole map, and it is this value which we wish to maximise by selecting appropriate new poses in information gain exploration.

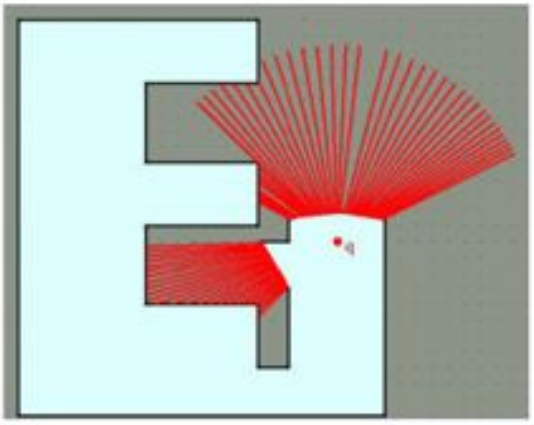


Figure 2.8: Weighted information gain exploration. New poses are chosen based on a weighted sum of potential information gain (found here by ray-tracing from a hypothesised pose), and distance from current pose. Image from [38].

2.5.5.3 Maximising information gain

The inverse sensor model allows us to estimate the information gain which we could expect to obtain if we were to move our sensor to a hypothetical new pose. As described in Section 2.4.2 this works by ray-tracing through the occupancy grid keeping track of the voxels through which the ray passes.

Following the suggestions given in [122] a simple yet effective proxy measure of information gain is the number of frontier voxels which could be expected to be seen from a given pose. It is this measure which we will use in our implementation in Chapter 6.

2.5.6 Weighted information gain

The information gain algorithm described in the previous section is greedy – it will always select the pose which will be expected to yield maximum information. This can result in the robot travelling long distances to reach the area with highest gain, and can result in repetitive and wasteful traversal across previously explored regions.

To balance the expected information gain and the distance to each target, [38]

uses ray-tracing in an occupancy grid map to obtain an estimate of the unexplored area which may be revealed by any given candidate pose. Candidate poses are ranked using the metric

$$g(q) = A(q)\exp(-\lambda L(q)) \tag{2.18}$$

where A(q) is a measure of the information gain expected at the q (the number of frontier voxels seen from a candidate pose), L(q) is the path length from the current pose to the candidate pose, and λ is a weighting factor which balances the cost of motion against the expected information gain. A version of this weighted evaluation of candidate poses is used as one of the exploration strategies in the results section of Chapter 6.

2.6 Assumptive and Partially Observable Planning

Roughly speaking approaches to planning can be subdivided into *assumptive* planning [88] and *partially observable* planning.

Assumptive planning makes planning and exploration tractable by only taking into account the most likely state of a given map grid cell. For example an assumptive planner will apply a threshold probability to an occupancy grid map and plan over a simplified map where every cell is either occupied or free.

Partially observable planners use the knowledge that the state of the world is only partially observable – that is we can only reason that a grid cell is occupied based on past sensor measurements. As well as the binary occupied/free distribution we may need to consider the probability that a given cell is a road, or contains grass, or water.

Considering the fully expanded terrain distribution makes the planning problem exponential in the number of grid cells in the environment and for this reason partially

observable methods have long been considered intractable for real problems.

Although not used in the work of this thesis, it is still worth describing the approaches to motion planning based on Partially Observable Markov Decision processes, as they are they are the leading approach to partially observable planning.

2.7 Partially Observable Markov Decision Processes

Partially Observable Markov Decision Processes (POMDPs) are a framework used to probabilistically model sequential decision processes [116]. When applied to the robot exploration and planning problem they take into account the fact that a robot's sensors and actuators are inherently noisy and imperfect, and therefore that observations of the current state, and the result of control actions to change state, are stochastic in nature.

The current robot state is modelled as a *belief* – a probability distribution over all all possible robot states. A planning algorithm selects *actions* based on the expected utility of choosing an action a given that a robot is in state s.

For example if action a will result in the robot being in a goal state s_g then this will have a high reward value. Non-goal states will have low rewards. The goal of a POMDP is to select the optimal policy – where a policy specifies an action to take given a state. The optimal policy will be the policy which maximises the expected reward over some time horizon. Note that this horizon can be infinite, but future rewards must be discounted proportional to how distant they are to ensure the total reward is finite.

Put more formally, a POMDP is a 6-tuple (S, A, O, T, Z, R). S is the set of robot states: possible $[x, y, \theta]^T$ poses in a 2D world for example. A is the set of actions which are available to the robot: for example "drive to the neighbouring grid cell". O is the set of observations the robot has made of the environment.

T is a set of conditional transition probabilities $T(s, a, s') = p(s'|s, a)$. For each action in state s, what is the probability of transitioning to new state s'?.

Z is the observation model $Z(s, a, o) = p(o|s, a)$. The probability that the robot will make observation o in state s after taking action a.

Finally R is the reward function $R(s, a)$ which is used to define the desired robot behaviour.

2.7.0.1 Choosing a reward function

The reward function R is a real-valued function which returns a measure of the utility of performing action a while in state s. The behaviour of the robot as it explores the world depends on the values returned by this function. Some possible desired behaviours are as follows

- **Minimise total distance travelled** A situation in which it is expensive to move the robot and so we want the total distance travelled to be as low as possible. The reward function here would penalise travelling long distances while giving higher rewards to actions leading to physically close goal states.

- **Minimise time taken to explore** In most cases this will be extremely similar to the previous example: time and distance are directly comparable if we make the simplifying assumptions that there are no accelerations required to reach maximum speed, and that the robot can orient itself instantly in any direction. If, for example, turning was a costly operation then the reward function may penalise actions which require large changes in orientation.

- **Minimise number of scans taken** If the expensive part of the robot's operation is taking sensor readings then we may want to encourage the exploratory behaviour to be frugal with the total number of scans taken. For example if taking a scan requires stopping and waiting for 2 minutes while a laser scanner

slowly pans up and down. In this case we could set the reward function to reward moving to a state from where we could expect to observe the largest fraction of unknown areas.

- **Explore most interesting areas quickly** In this example we want the robot to quickly obtain scans of the most interesting areas of the map (where areas with some valuable property such as likelihood of containing a resource are defined as interesting). The reward function here would give large rewards to actions which with high probability will result in a transition to a state from which we can expect to observe an interesting area.

2.7.0.2 Choosing a policy

To select an action given the current belief state we need a *policy* $\pi(b) \rightarrow a$. Ideally we want to find the optimal policy, that is the policy which maximises the expected sum of total rewards over some number of future timesteps. This is typically found iteratively: remember that when choosing an action we must take into account not only the immediate reward, but the possible future rewards which could result from making that choice.

An optimal policy is one which greedily maximises Bellman's equation [8]

$$V^*(b) = max_a[\sum_s b(s)R(s,a) + \gamma \sum_o P(o|a,b)V^*(\tau(b,a,o))]. \tag{2.19}$$

where $V^*(b)$ is the optimal value function given a belief b, and τ is belief transition function which updated the current belief given new a observation and action. The details of finding $V^*(b)$ can be found in [62], through the process of value iteration – using dynamic programming techniques to iteratively improve the solution until convergence is reached.

2.7.0.3 Intractability

Although providing a principled framework for planning under uncertainty, solving POMDPs for large scale real-world problems are still intractable. Attempting to solve over a continuous belief space is infeasible and so approximate solutions such as point based methods [24] have been developed. These apply dimensionality reduction techniques to the belief space and sample points in the reduced space before producing approximate solutions which focus on areas of the belief space which have a high probability of being encountered.

Even with these approximations the scale of problems which can be tackled online are severely limited. Problems with hundreds of states can now be approximately solved in seconds on modern hardware, but this is orders of magnitude lower than the situations we will encounter. More realistic problems still require many minutes or hours [97] to solve, and even navigation with a real quadcopter robot through a simplified grid world can take 15 minutes to solve [24].

For practical reasons we therefore will therefore employ the assumptive exploratory techniques in this thesis. Our occupancy grids will maintain an occupancy probability but when it comes to planning we condense this to a binary value for each cell: occupied or free. Knowledge of the robot's 6DoF position is assumed to be known, given the state of the art visual odometry system we employ [113].

2.8 Evaluation of Exploration Strategies

A couple of experimental evaluations have been performed which compare the exploratory behaviour of different algorithms. Presented in [3] is an evaluation for robots confined to a 2D ground plane in a block-world simulated environments. They compare random exploration, a greedy information gain strategy, and the weighted information gain of [38]. They find that the greedy strategy has good performance

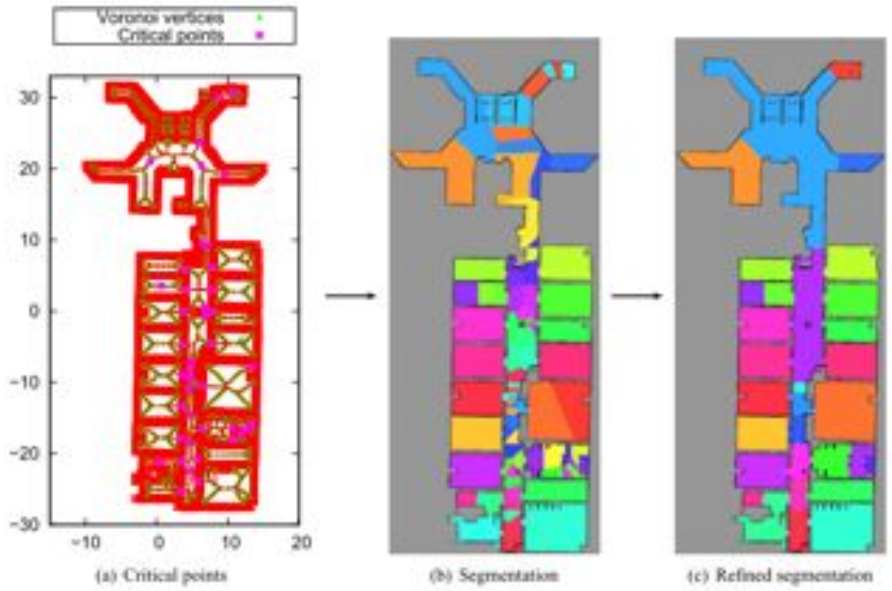


Figure 2.9: Map segmentation. Based on the work of [121], the map segmentation presented in [45] involves finding critical points in a Voronoi diagram of the map's free space. Frontier voxels which lie outside of the current segment are then given lower weighting, encouraging the robot to fully explore each room before moving on. Image from [45].

in terms of number of scans required to map an environment, but the resulting path lengths are much longer than strategies which take into account distance to candidate poses.

Another evaluation of exploration strategies is presented in [45]. They compare random exploration, a weighted information gain strategy, and some modified closest frontier strategies. Like [3] they find that closest frontier repeatedly finds the shortest path length – they manage to improve performance slightly by introducing a map segmentation algorithm to avoid multiple visits to the same rooms. This is based on the work of [121], and involves finding critical points in a Voronoi diagram of the map's free space. Typical results are shown in Fig. 2.9.

Frontier cells which are found in the same segment as the robot are selected as exploration targets ahead of all other frontiers, ensuring that rooms are fully explored before the robot leaves through a doorway.

Chapter 3

Acquiring 3D Data

3.1 Introduction

This chapter is about the acquisition of 3D data from sensors mounted on a mobile robot. Obtaining accurate measurements of the environment is critical for almost every task a robot might face – from obstacle avoidance to object recognition.

This challenge of sensing the world has been addressed in numerous ways, using various sensor modalities with different properties. We consider our choice of sensor, keeping in mind the following provisos: the chosen sensor must be able to generate an accurate and dense workspace representation and it must be practical to mount it on a range of mobile robots, from the wheeled robots of the Mobile Robotics Group, to the end of an OC Robotics snake-arm.

3.1.1 Sensor classes

If we consider a sensor to be a device which measures some physical property then we quickly realise that this covers a huge variety of technologies. Narrowing our scope to those sensors pertinent to mobile robotics and following the class decomposition given in [25] we get the following breakdown:

(a) SICK LMS 291 **(b)** Microsoft Kinect **(c)** Point Grey Bumblebee

Figure 3.1: Range sensors.
(a) Laser: Active sensor which measures distance by firing a laser into the environment and measuring time-of-flight of reflection. Image from [114]
(b) Kinect: Active sensor which obtains 3D data by projecting a structured light pattern into the environment and measuring distortion. Image from [71]
(c) Stereo camera: Passive sensor in which 3D measurements are obtained by finding corresponding features between two offset images. Image from [98]

Range sensors measure distances from sensor to objects in the world. Laser range finder, stereo camera, structured light projection (Kinect), sonar, etc.

Absolute position sensors measure robot pose in some global or absolute frame. Examples include GPS and ViCon camera networks.

Environmental sensors measure environmental properties such as humidity or temperature.

Inertial sensors measure differential properties of the robot's position such as acceleration and orientation.

Our robots have a wide range of sensors, falling into many of the above categories. However this chapter will focus specifically on range sensors, as our goal is to build 3D environment maps and thus we require 3D measurements of the environment.

We will look at a number of alternative technologies and the advantages and disadvantages of each, and then describe in detail the algorithms needed to retrieve range data from our sensor of choice: a binocular stereo camera

(a) Laser **(b)** Kinect **(c)** Stereo

Figure 3.2: Sensor data. Example range data from each of the three sensors shown in Fig. 3.1. (a) shows laser data gathered in a large scale outdoor environment [117]. (b) is RGB+depth data from the Kinect and highlights its capabilities at capturing small scale indoor scenes. (c) stereo falls somewhere between these two modalities – it works well at small to medium scale, and at higher resolution than the Kinect.

3.1.2 Range Sensors

A well-known example of a range sensor is that of echo-location as used by bats and dolphins. By emitting a pulse of sound and measuring some aspect of the reflected pulse (phase shift, time of flight, attenuation) these creatures are able to infer something about their environment – the distance to the seafloor for example.

This is an example of an *active* sensor, a sensor which actively emits energy into the environment. The alternative is a *passive* sensor such as a camera, which passively absorbs energy (photons in the case of a camera) and processes it.

An active sensor has the advantage of being able to operate in environments where there is not enough ambient energy for a passive sensor. A camera in a dark room will struggle to obtain quality images, and they will almost certainly be contaminated with high levels of noise. In the same situation a laser range finder will have no trouble.

Disadvantages of active sensors include increased power consumption, and the fact that they are actively emitting energy which, in fragile environments, may not be acceptable.

3.1.2.1 Laser Rangefinder: SICK LMS 291

The laser range finder is ubiquitous in the field of mobile robotics. The SICK LMS 291 shown in Fig. 3.1(a) is a popular choice for many mobile robotics applications. They are accurate on large scales (up to 80 m) and require little post-processing to produce 3D points.

Internally a laser is fired against a rotating mirror which angles the beam into the environment over a range of angles, typically an 180° arc in front of the sensor. Measuring the time-of-flight of reflected laser light from objects in the world allows it to produce a point cloud covering a thin strip of the world which lies in the sensor plane.

To produce 3D point clouds the device must be externally actuated to rotate the scan plane through the environment. The result shown in Fig. 3.2(a) comes from mounting two scanners in a vertical orientation on the side of a moving robot so that they sweep through the world (more information in Chapter 4).

3.1.2.2 Infrared Structured-light: Kinect

In late 2010 Microsoft released the Kinect sensor, shown in Fig. 3.1(b), as a peripheral input device for the Xbox 360 games console. It projects a pre-defined pattern of infrared light into the environment which is observed by an infrared camera capturing 640x480 frames at 30Hz. Objects in the world result in distortions in the reprojection of this structured light field. By measuring the offset (or disparity) between the expected reprojection and the measurement the Kinect can produce a 3D point cloud at framerate. The projection used is proprietary, but a similar structured light projection system used to aid depth reconstruction from stereo images is described in [56].

Internal calibration data is then used to map colour data from a second camera to the point cloud, resulting in full RGB+D (red, green, blue, depth) data as shown

in Fig. 3.2(b).

The Kinect provides excellent data, as is demonstrated in the mapping work of [23, 42, 83], but is limited to a range of around 2-6 m. It also will not work outdoors as the ambient radiation from sunlight drowns out the infrared projection and saturates the infrared camera.

3.1.2.3 Stereo vision: Bumblebee

A binocular stereo camera consists of two monocular cameras, rigidly connected such that there is considerable overlap between the images produced by each camera. If a feature in one image can be identified in the second image, and the physical characteristics of the camera are known, then the position of the feature can be triangulated resulting in a 3D point. Searching over the entire image and looking for correspondences at each pixel results in a dense point cloud as shown in Fig. 3.2(c).

Physically smaller and lighter than a Kinect or a laser scanner, a high resolution stereo camera can be mounted on all our mobile robots and on the tip of a snake-arm robot. A stereo camera can produce 3D point clouds both indoors and outdoors, a caveat being that surfaces must be sufficiently textured to allow correspondences to be found – in the case of a perfectly blank surface the projected light from a Kinect could reveal the 3D structure where a stereo camera would fail.

The vast majority of environments encountered by a mobile robot have sufficient texture for stereo however, and a stereo camera was used to capture all of the real-world data used in this thesis. We use a Point Grey Bumblebee stereo camera, shown in Fig. 3.1(c). The pertinent physical characteristics are as follows:

Model	Point Grey Research Inc. Bumblebee2
Resolution	Sensor resolution (each camera) 1024x768
Framerate	20 Hz in colour or mono
Focal length	3.8mm with 66° horizontal field of view
Baseline	120mm (horizontal separation of the two cameras)

We will now discuss the processing of images and the algorithms used to find correspondences between stereo frames.

3.2 Stereo

A binocular stereo camera consists of two cameras, rigidly connected such that in a typical scene there is considerable overlap between images produced by each camera. Recovering 3D data from multiple 2D images is known as the multi-view stereo problem [41], and binocular stereo is a special case of this problem. Throughout this document we will use the words *stereo* and *stereo camera* as shorthand for *binocular stereo* and *binocular stereo camera* respectively.

A Bumblebee stereo camera can provide accurate 3D data over a range of approximately 1 m to 10 m. This is heavily dependent on the processing of the images as false positive correspondences between images will lead to erroneous 3D points. The task of finding these correspondences is discussed in Section 3.2.5, but before this can take place the images must be preprocessed to remove lens distortion and to compensate for sensor noise. This process is discussed in Section 3.2.1.1,

First, we will look at an ideal camera model and the mathematics which describe projection from 2D images to points in $\mathbb{R}^3$.

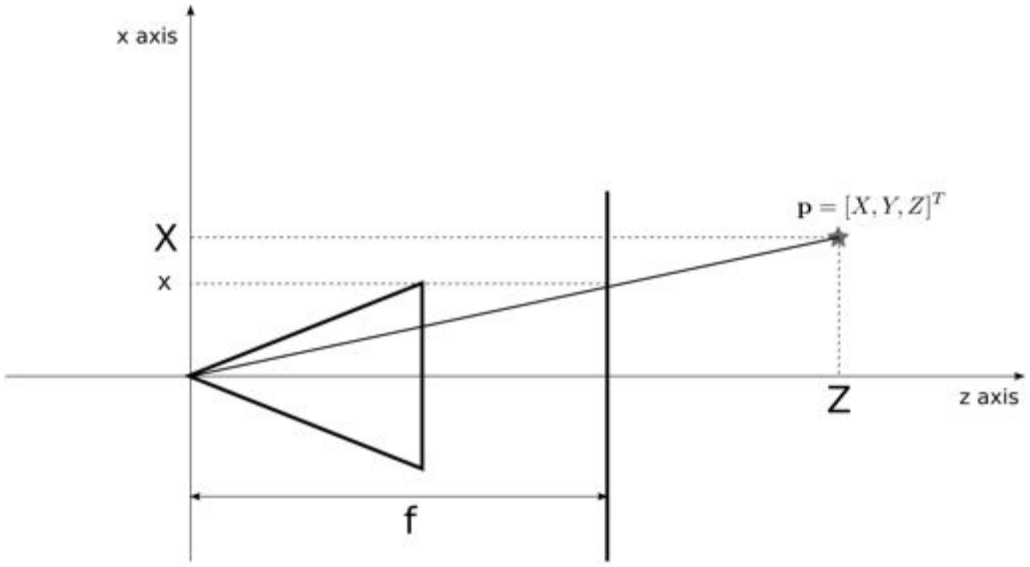


Figure 3.3: Pinhole camera model. The star represents a point on the surface of an object which is being imaged by the camera.

3.2.1 Projective Camera Models

A *camera* is, in the general sense, a mapping from $\mathbb{R}^3$ to a 2D image [41]. A *camera model* is a matrix which represents this mapping. It is instructive to consider the case of the ideal pinhole camera model, shown in Fig. 3.3, before adding the complications of real-world irregularities such as lens distortion.

An image of a point, $\mathbf{p}$ in front of the camera is formed on the image plane by a light ray passing from $\mathbf{p}$ through the centre of projection of the camera. The focal length, f, of the camera is the distance of the image plane from the centre of projection.

In the example in Fig. 3.3 a ray is emanating from the point $\mathbf{p} = [X, Y, Z]^T$, and intercepting the image plane at $[x, y]^T$ (the y dimension is perpendicular to the page and is not shown for clarity).

From similar triangles in Fig. 3.3 we see that the following relationships hold

$$\frac{x}{f} = \frac{X}{Z} \tag{3.1}$$

$$x = \frac{fX}{Z} \tag{3.2}$$

and equivalently for y

$$y = \frac{fY}{Z} \tag{3.3}$$

The mapping from a point in 3D space, $\mathbb{R}^3$, to 2D space, $\mathbb{R}^2$, can therefore be written as

$$\{\pi(X, Y, Z) : \mathbb{R}^3 \to \mathbb{R}^2\} = \left(\frac{fX}{Z}, \frac{fY}{Z}\right)^T \tag{3.4}$$

This can be written in terms of matrix multiplication using homogeneous coordinates

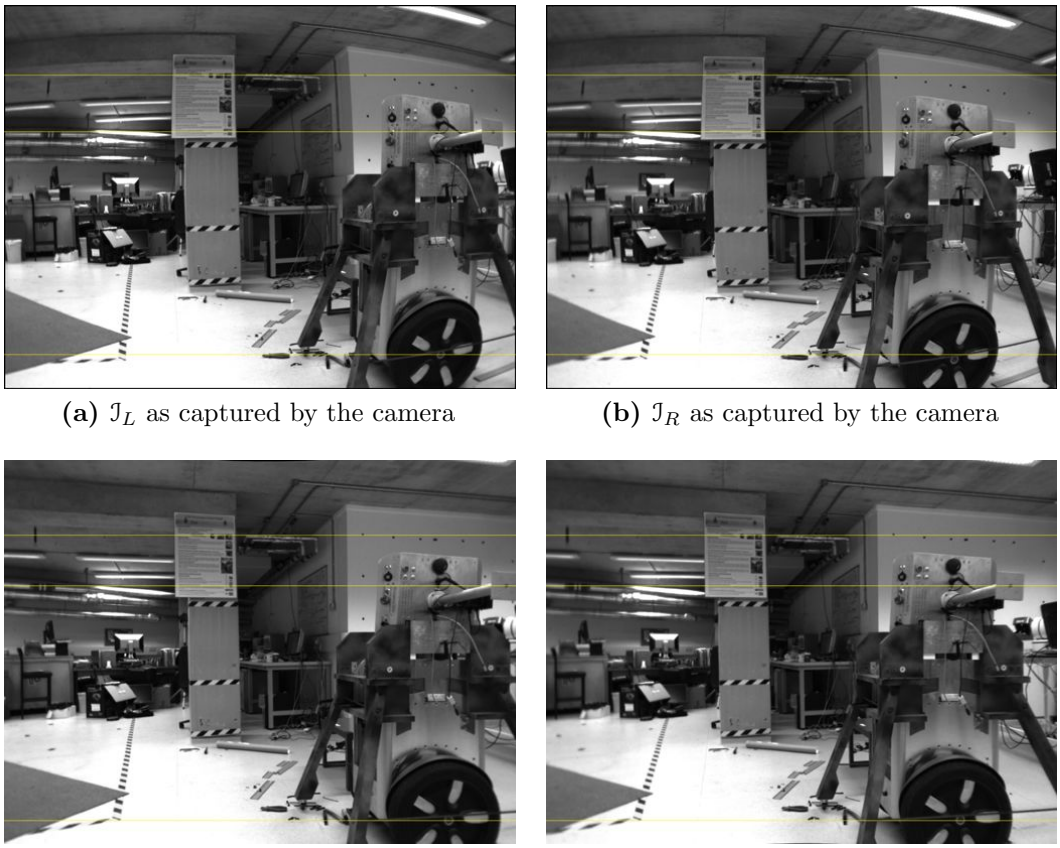
$$\begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f_x & 0 & 0 & 0 \\ 0 & f_y & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = P \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.5}$$

The 3×4 matrix P is known as the *camera projection matrix*. More data can be added to this matrix to account for principal point offset (intersection of optical axis and image plane), non-square pixels in the CCD sensor, and the skew parameter

(necessary if the x and y axes are not perpendicular).

$$\begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} \alpha_x & s & x_0 & 0 \\ 0 & \alpha_y & y_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = K \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.6}$$

Matrix K is called the *camera calibration matrix*. α_x and α_y are the focal lengths scaled due to non-square pixels, (x_0, y_0) is the position of the principal point, and s is the skew parameter. These parameters are the *intrinsic camera parameters* as they deal with characteristics which are internal to the camera.



(a) $\mathcal{I}_L$ as captured by the camera

(b) $\mathcal{I}_R$ as captured by the camera

(c) $\mathcal{I}_L$ undistorted and rectified

(d) $\mathcal{I}_R$ undistorted and rectified

Figure 3.4: Undistortion and rectification. Yellow lines are overlaid to highlight the effects of rectification. Consider the middle of the Segway wheel in the bottom right of all four images. In the unrectified images (top two images), we see that the matching points are displaced vertically between the two images. In contrast, the same points in the bottom two images lie on the same horizontal scanline, reducing the search problem from 2D to 1D.

3.2.1.1 Camera Calibration

No lens is perfect – distortions introduced by the manufacturing process are inevitable. To compensate for these imperfections a calibration must take place which results in a set of parameters which are used to remove distortion from captured images.

The process of calibrating a stereo camera results in two sets of parameters – *intrinsic* parameters and *extrinsic* parameters. The *extrinsic* camera parameters are those describing the physical configuration of the two lenses – the 6DoF transformation from the centre of the one lens to the centre of the other. The *intrinsic* camera

parameters describe the properties of each lens – radial and tangential distortions, the focal length etc.

Most stereo matching algorithms assume that the two images from the camera have had lens distortion compensated for, and are aligned such that the image of a point appears on the same horizontal scanline in both images. The lens distortions are removed using the intrinsic distortion parameters, and rotated and translated using the extrinsic parameters such that corresponding scanlines are aligned. These two stages are known as *undistortion* and *rectification* respectively.

The two lenses in the Bumblebee are enclosed in a high quality, rigid metal enclosure with a fixed baseline. This means that calibration can be done once, and once only, and that the cameras will stay calibrated even if subjected to quite severe impact. Onboard factory calibration data can be extracted from the Bumblebee and a look-up table of unrectified to rectified pixel mappings is built. This is used to undistort and rectify images at framerate.

Figure 3.4 shows the results of undistorting and rectifying a pair of images captured by the Bumblebee2. The yellow lines are overlaid to highlight the effects of rectification. Consider the middle of the Segway wheel in the bottom right of all four images. In the unrectified images (top two images), we see that the matching points are displaced vertically between the two images. In contrast, the same points in the bottom two images lie on the same horizontal scanline, reducing the search problem from 2D to 1D.

3.2.2 Binocular Stereo Techniques

The binocular stereo correspondence problem is as follows: We are given two images, $\mathcal{I}_L$ and $\mathcal{I}_R$, and wish to determine pixel to pixel correspondences between the two images. That is, for a given pixel $\mathcal{I}_L(x_0, y_0)$, we wish to determine x_1 and y_1 such that $\mathcal{I}_R(x_1, y_1)$ is the image of the same physical point, $\mathbf{p}$, in the right image.

Once we have calculated a correspondence, we wish to compute the three dimensional coordinates of $\mathbf{p}$ relative to some reference frame. This reference frame is typically the centre of the left camera, but it may be some global frame if we know the 3D pose of the camera. The difference in horizontal displacement between two corresponding pixels is defined as the *disparity*, $d = x_0 - x_1$.

Figure 3.5 shows a pair of ideal cameras that have both imaged a point in space. This example shows a point in a 2D world being projected into a 1D image, but this is easily extendable to 3D – consider the one dimensional image to be a single horizontal scanline from a two dimensional image. The image pixels that correspond to this point have been found in both images – u_L is the shift from the left hand edge of $\mathcal{I}_L$ to the correct pixel in $\mathcal{I}_L$, and u_R is the shift from the left hand edge of $\mathcal{I}_R$ to the correct pixel in $\mathcal{I}_R$.

The focal length and baseline are physical properties of the stereo camera, and are assumed to be known. The unknown value, the distance of the point along the z-axis, can be calculated by looking at the similar triangles shown in Fig. 3.5

$$d = u_L - u_R \tag{3.7}$$

$$\frac{z}{b} = \frac{f}{d} \tag{3.8}$$

$$z = \frac{f.b}{d} \tag{3.9}$$

As Eq. (3.9) shows, a large disparity corresponds to a point which is physically close to the camera. Conversely a small d gives a large value for z. As described

in [68] the nonlinearity of Eq. (3.9) means that as d approaches 0, small errors will correspond to large errors in z. For this reason we are wary of trusting the depth measurements generated by small disparities.

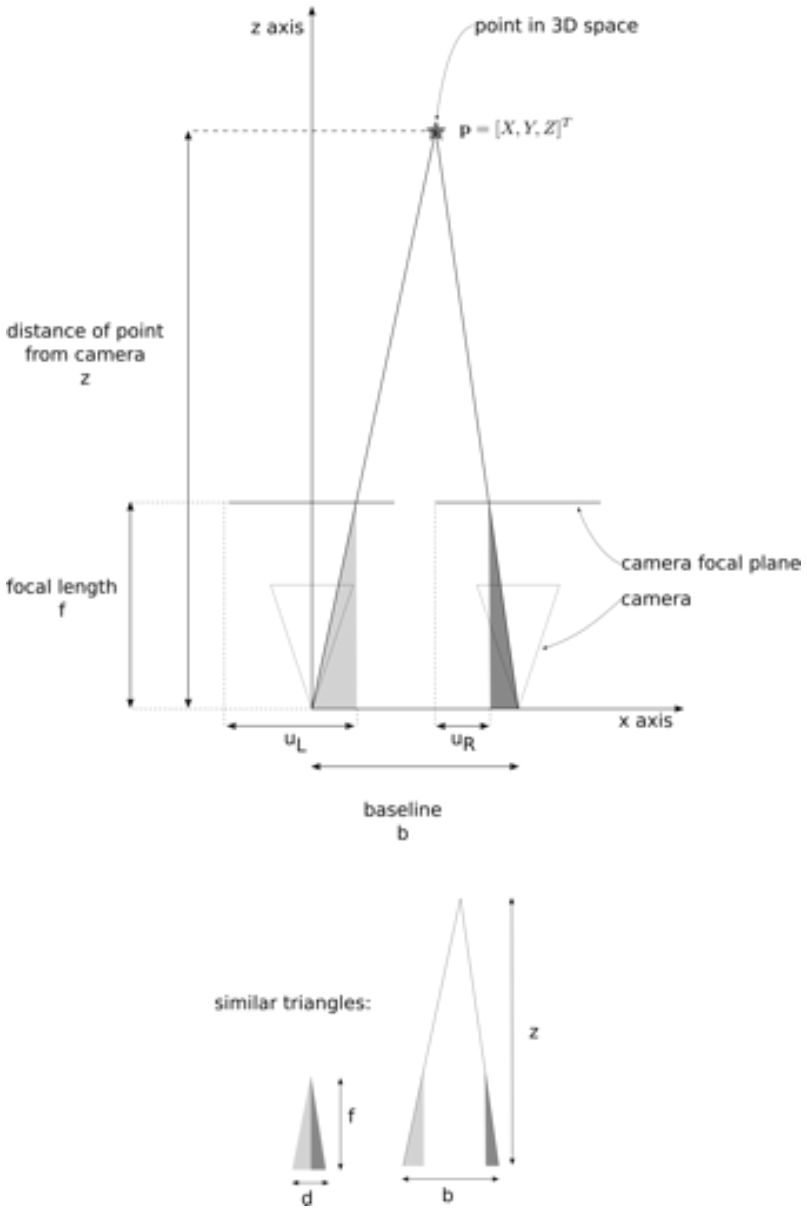


Figure 3.5: Disparity calculation. The upper image shows the left and right cameras of a stereo pair as ideal pinhole cameras imaging a point in space. The lower part of the image shows similar triangles constructed from the upper image, used to generate the disparity equations. The shading of the triangles is the same in the upper and lower images.

3.2.2.1 Feature-based Stereo and Dense Stereo

The goal is to find correspondences between pixels in $\mathcal{I}_L$ and $\mathcal{I}_R$. One approach is to use a point of interest (POI) detector to find distinctive patches, or features, in $\mathcal{I}_L$ and then search for the same distinctive patch in $\mathcal{I}_R$. Lowe's Scale Invariant Feature Transform (SIFT) [64] is a popular choice – the features it selects are largely invariant to scale, illumination and local affine distortions. Other more recent, faster alternatives are SURF [7], FAST [103, 104], and BRIEF [16].

The benefit of this approach is that if the feature detector finds highly distinctive features in $\mathcal{I}_L$ then it is relatively simple to locate the same feature in $\mathcal{I}_R$ with a high confidence. However this means that only a sparse set of pixel correspondences are found – running SIFT on a typical 512×384 image tends to find around 1500 features. Contrast this with the approximately 150000 correspondences found in the same image by the window based method described Section 3.2.5.

We need a dense representation of the workspace for our purposes, and the sparse point set derived from feature based stereo does not meet our requirements. A *dense stereo* approach attempts to find a pixel to pixel correspondence for *every* pixel in $\mathcal{I}_L$, not just patches deemed to be distinctive by a feature detector. As shown in Section 3.2.6, when done properly this can result in typical matches for upwards of 80% of pixels in an input image pair.

There are two main approaches to dense stereo matching which we will consider: local matching through sliding window correlation, and global matching techniques which tend to involve minimising an energy term over the entire image. The output of both approaches will be a *disparity map*, $d(x, y)$. $d(x, y)$ has the same dimensions as the input image $\mathcal{I}_L$, and each pixel stores either the horizontal disparity to the matching pixel in $\mathcal{I}_R$, or an error code if no correspondence could be found.

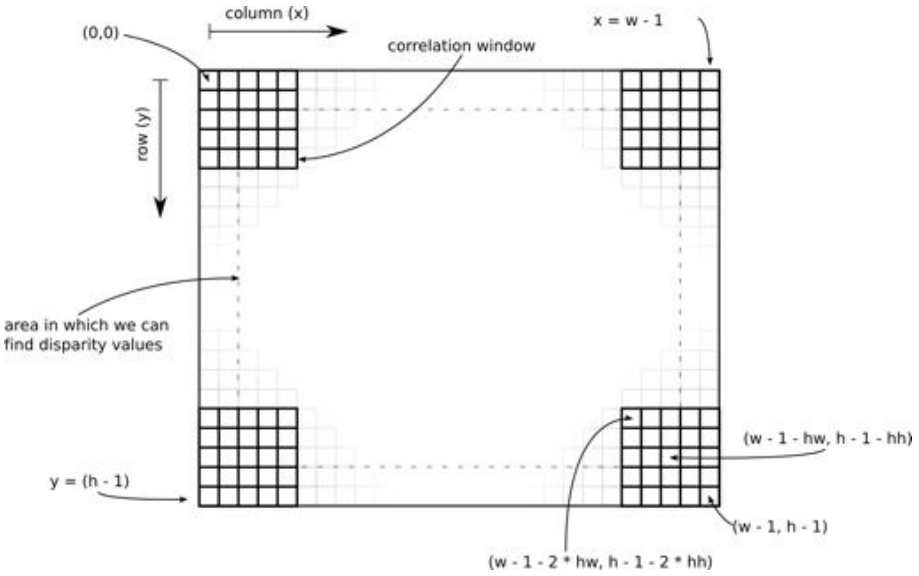


Figure 3.6: Image coordinate conventions. Pixels in an image are referenced with respect to an origin in the top-left of the image, and are addressed as a row and column pair. A disparity search window of size 5×5 as shown here results in no disparity values being calculated in a 2 pixel border around the image.

3.2.2.2 Local Dense Stereo

We now concentrate on the class of stereo algorithms which involve using a sliding-window to identify correspondences. Figure 3.6 shows the layout of a typical image and the coordinate system used to address pixels. Around each pixel in the left image we take a small window (5×5 is shown in Fig. 3.6), and try to find a matching window in the right image.

Figure 3.7 shows a real example: The lower left hand image is the reference image, with the correlation window centred on the pixel that we wish to find a correspondence for.

To find a match we slide the correlation window across the right hand image, and at each position measure the similarity of the pixels underneath the current position to the reference window. There are many similarity measures found in the literature, a selection are enumerated in [15] and in [107]:

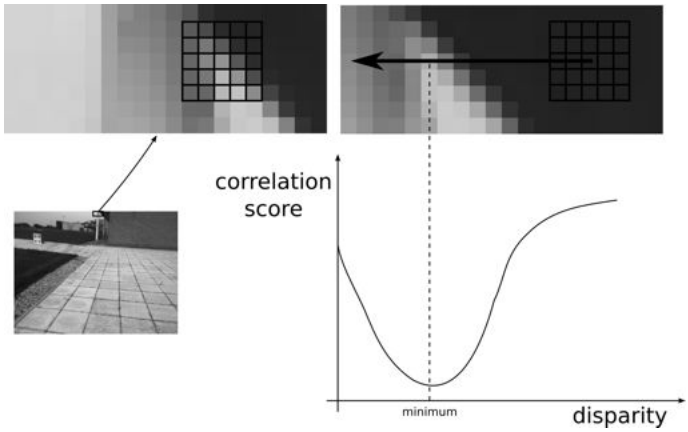


Figure 3.7: Local window matching. We take a window around the pixel in $\mathcal{I}_L$ which we wish to generate a correlation curve for. We then search along the corresponding horizontal scanline in $\mathcal{I}_R$ by sliding a window of the same size along $\mathcal{I}_R$ and at each pixel taking some measure of the similarity between the two windows. A common and fast similarity measure is Sum of Absolute Differences. This generates a correlation curve as shown, and the minimum of this curve gives us the best match for the original pixel in $\mathcal{I}_L$.

Sum of Absolute Differences (SAD)

$$\sum_{row}\sum_{col} |\mathcal{I}_L(row, col) - \mathcal{I}_R(row, col + d)|$$

Sum of Squared Differences (SSD)

$$\sum_{row}\sum_{col} (\mathcal{I}_L(row, col) - \mathcal{I}_R(row, col + d))^2$$

Normalised Cross-Correlation (NCC)

$$\frac{\sum_{row}\sum_{col} (\mathcal{I}_L(row, col) - \bar{\mathcal{I}_L}).(\mathcal{I}_R(row, col + d) - \bar{\mathcal{I}_R})}{\sqrt{\sum_{row}\sum_{col} (\mathcal{I}_L(row, col) - \bar{\mathcal{I}_L})^2.(\mathcal{I}_R(row, col + d) - \bar{\mathcal{I}_R})^2}}$$

The most commonly used similarity measure is Sum of Absolute Differences (SAD), employed by [4, 44, 81, 129], amongst others. Its big advantage is the low computational complexity as it involves only one subtraction per pixel.

The size of the correlation window must be considered and it must be: "large enough to include enough intensity variation for matching but small enough to avoid the effects of projective distortion" [111]. A larger window is less sensitive to image noise and it will be easier to find a match, but a smaller window will provide more

fine grained results. The computational cost of calculating the similarity measure also increases with the size of the correlation window. The window size will depend on the application, but is typically somewhere between 7×7 and 21×21.

3.2.2.3 Global Dense Stereo

Global dense stereo refers to the introduction of non-local constraints into the disparity map calculation, which can greatly increase the computational complexity of stereo matching [15]. One approach is dynamic programming – searching for an optimal path through disparity space by recursive calculation of sub-paths [130].

Generally, modern global stereo techniques cast the problem of disparity map calculation in an energy minimisation framework. Scharstein and Szeleski [107] describe the problem as follows. We wish to find a disparity map, d, which minimises a global energy function

$$E(d) = E_{data}(d) + E_{smooth}(d) \tag{3.10}$$

The data term, $E_{data}(d)$, is a measure of how well the disparity map matches the input data

$$E_{data}(d) = \sum_{(x,y)} C(x, y, d(x, y)) \tag{3.11}$$

where $C(x, y, d(x, y))$ is a cost function which assigns a cost to each point in the disparity space.

The smoothness term, $E_{smooth}(d)$, is defined by the explicit smoothness assumptions made. A simple example is to look at first order smoothness of the disparity map and penalise disparities which introduce discontinuities. More recent work [134] uses second order priors for the smoothness term – a more realistic model of the world, which contains curved surfaces as well as flat walls. An example first order smoothness term in a one dimensional disparity image ($\mathcal{I}_D$), only considering

neighbouring pixels is

$$E_{smooth}(d) = \sum_x |\mathcal{I}_D(x) - \mathcal{I}_D(x+1)| + |(\mathcal{I}_D(x) - \mathcal{I}_D(x-1))| \tag{3.12}$$

Some optimisation technique is then employed to find a minimum of the full energy equation (Eq. (3.10)). Minimising this energy is a non-trivial problem, and is in fact NP-hard meaning that approximations must be made.

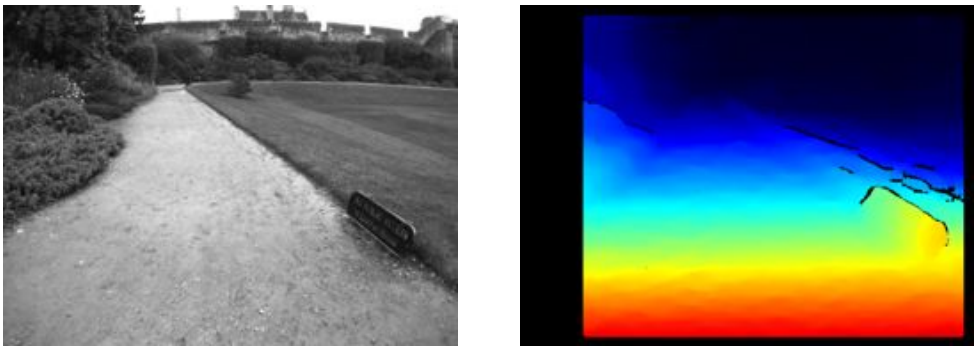
Typical modern approaches use graph-cut techniques [13, 55]. The disparity map is represented by a graph – each pixel is a vertex and there are edges between neighbouring pixels. These edges are weighted according the the E_{smooth} term, so the edge between two pixels of vastly different disparities will have a high weight. Each pixel also has edges to source and terminal vertices, the importance of which will be explained in a moment.

A graph-cut is a partitioning of a graph into two distinct sets, one containing a designated source vertex and the other containing a terminal vertex. A *minimum* graph-cut is a graph-cut which minimises the sum of the edge weights crossing the cut. Finding this minimum graph-cut is an important step in stereo correspondence algorithms utilising graph-cuts, and can typically be solved in close to linear time.

Finding an approximation to the global minimum using graph-cuts is an iterative process. One of the common algorithms employs the notion of α-expansion [55]. α refers to a particular disparity value. An α-expansion is defined as altering the current disparity map such that all pixels either retain their existing disparity values or are updated to have the disparity value α.

The idea is that at each stage we find the α-expansion which will give us a new graph with the lowest local energy, calculated efficiently using graph-cuts. The algorithm iterates, performing α-expansion at each step until no α-expansion can reduce the energy.

The results of graph-cut algorithms tends to be excellent – evidence can be found in the Middlebury 'league table' of stereo correspondence algorithms (see Section 3.2.6.2). However they are computationally far more expensive than local window based methods, taking anything from 30 seconds to many hours to calculate a disparity map for a typical image pair.



(a) left image **(b)** disparity map

Figure 3.8: Disparity map. Example result of stereo processing. (a) is the left image as captured by the camera (the right image is not shown). (b) is the resulting disparity map – the result of stereo processing. Pixels for which no disparity could be calculated are black. The colour of pixels indicate distance from camera – red pixels are close to the camera blue are further away.

3.2.3 Dense Stereo Implementation

This section discusses the implementation details of our dense stereo algorithm. It is a local window based method which uses multiple supporting windows and a number of refinement and error-checking steps.

We did not wish to be tied to proprietary commercial software such as the Point Grey Triclops SDK [99] or the SRI Small Vision Systems [119]. At the time of implementation the development of the Open Computer Vision library (OpenCV) [93] had stalled, and as we show in Section 3.2.6 our implementation outperformed all these systems.

3.2.3.1 Assumptions

A number of assumptions are made when applying a local matching method to the stereo problem:

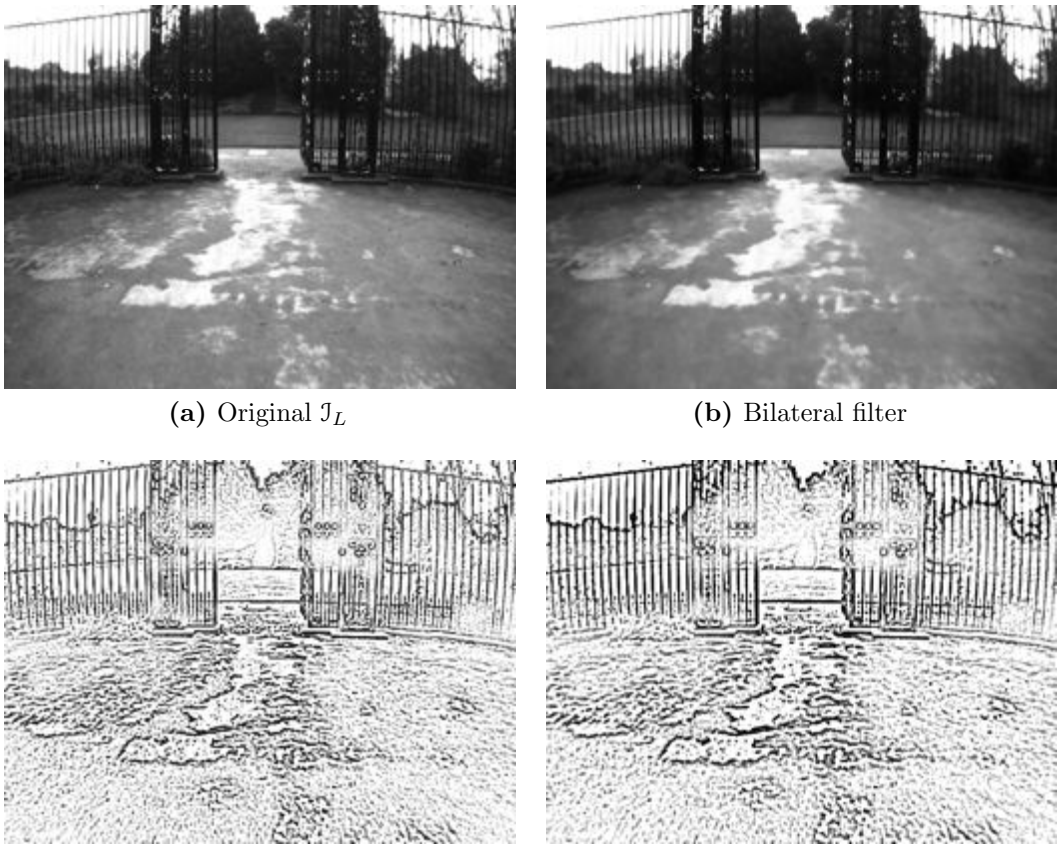
Lambertian surfaces A Lambertian surface exhibits uniform reflection of light in all directions. This is not true of most real world objects which fall somewhere between the two extremes of perfectly mirrored surfaces (specular,

non-Lambertian), and completely matte surfaces (Lambertian). We assume that all surfaces are Lambertian – a common assumption in stereo correspondence [31, 128] – and this is justified by the fact that the two lenses in the stereo camera are spatially close to each other, making non-Lambertian reflectance negligible in most cases.

Constant disparity in window Inside the search window we assume a disparity gradient of 0. This introduces errors at discontinuities (see Section 3.2.6.2 for a quantitative evaluation), which is partly compensated for by the multiple supporting windows technique, described in Section 3.2.5.

Points in both images A safe assumption to make is that most points seen in the left image will also be seen in the right image. This will be true unless an object is extremely close to the stereo pair.

1D epipolar line search We assume that the rectification described in Section 3.2.1.1 is accurate enough so that our correspondence search is reduced from a 2D search to a 1D search along image scanlines.



(a) Original $\mathcal{I}_L$ **(b)** Bilateral filter

(c) Laplacian of Gaussian **(d)** Difference of Gaussian

Figure 3.9: Preprocessing filters. The effects of various preprocessing filters are shown. Note how the bilateral filter successfully smooths out noise while retaining sharp edges (this can be seen in the puddles of water). The Laplacian of Gaussian and difference of Gaussian images are visibly very similar.

3.2.4 Image Preprocessing

As previously discussed, no two cameras are identical, even in the highest quality stereo rig. In addition to the lens distortion and misalignment of the focal planes, two cameras viewing the same scene will inevitably display some photometric variation – i.e. a surface visible in both frames may be recorded with slightly different intensity values in the two images. To eliminate this effect, the two images $\mathcal{I}_L$ and $\mathcal{I}_R$ must be processed, filtered in some way, before the stereo matching algorithm is run. This filtering step can also remove unwanted high-frequency noise.

Filtering an image involves, in the most general sense, calculating a new value

for each pixel (which is a single intensity value in the monochromatic case) as a function of one or more pixels in the original image. We will only consider filtering in the image domain, although techniques for filtering in the frequency domain exist and can be more efficient when dealing with large filter sizes (convolution is a computationally expensive process and in the frequency domain it is replaced with a cheaper multiplication).

We begin with some definitions:

- $\mathcal{I}_{in}$ – the input image: dimensions are x columns by y rows
- $\mathcal{I}_{out}$ – the output image (filtered): dimensions are x columns by y rows
- $\mathcal{H}$ – the filter kernel or support window: dimensions are w by h pixels

The filtering process uses the discrete form of the two dimensional convolution operator

$$\mathcal{I}_{out}(x,y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} \mathcal{I}_{in}(x-i, y-j)\, \mathcal{H}(i,j) \tag{3.13}$$

However the filter support window is typically much smaller than the linear dimensions of $\mathcal{I}_{in}$, and so the summation limits change to reflect this. Note a square support window is typically used, hence $k_{rows} = k_{cols} \equiv k$.

$$\mathcal{I}_{out}(x,y) = \sum_{i=-(w-1)/2}^{(w-1)/2} \sum_{j=-(h-1)/2}^{(h-1)/2} \mathcal{I}_{in}(x-i, y-j)\, \mathcal{H}(i,j) \tag{3.14}$$

3.2.4.1 Bilateral Filter

The bilateral filter is an image filtering technique proposed by Tomasi and Manduchi [125] for edge preserving smoothing. Domain filtering, as employed by traditional filters, determines pixel weighting based on spatial distance from the filter

kernel centre. For example a Gaussian blur has its maximum value at the centre kernel pixel and the coefficients decay with distance. The bilateral filter introduces range filtering – taking into account both distance from kernel centre and pixel similarity.

The bilateral filter is the pre-processing approach used by [4], in which it is shown to be effective at suppressing photometric variation between cameras and maintaining high fidelity of data at discontinuities. We implemented the standard, Gaussian case of the bilateral filter – that is both the distance function and the similarity function are Gaussian functions.

A clear description of the bilateral filter is given by [47]. First, we define the function $f : \mathbb{R}^2 \to \mathbb{R}$ to be the brightness, or intensity function which maps from a pixel's coordinates to an intensity value. Considering a pixel a_0 which is at the centre of an $n \times n$ patch, the range (similarity) function is as follows

$$r(a_0, a_i) = e^{-\frac{[f(a_i)-f(a_0)]^2}{2\sigma_r^2}} \tag{3.15}$$

The domain (Euclidean distance) function is as follows

$$d(a_0, a_i) = e^{-\frac{|a_i-a_0|}{2\sigma_d}} \tag{3.16}$$

Combining these, the value of a_0 in the filtered image is

$$\mathcal{I}(a_0) = \frac{\sum_{i=0}^{n-1} f(a_i) \times d(a_0, a_i) \times r(a_0, a_i)}{\sum_{i=0}^{n-1} d(a_0, a_i) \times r(a_0, a_i)} \tag{3.17}$$

As can be seen in Eq. (3.17) the bilateral filter has two parameters, namely the two standard deviations σ_r, σ_d. Increasing σ_d increases the blurring of the image, and increasing σ_r decreases the sensitivity of the filter to changes in intensity. Typical values used are $\sigma_d = 10$ and $\sigma_r = 5$, following the results of Ansar [4].

3.2.4.2 Laplacian of Gaussian Filter

The Laplacian of Gaussian filter is, in this context, the Laplacian operator, ∇^2, applied to a two dimensional Gaussian. Introduced by Marr and Hildreth in 1980 [67] for detecting intensity changes in an image, it is a commonly used filter for stereo pre-processing [44, 129].

We start with a two dimensional Gaussian

$$f(x,y) = \frac{1}{(2\pi)^{\frac{N}{2}}|\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(x^T\Sigma^{-1}x)} \tag{3.18}$$

$$= \frac{1}{(2\pi)^{\frac{N}{2}}\sigma^2} e^{-\frac{1}{2\sigma^2}(x^2+y^2)} \tag{3.19}$$

The Laplacian operator is then applied to give the filter kernel, $\mathcal{H}_L$

$$\mathcal{H}_L = \nabla^2 f(x,y) \tag{3.20}$$

$$= \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} \tag{3.21}$$

$$\frac{\partial^2 f(x,y)}{\partial x^2} = -\frac{1}{2\pi\sigma^4} e^{-\frac{1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)} + \frac{x^2}{2\pi\sigma^6} e^{-\frac{1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)} \tag{3.22}$$

Similarly

$$\frac{\partial^2 f(x,y)}{\partial y^2} = -\frac{1}{2\pi\sigma^4} e^{-\frac{1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)} + \frac{y^2}{2\pi\sigma^6} e^{-\frac{1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)} \tag{3.23}$$

Substituting back into Eq. (3.21)

$$\mathcal{H}_L = \nabla^2 f(x,y) = \left(\frac{x^2+y^2}{2\pi\sigma^6} - \frac{1}{\pi\sigma^4}\right) e^{-\frac{1}{2}\left(\frac{x^2+y^2}{\sigma^2}\right)} \tag{3.24}$$

3.2.4.3 Choice of Pre-processing Filter

Our chosen filter is an approximation to the Laplacian of Gaussian, namely a Difference of Gaussians (DoG) [4]. Although only an approximation to the LoG,

the DoG has a big advantage in that it is separable – that is, the convolution can be split into two passes over the image, vertical and horizontal. This reduces the computational complexity from $O(\mathcal{I}_{width} \times \mathcal{I}_{height} \times w \times h)$ to $O(\mathcal{I}_{width} \times \mathcal{I}_{height} \times (w+h))$, where w and h are the dimensions of the filter kernel.

Although the bilateral filter copes well with discontinuities at object boundaries, it is not a separable filter and therefore cannot be implemented as efficiently as a DoG filter. Additionally the qualitative and quantitative results shown in Section 3.2.6 demonstrate that the additional computation required for the bilateral filter is not justified by the marginal improvements in disparity map quality.

As the name suggests, the difference of Gaussians requires two Gaussians with different σ values to be convolved with two copies of the image, and then for one of these copies to be subtracted from the other. The best approximation to the Laplacian of Gaussian occurs when the two sigmas are in the approximate ratio $\frac{\sigma_{large}}{\sigma_{small}} \approx 1.6$ [12].

$$\mathcal{H}_D(x, y, \sigma) = (G(x, y, \sigma_{large}) - G(x, y, \sigma_{small})) * \mathcal{I}(x, y) \tag{3.25}$$

$$= G(x, y, \sigma_{large}) * \mathcal{I}(x, y) - G(x, y, \sigma_{small}) * \mathcal{I}(x, y) \tag{3.26}$$

where $G(x, y, \sigma)$ is a 2D Gaussian and $*$ is the convolution operator.

We therefore make our DoG filter by taking two copies of the image, convolving each with a Gaussian filter, and subtracting one from the other. Figure 3.9 shows the accuracy of the DoG approximation to the LoG filter. Disparity maps resulting from the two approaches show minimal differences.

3.2.5 The SAD5 matching algorithm

As discussed in Section 3.2.2.2, there are a number of different similarity measures for regions of pixels. We choose to use sum of absolute differences (SAD). SAD of two N by N patches requires N^2 subtractions and absolute value checks, compared to the more costly N^2 subtractions and N^2 multiplications needed for sum of square differences (SSD), or $2N^2$ subtractions, $4N^2$ multiplications, and a square root required by normalised cross correlation (NCC). Additionally SAD is less sensitive to outliers than SSD and NCC [79].

Using the left image, $\mathcal{I}_L$, as the reference image, a correlation curve is generated for each pixel, as shown in Fig. 3.7. For each possible disparity, d, a correlation score is calculated using sum of absolute differences (SAD) over the correlation window (typically 11×11 pixels):

$$\sum_{row} \sum_{col} |\mathcal{I}_L(row, col) - \mathcal{I}_R(row, col + d)| \tag{3.27}$$

3.2.5.1 Multiple Windows

These correlation scores are refined using the multiple supporting windows technique described in [44], which is now summarised. To reduce discontinuity errors, we would like to only take into account parts of the correlation window which do not introduce errors. We do this by combining the correlation score of the window centred on the pixel with a subset of scores from surrounding windows. Figure 3.10 shows 4 different configurations – (a) is the standard one window approach, and (b) (c) and (d) show 5, 9, and 25 window configurations.

As suggested by [44], 5 supporting windows are used for speed of computation, shown in Fig. 3.10(b). The best (lowest) 2 supporting window scores, C_{1i_1} and C_{1i_2}

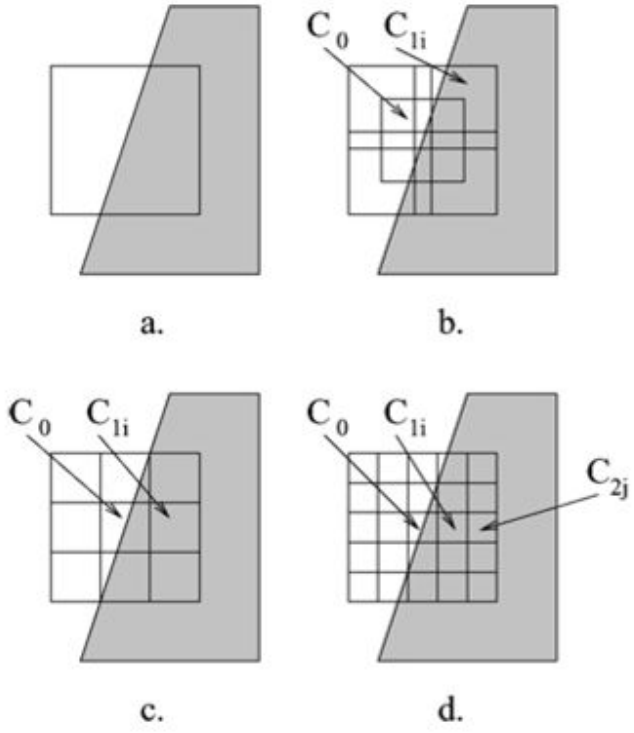


Figure 3.10: Multiple supporting windows. Different configurations of multiple supporting windows (from [44]). (a) shows one window overlapping a discontinuity. This necessarily introduces errors into the correlation matching, and so we try to compensate for this by selecting the best subset of smaller subwindows. We choose to use configuration (b), with 5 windows. The central window score and the two best scores from the other windows give us our total correlation score.

are added to the score of the middle window, C_0.

$$C = C_0 + C_{1i_1} + C_{1i_2} \tag{3.28}$$

For each pixel a search of the corresponding correlation curve is performed, looking for the minimum correlation score (SAD results in a score of 0 for perfectly matching windows).

3.2.5.2 Consistency

A left/right consistency check, as proposed by Fua [34], performs the correlation search twice by reversing the roles of the two images and marks a disparity as invalid if the correlation curve minimum is at a different position in the search $\mathcal{I}_L \rightarrow \mathcal{I}_R$ and $\mathcal{I}_R \rightarrow \mathcal{I}_L$.

3.2.5.3 Uniqueness

The uniqueness of the correlation curve minimum is used as another validation technique. Consider the example correlation curve in Fig. 3.7. A flat, or close to flat, correlation curve indicates low texture, whereas a well defined, solitary minimum strongly suggests a solid match. Following Hirschmüller [44], we take C_1 as the minimum correlation value, and C_2 as the second lowest (with the proviso that C_1 and C_2 are not direct neighbours) We use the relative difference shown in Eq. (3.29) to invalidate disparities which fall below an empirically determined threshold.

$$C_d = \frac{C_2 - C_1}{C_1} \tag{3.29}$$

3.2.5.4 Final Refinements

Subpixel interpolation is performed by fitting a parabola to the correlation minimum and the two neighbouring values. The minimum of this parabola is taken to be the subpixel disparity.

Finally we look at the 8-way connected components of each pixel in the resulting disparity map, and discard pixels which are not connected to a minimum number of pixels with similar disparities. This final step helps to remove isolated incorrect pixels.

3.2.6 Results

To measure the performance of our SAD5 implementation we ran two sets of tests. Firstly we performed a qualitative comparison between the output of our software and various existing stereo implementations. Secondly we performed a quantitative analysis using an online evaluation framework.

3.2.6.1 Qualitative Analysis

To qualitatively judge the performance of our implementation we ran our test images through the following processing pipelines:

SAD5 + DoG Our implementation of the SAD5 + DoG algorithm

SAD5 + bilateral Our implementation of the SAD5 + bilateral algorithm

Triclops 3.2-r14b Stereo software from Point Grey Research Inc.

SVS 44e Small Vision Systems software from SRI International

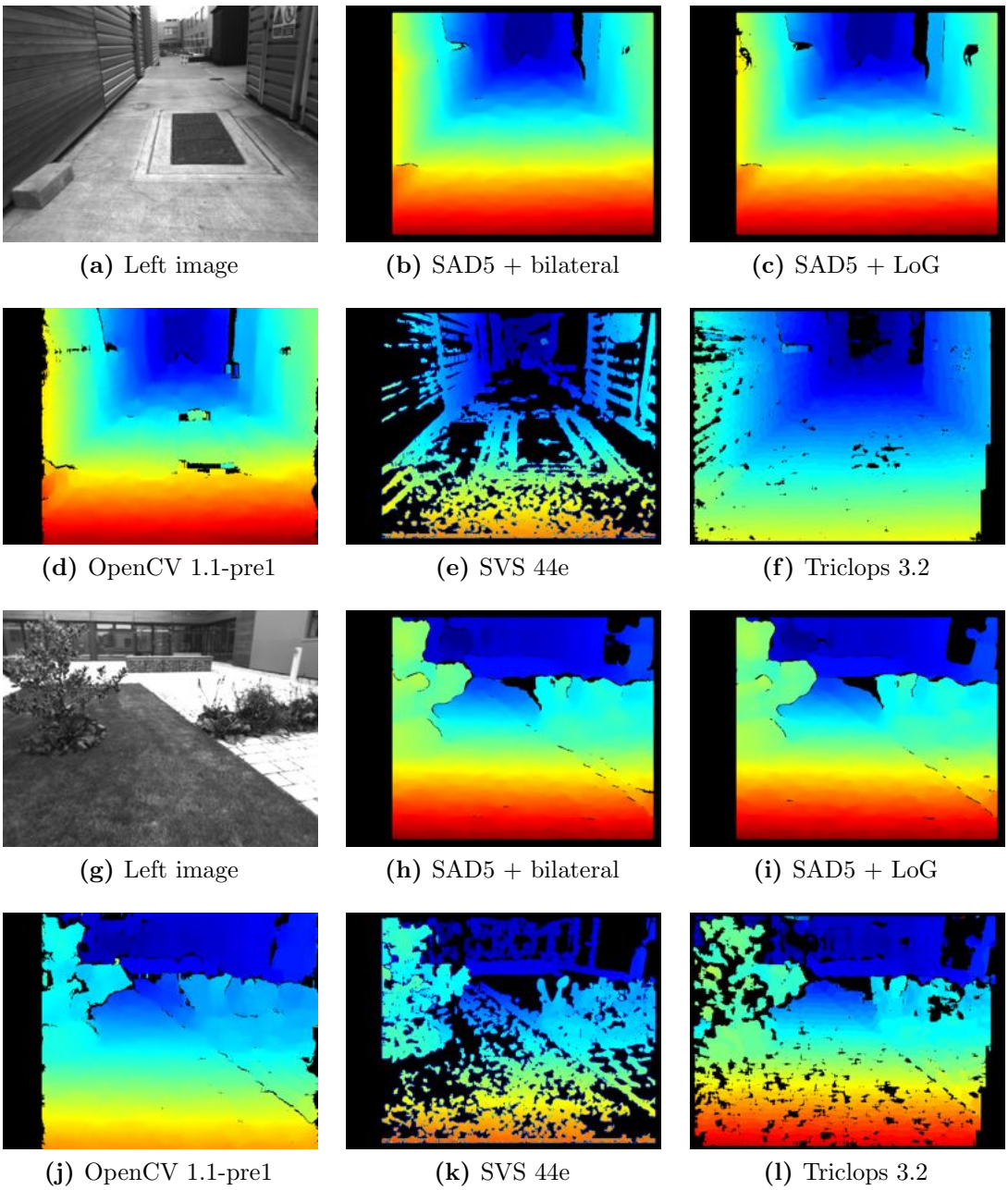
OpenCV 1.1-pre1 Computer vision library, supported by Willow Garage

Each piece of software has a wide range of tunable parameters, and thus can be fine tuned to perform well for specific applications. In our case we left the parameter settings at their default values, reasoning that these must be reasonable values that could be expected to work over a range of typical outdoor scenes, such as the ones used in this test. To make comparisons more meaningful, the correlation window sizes and disparity search ranges for each algorithm were set to be identical: a window of size 7×7 and a disparity search range of 64 pixels.

Six stereo image pairs were used, covering a range of typical environments encountered by a mobile robot, and we show typical results from two of these. Figure 3.11(a) shows an alleyway with concrete floor and wooden walls, and Fig. 3.11(b)-(f) show

the results of applying the various algorithms to the input image. Figure 3.11(g)-(l) shows the same processing applied to a pair of images of an outdoor scene with grass, bushes, and paving tiles.

A visual inspection of the results suggest that SVS and Triclops give us results that are not necessarily wrong, but definitely sparser than the results of OpenCV or our implementation of SAD5. The OpenCV results are promising, but the blocky artefacts are worrying. Our results suggest that the differences in preprocessing images with a bilateral filter as opposed to a Laplacian of Gaussian filter are not significant enough to justify the extra processing required.



(a) Left image **(b)** SAD5 + bilateral **(c)** SAD5 + LoG

(d) OpenCV 1.1-pre1 **(e)** SVS 44e **(f)** Triclops 3.2

(g) Left image **(h)** SAD5 + bilateral **(i)** SAD5 + LoG

(j) OpenCV 1.1-pre1 **(k)** SVS 44e **(l)** Triclops 3.2

Figure 3.11: Qualitative disparity map comparison. Typical results from running five different stereo algorithms on input image pairs. (a) and (g) show the left input images (right images not shown), while (b)-(f) and (h)-(l) show the result of processing by each of the different algorithms. Note that the SAD5 and OpenCV results are much denser than both SVS and Triclops, and that OpenCV produces some unwanted blocky artefacts – especially visible in (d). Colour maps to depth, where blue is more distant, red is close to the camera. The colour maps in different algorithms do not align exactly due to different internal representations of depth values.

	Non-occluded	Near discontinuities	All
Tsukuba	6.7%	24.9%	7.8%
Venus	3.0%	25.0%	3.7%
Teddy	14.7%	33.2%	21.8%
Cones	8.8%	23.7%	16.0%

Table 3.1: Results of SAD5 + bilateral filter algorithm applied to Middlebury dataset. The percentage scores are *incorrectly* assigned disparity values. Non-occluded pixels are those pixels visible in both $\mathcal{I}_L$ and $\mathcal{I}_R$. Pixels near discontinuities are much harder to match – correlation windows overlapping discontinuities introduce errors. The final column lists total incorrectly assigned pixels for the entire image

3.2.6.2 Quantitative Analysis

The Middlebury Vision group provides a set of stereo pairs with ground truth disparity maps [72]. Their website allows a user to upload their own disparity maps for each stereo pair and the differences between these and the ground truth disparity maps are measured quantitatively. A number of statistics are given – such as total percentage of incorrect disparities and percentage of incorrect disparities near occlusions (an incorrect disparity is defined as being more than ± 1 disparity away from the true disparity). The submitted disparity maps are then ranked according to the the average percentage of bad pixels across all the images, and the submission is displayed in a 'league table' of stereo algorithms.

The left image of each Middlebury stereo pair, the corresponding ground truth disparity map, and our calculated disparity map are shown in Fig. 3.12. The disparity ranges in our results have been scaled to match the Middlebury disparity maps, and any gaps in our disparity maps are filled in using a simple area filling algorithm as suggested on the Middlebury website.

Table 3.1 and Table 3.2 show the quantitative results of our stereo implementation. These numbers place us in the Middlebury evaluation table at a similar position to other local window based stereo algorithms such as Scharstein and Szeliski's Sum Squared Differences with shiftable windows [107]. Their algorithm performs slightly

	Non-occluded	Near discontinuities	All
Tsukuba	6.9%	26.9%	8.36%
Venus	6.0%	24.8%	6.82%
Teddy	20.7%	29.0%	27.6%
Cones	19.8%	27.5%	28.5%

Table 3.2: Results of SAD5 + DoG filter algorithm applied to Middlebury dataset. The percentage scores are *incorrectly* assigned disparity values. Non-occluded pixels are those pixels visible in both $\mathcal{I}_L$ and $\mathcal{I}_R$. Pixels near discontinuities are much harder to match – correlation windows overlapping discontinuities introduce errors. The final column lists total incorrectly assigned pixels for the entire image.

better on some of the images (5.2% bad pixels on Venus for example), and slightly worse than ours on others (24.8% bad pixels on Teddy for example).

Global stereo methods give better results as expected (Section 3.2.2.3) – at the time of writing the best performing algorithm was a belief propagation method by Klaus et al. [54]. This gives total bad pixels as 1.37% for Tsukuba, 0.21% for Venus, 7.06% for Teddy, and 7.92% for Cones. Although impressive, the downside is processing time – this particular algorithm takes up to 25 seconds to process one 450x375 stereo pair. This is obviously not an acceptable framerate for a realtime robotics application.

(a) Tsukuba left image

(b) Tsukuba ground truth

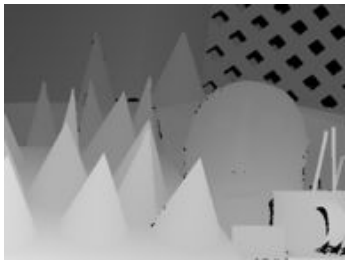
(c) MRG disparity map

(d) Venus left image

(e) Venus ground truth

(f) MRG disparity map

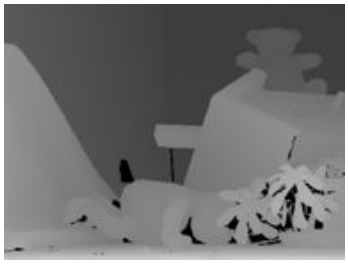
(g) Cones left image



(h) Cones ground truth

(i) MRG disparity map

(j) Teddy left image



(k) Teddy ground truth

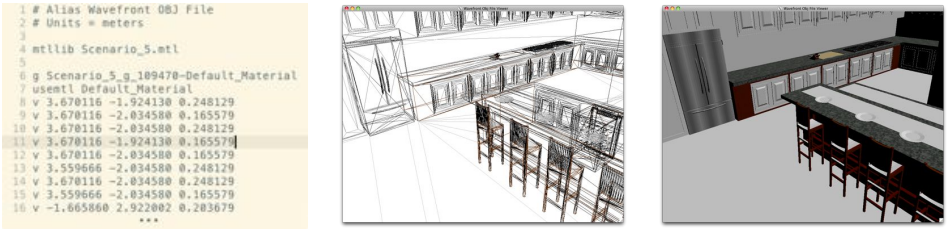
(l) MRG disparity map

Figure 3.12: Quantitative disparity map comparison. Left hand images of the image pairs used in the Middlebury online evaluation, the corresponding ground truth disparity maps, and the results of our stereo implementation

3.2.7 Conclusions

We have described the design and implementation of a local window-based dense stereo processing algorithm and the preprocessing and camera calibration required. We have demonstrated qualitative and quantitative results which suggest that our results are as good as, and in many cases better, than existing and available real-time stereo implementations.

This competency in acquiring 3D data at framerate from a high resolution stereo camera is a fundamental part of the exploration process. Integrating this data into a 3D map structure such as the occupancy grid maps described in Chapter 2 or the surface graph structure which will be described in Chapter 5 we are then in a position where we can look at our world model from a higher level – that of a cohesive map rather than a unconnected collection of points – and start to think about choosing targets for exploration.



(a) **(b)** **(c)**

Figure 3.13: OBJ file loading. (a) is a snippet of an OBJ file specifying a kitchen interior model. (b) shows the wireframe version of this model after the OBJ file has been parsed and loaded. (c) is the final textured model which is used in the simulator.

3.3 Simulated Depth Sensor

For the purposes of quickly testing and evaluating exploration algorithms a simple 3D depth simulator was created. The idea was to create a drop-in replacement for a physical stereo camera – the abstraction and modularisation in our software framework described in Chapter 4 means that the exploration code is completely separate from the sensor drivers.

To display 3D graphics we use OpenGL [94], a cross-platform, cross-language library for 2D and 3D visualisation. OpenGL provides commands for drawing various primitives such as vertices, lines, and polygons. It also provides callback mechanisms for dealing with user input (keyboard and mouse) and allows quick creation of 3D user interfaces.

3.3.1 Loading 3D Models

We store 3D models as OBJ files. This is an open OBJ file format which is almost universally accepted by 3D modelling programs. An OBJ file is a plaintext file which specifies model parameters as lists of coordinates – vertices, normals, textures, colours. Listing 1 shows the basics of the file format.

An OBJ file is read by our simulator and parsed using the open-source GLM

```
# list of vertices with [x y z] coordinates
v 1.55 0.40 0.91
v ...
...
# list of texture coordinates with [u,v] coordinates
vt 1.21 -0.34
vt ...
...
# list of surface normals in [x,y,z] form.
vn 0.0 0.7 0.7
vn ...
...
```

Listing 1: The OBJ file format

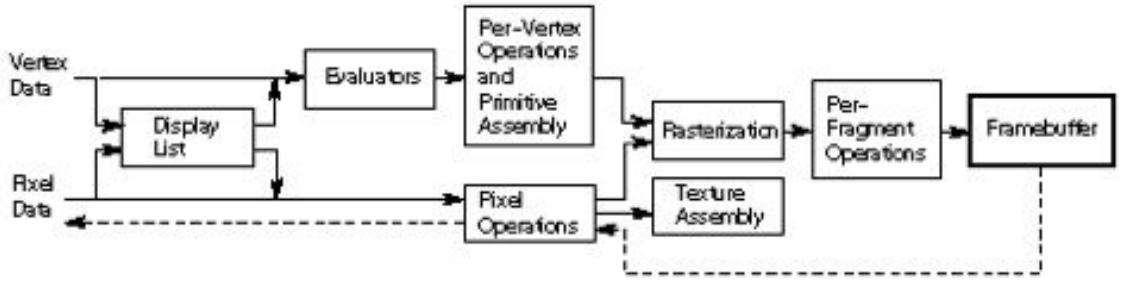


Figure 3.14: OpenGL pipeline. The OpenGL rendering pipeline, ending in a framebuffer. The default framebuffer is visible to the user in an OpenGL window, but we also make use of a *framebuffer object* which is an offscreen render target allowing us to simulate a depth sensor with it's own viewpoint.

library [37]. It can then be displayed in an OpenGL context by calling the `model->glmDraw()` method. Figure 3.13 shows an example file snippet, followed by the wireframe model displayed in an OpenGL window, and finally the textured model.

3.3.2 OpenGL Framebuffer Objects

The OpenGL rendering pipeline is shown in Fig. 3.14. It begins with geometric data (lines, polygons, etc.) and pixel data (pixels, images, etc.) and ends with rasterisation into a *framebuffer* (taking the current camera position, clipping planes, lighting, etc. into account). A framebuffer is a memory buffer which stores data corresponding to a single frame of rendered data – the default framebuffer is processed by the graphics

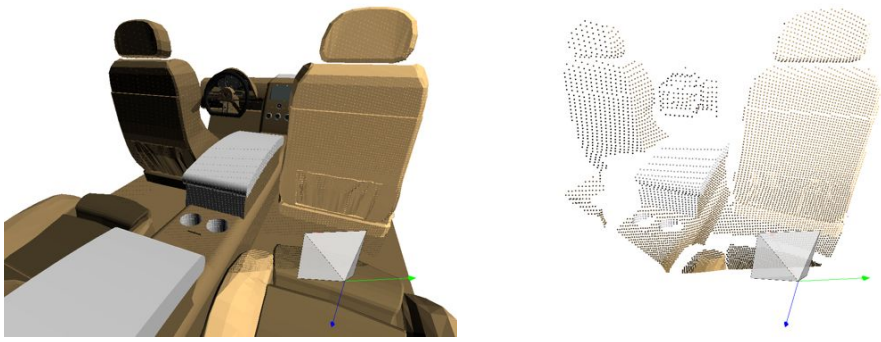


Figure 3.15: Simulated depth sensor. Screenshots from the OpenGL simulator. The camera in the bottom right views the car interior model and returns a cloud of 3D points, analogous to a stereo camera.

card and displayed on screen. The framebuffer stores not only RGB data, but also normalised depth values for each pixel.

A *framebuffer object* (FBO) is an extension to OpenGL which allows the creation of a second framebuffer which is not rendered to screen. Rendering to this FBO from the viewpoint of a simulated depth sensor *and* rendering to the default framebuffer from a second OpenGL camera viewpoint allows us to have a virtual camera with a pose which is independent of the main OpenGL camera's view. In pseudocode, the main rendering loop is as follows

```
1 OpenGLWindow::Render()
2 {
3     SetViewpoint( VirtualSensor )
4     RenderScene( FBO )
5
6     SetViewpoint( OpenGLCamera )
7     RenderScene( Display )
8 }
```

3.3.3 Results

The view from the virtual depth sensor is stored in the framebuffer object as an RGB image and corresponding depth values per pixel. We can read these back from

the GPU and with knowledge of the virtual sensor properties (focal length, field of view, etc.) we can project them as 3D points in the frame of the virtual sensor. The pose in $\mathbb{R}^3$ of the virtual sensor is known and the point cloud can therefore be transformed to be in the global OpenGL coordinate frame. We can then publish the $[x, y, z]^T$ points to be consumed by the software framework described in Chapter 4.

Figure 3.15 shows the system in action, with the user's view in (a) and the point cloud as seen by the virtual depth sensor shown in (b).

Chapter 4

Systems

4.1 Introduction

This chapter is about the software and hardware systems used in the Mobile Robotics Group. In it we will describe our robotic platform from the ground up: hardware design, communication protocols, middleware design and structure, and finally the engineering that has gone into our top-level software.

A thorough reading of this chapter is not strictly necessary for the chapters that follow, but familiarity with the systems design will of course aid understanding.

4.1.1 Layers of Abstraction

To begin, we describe each layer as shown in Fig. 4.1.

4.1.1.1 Hardware and Low-Level Drivers

At the lowest level there is the hardware: the motors which drive the wheels, the wiring which carries power and data around the machine, the sensor payload and mountings, and the batteries which keep it running.

Sensors communicate using various protocols and over different hardware inter-

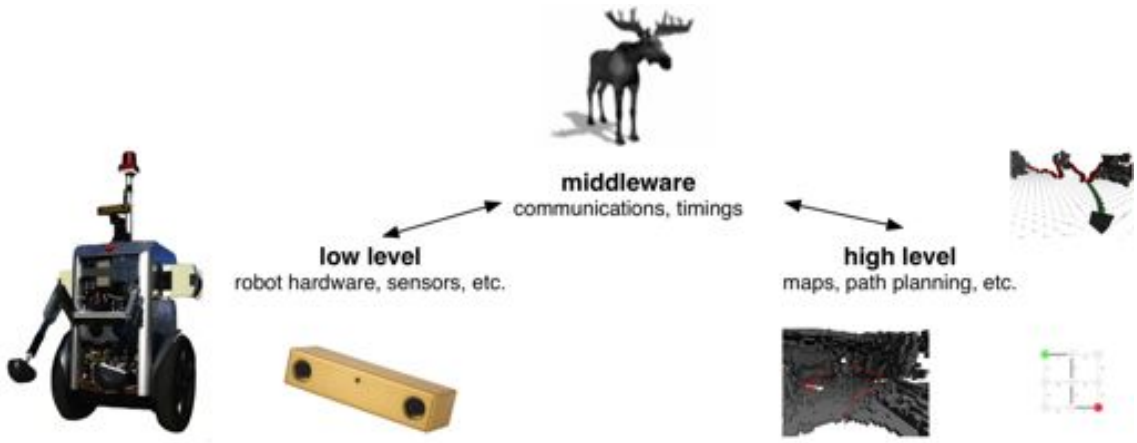


Figure 4.1: Software architecture layers. Software systems in a robot system can be roughly classified into three categories: Low level code which interacts directly with hardware, middleware which facilitates communications and provides an opaque abstraction layer, and finally the high level algorithms which perform exploration, object recognition etc.

faces. The data coming from these sensors must be translated into a common format to allow it to be consumed by other processes, and this is done at the interface between low-level drivers and the middleware.

4.1.1.2 Middleware

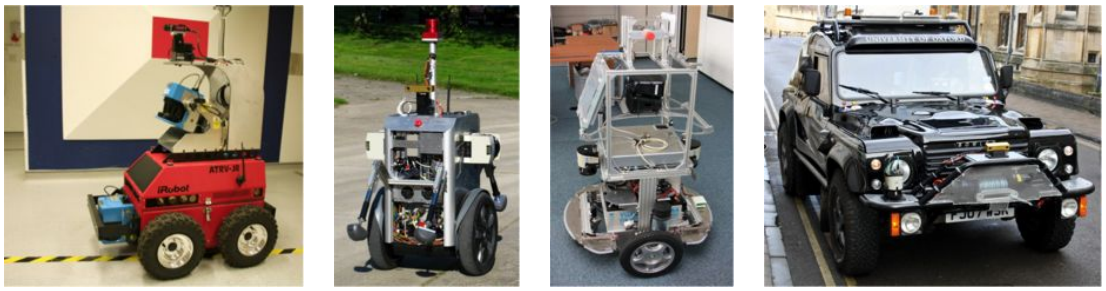
Middleware is the software which acts as a conduit between the top-level software and the hardware. It acts as an opaque layer which insulates high level algorithms from hardware implementation details (an exploration program which decides that the robot should move from pose A to pose B shouldn't have to worry about what voltages to apply to the wheel motors).

The middleware facilitates communication between processes and is therefore a vital component in the software stack.

4.1.1.3 High-level Algorithms

The top level consists of the algorithms which ultimately control the behaviour of the machine. It is here we begin to think about abstractions such as grid maps

and path planning over graphs. Ultimately this is where the majority of interesting research takes place, with the lower levels supporting and facilitating these high level implementations. Chapter 5 and Chapter 6 describe in detail the design and implementation of two exploration algorithms which fall into this category.



(a) Marge **(b)** Lisa **(c)** Europa **(d)** Wildcat

Figure 4.2: MRG robots. The fleet of robots owned and operated in the Mobile Robotics Group. From oldest on the left, Marge in (a), to the latest acquisition: the autonomous Wildcat in (d).

4.2 Hardware

The Mobile Robotics Group at Oxford maintains an expanding collection of mobile robots of varying sizes and capabilities. A selection are shown in Fig. 4.2.

Marge, Fig. 4.2a, is the oldest member of the group. She has a 2D SICK laser mounted on a nodding platform which allows 3D scans to be taken, a fixed 2D laser, and a camera on a pan-tilt unit. Once the workhorse of the group she has been superseded first by Lisa, Fig. 4.2b, and then the Europa robot, Fig. 4.2c.

Lisa is a segbot based on Segway's Robotic Mobility Platform [108]. She has SICK LMS 291 laser scanners at 90° which "sweep" through the world building 3D maps, a forward facing Point Grey Bumblebee stereo camera, and an omnidirectional Point Grey Ladybug camera on a mast. Lisa was used to capture the data published in the New College Data Set [117] and results in Chapter 5.

The Europa robot is the latest regular wheeled robot to be added to the collection, and has been the primary platform used in this thesis (see results in Chapter 6) – more details are given below.

Finally the Wildcat, Fig. 4.2d, is an autonomous car which will be used for research into lifelong learning and large scale autonomy. This is a recent addition to the group and falls outside the scope of this thesis so will not be discussed further.

4.2.1 EUROPA

The Europa robot is a robotic platform developed as part of a European Commission project. It is designed and built by Bluebotics [10], Switzerland, and identical units are in use at all collaborating universities: Oxford, Zurich, Leuven, Freiburg. Shown in Fig. 4.2c, it stands at approximately shoulder height, as the primary purpose is to be a guide robot in public areas. It will communicate with people using speakers and a microphone and through the touchscreen mounted on its back. However in this work we are only concerned with the data gathering capabilities of the platform.

4.2.1.1 Sensor payload

The robot is equipped with numerous sensors. It has 3 SICK LMS 151 laser scanners for map building and obstacle avoidance. Two of these are mounted in the horizontal plane approximately 30 cm from the ground plane; one facing forwards and one facing backwards. These are used for localizing and map building. The third laser is also forward facing but is angled downwards such that its scanning plane intersects the ground a metre or so in front of the robot. This is to detect small obstacles such as kerbs.

It has a high quality Trimble Pathfinder GPS unit for localization when outdoors, an XSens MTi inertial measurement unit, collision sensors on the front and rear bumpers, and two AVT StingRay F046C monocular cameras for wide baseline stereo and pedestrian detection.

Finally, and most importantly for this research, it has a mounting bracket 1.5m from the floor which can take a Bumblebee stereo camera or a Kinect sensor.

4.2.1.2 Controller

An embedded computer, known as the ANT rack, is installed at the base of the robot. This machine deals directly with controlling the wheel motors. An interface

to the ANT rack allows external processes to connect and send velocity and rotation commands to it. It outputs 2D pose estimates based on wheel odometry, but these are disregarded in favour of our more accurate visual and scan matching odometry estimates described in Chapter 2.

4.3 Middleware

Between low-level hardware such as laser scanners or motors, and high-level applications such as path planning or object recognition, lies the middleware. Middleware is a software layer which facilitates communication between software components or applications and acts as an opaque abstraction layer which is vital for modularising components in a robot system.

4.3.1 ROS: Robot Operating System

A recent entry to this field is the **R**obot **O**perating **S**ystem (ROS) [101]. ROS began life at the Stanford AI Lab before being adopted in 2007 by Willow Garage where it is under active development. ROS is an open-source (BSD license) project which is gathering a lot of momentum and is building a large repository of community-developed drivers and packages.

ROS is not simply middleware however – as the name implies it aims to be a complete operating system for a robot, supplying low-level drivers, a communications layer, and high-level recognition and planning applications. For a new project ROS could be a good choice, but the Mobile Robotics Group has an existing software stack including the MOOS (Section 4.3.2) middleware layer. This middleware is actively developed in house and has been field tested extensively by many groups worldwide. It was decided that the possible advantages of migrating to ROS were outweighed by the level of trust we have in our existing system and expert inside knowledge of its structure and operation have minimised many debugging sessions.

4.3.2 MOOS: Mission Oriented Operating System

The **M**ission **O**riented **O**perating **S**ystem (MOOS) [73] is cross platform middleware software. It is a collection of libraries and applications that facilitates high speed,

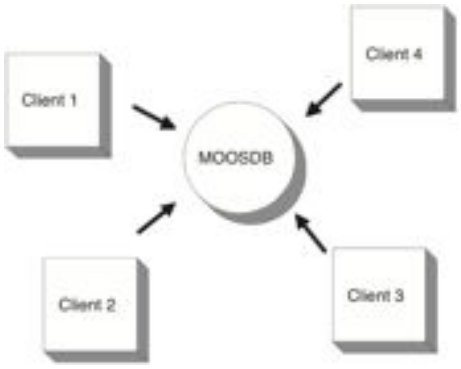


Figure 4.3: MOOS star topology. The star topology of a MOOS application. Clients communicate through the central `MOOSDB` – there is no peer-to-peer communication. Image from [85].

robust, lightweight communications between components in a robotic system. Written in C++ and with contributions from collaborators worldwide, it is freely available and open-source (GNU GPL license).

4.3.2.1 Star Topology

A star network topology is the underlying structure of all MOOS sessions. At the heart of this network lies the MOOS Database or `MOOSDB` – a lightweight hub through which all communication takes place. Figure 4.3 shows four client applications communicating with the `MOOSDB`.

The choice of a star topology brings with it advantages and disadvantages. The obvious disadvantage is the single point of failure: if the `MOOSDB` crashes or becomes isolated from the network then all communications will fail. While the `MOOSDB` is running it is susceptible to "bottle-necking" if it is swamped by data from a client.

The advantages of this design are numerous however, and the MOOS design documents [85] argue that these outweigh the disadvantages:

- **logging**: since all messages must pass through the `MOOSDB` then a central logger can record all communications during a session
- **client isolation**: a badly written or malicious client can't interfere directly with

Variable	Meaning
Name	The name of the data
String Val	Data in string format
Source	Name of client that sent this data to the MOOSDB
Time	Time at which the data was written
Message Type	Type of Message (typically notification)

Table 4.1: Typical Contents of a MOOS Message

other clients and the system performance is not dependent on the functioning of any single client

- **network traffic**: a message from client 1 to client 2 will make at most two hops – one from client 1 to the MOOSDB, one from the MOOSDB to client 2
- **scaling**: it is easy to add a new client to the network and this does not affect the complexity of the network
- **distribution**: clients can be distributed across different machines with different operating systems

4.3.2.2 Message Structure

All messages in MOOS are sent as comma separated `key=value` pairs stored in a string. For example a timestamped 3DoF pose estimate may be sent as:

```
Name=POSE_ESTIMATE, Time=1308749109, Pose=[3x1]{0.4,1.9,2.2}
```

The fields present in a typical MOOS message can be seen in Table 4.1. The practise of sending every piece of data as a string may seem surprising and inefficient but again it has a number of advantages which may not be immediately obvious

- **human readable** it is easy to inspect the contents of any data passing through the MOOSDB and this is enormously useful in debugging

- **logging and replay** strings can be appended to a log file and replayed by iterating over the file and throwing each line back at the `MOOSDB`
- **flexible** not relying on a specific binary format means that the order and content of messages can be reordered and changed without recompilation

The main benefit of using a binary format would be the reduction in network traffic with a more compressed representation. Until recently this has never been of concern, but the latest releases (SVN revision r2335 and later) of MOOS do support transmission of binary data. Full frame stereo data and 3D points from 3 $50Hz$ lasers (the sensor payload of the Europa robot described in Section 4.2.1) can all be pushed through a central `MOOSDB` using this new protocol.

4.3.2.3 Communication Protocol

A central design decision in MOOS is that clients should not need to know of the existence of other clients – there is no peer-to-peer communication, and all clients must have unique names (and they will be rejected by the `MOOSDB` if not).

Sending messages to the `MOOSDB` Each client connects to the central `MOOSDB` through an instantiation of the `MOOSCommClient` class provided by the core MOOS libraries. The `MOOSCommClient` abstracts the details of communication away from the client object – a client should not concern itself with details of network protocols or timestamping data. As shown in Fig. 4.4, this `MOOSCommClient` maintains a OutBox database which the `MOOSApp` can post messages to. Periodically, and when the `MOOSDB` is ready to accept a connection, the `MOOSCommClient` will package all the messages in this database into a single packet to send over the network to the `MOOSDB`. Once it has arrived the `MOOSDB` acknowledges receipt of the data, and retains a copy of the messages internally, ready for delivery to other clients which may need the data.

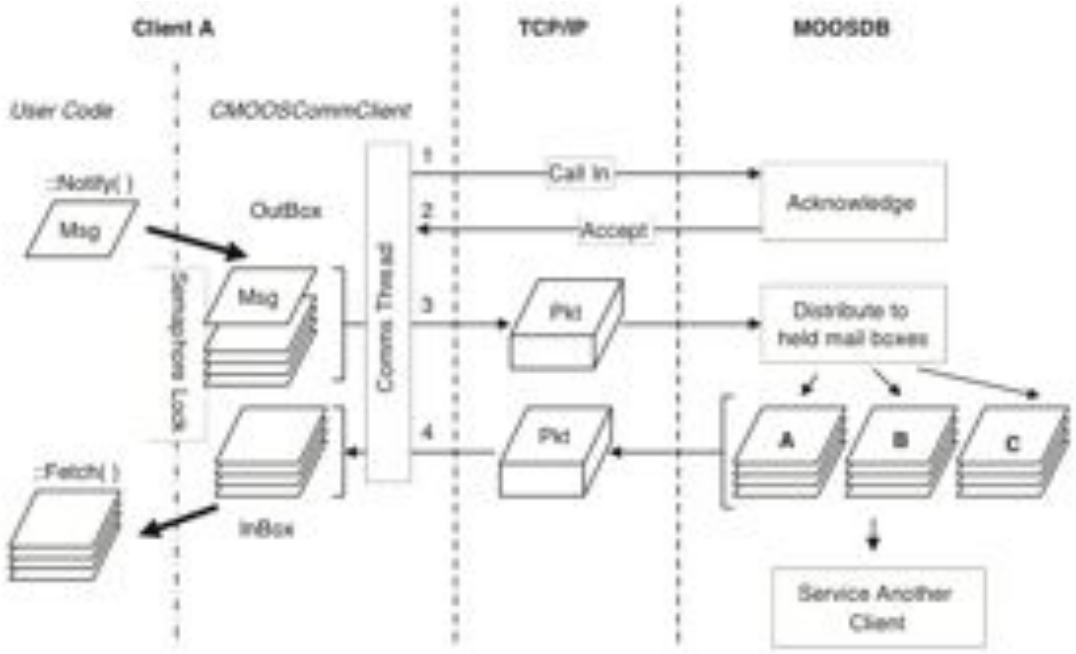


Figure 4.4: MOOS comms structure. An overview of the client server interaction in MOOS. Client A on the left sends data to the `MOOSDB` using the CMOOSCommClient interface. This maintains an OutBox database which is periodically checked by a thread which attempts to send the contents over TCP/IP to the `MOOSDB`. If the `MOOSDB` is not busy then it accepts this transmission, acknowledges receipt of the data, and distributes the messages to the mailboxes of other clients which have subscribed to the data produced by Client A. A client can call the Fetch method at any time to retrieve stored messages from the `MOOSDB`. Image from [85].

Receiving messages from the `MOOSDB` To transmit data from Client A to Client B requires some mechanism in the `MOOSDB` to store messages from Client A until such a time as Client B is ready to receive them. Client A may be publishing many different messages, and Client B may only wish to consume a fraction of these – only pose estimates for example.

As shown in Table 4.1 every message has a `Name` associated with it. On startup Client B registers its interest in messages with `Name=POSE_ESTIMATE`. Now, when Client A publishes such a message the `MOOSDB` will put a copy of the message in the mailbox of Client B. When Client B next calls the `Fetch` method it will be sent all the messages in its mailbox as one packet, ready to be unpacked on receipt.

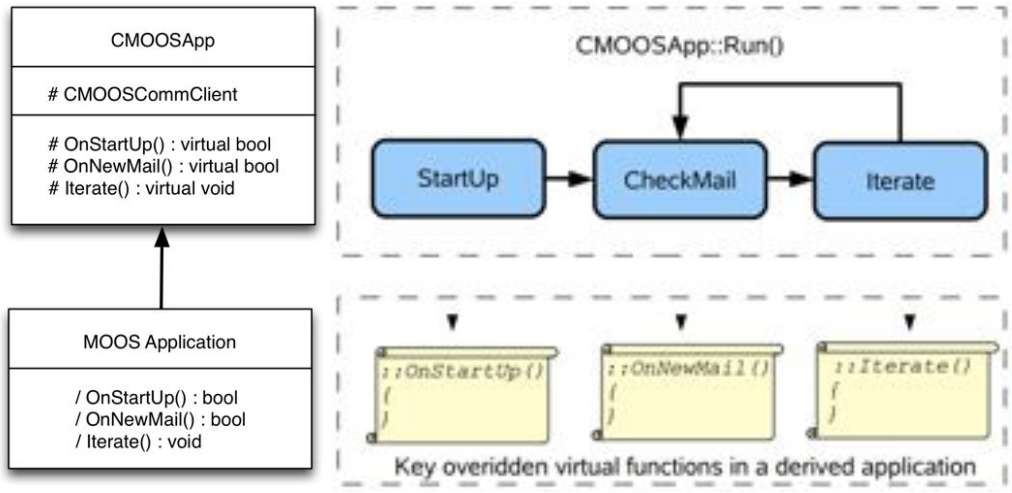


Figure 4.5: MOOS app structure. A `MOOSApp` is a class derived from the `CMOOSApp`. A derived class will typically override three key virtual methods. `OnStartUp()` initialises communication with the `MOOSDB`. `OnNewMail()` is called by the `MOOSApp` thread whenever the `MOOSDB` has new messages for the client. `Iterate()` is where the bulk of the application's work is done. Called repeatedly by the parent class thread. Image from [85]

4.3.2.4 `MOOSApp` Structure

A `MOOSApp` is a class derived from `CMOOSApp`. A derived class will typically override three key methods

- `OnStartUp()` performs initialisation and registers with the MOOSCommClient the the variable names the app wishes to subscribe to
- `OnNewMail()` is called by the `MOOSApp` thread whenever the `MOOSDB` has new messages for the client. A typical `MOOSApp` will iterate through the messages received and store or process their data in turn
- `Iterate()` is called repeatedly at a frequency which can be specified in a configuration file at runtime. This method is where an application will do whatever work the app is designed to do, executing one cycle of a control loop for example.

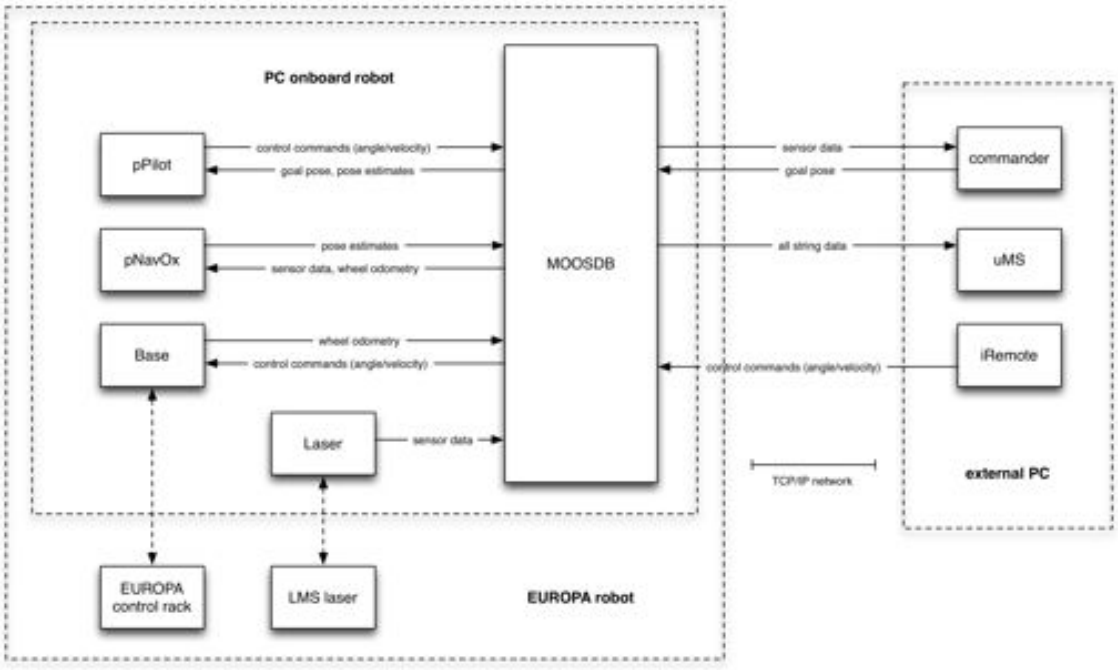


Figure 4.6: MOOS and Europa structure. A typical collection of processes on a working robot. Independent processes are shown as blocks. The central `MOOSDB` is the hub through which all inter-process communication happens. The MOOS star topology is network layout agnostic, but it is sensible to distribute processes to hardware which they directly interact with. For example the `Base` process which communicates control commands to the motor controllers in the ANT rack should run onboard the EUROPA robot itself to avoid unnecessary network traffic. Following this line of reasoning leads us to the situation pictured – the visualisation and high-level exploration algorithms are run on an external machine (right of image), while the lower level sensor drivers and controllers run onboard the EUROPA robot alongside the `MOOSDB` (dashed box on left of image).

4.4 High Level

Above the hardware and middleware comes the application layer. This is where the high level abstract problems are tackled, problems such as object recognition, environment mapping, goal selection.

This section will detail the design and implementation of the `Commander` application which was used to produce the results in Chapter 6.

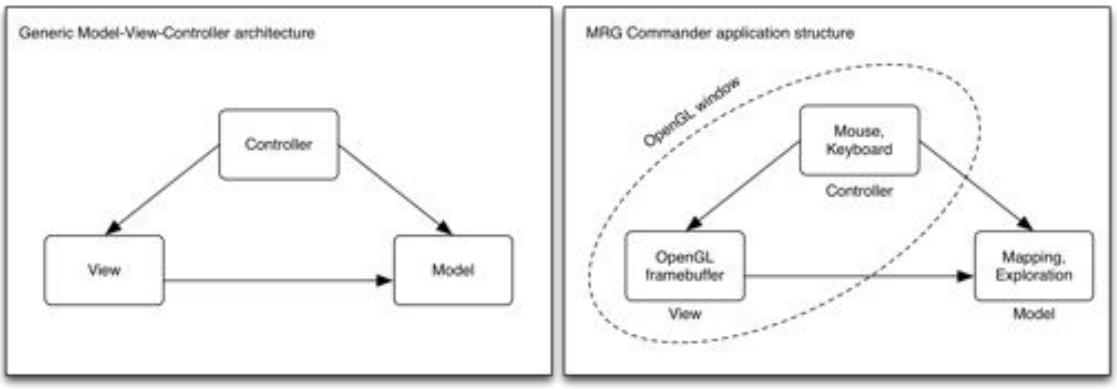


Figure 4.7: Model-view-controller architecture. The model-view-controller software architecture is a design pattern which enforces the separation of data (the `Model`) from the user interface (the `View`) and the method of interaction (the `Controller`). This results in a modular design and with well-defined interfaces means that we can add new components with relative ease. On the left is the generic model-view-control architecture, with arrows showing the interaction. On the right is the MVC structure of the `Commander` application.

4.4.1 Commander

The `Commander` is a `MOOSApp` which subscribes to 3D data (points and pose estimates) and calculates and publishes camera trajectories. It is designed as an easily extensible framework which allows new exploration or mapping algorithms to be integrated quickly, and can be configured by plain text configuration files. This section will discuss the modular structure of the `Commander` code, and the configuration file format.

4.4.1.1 Model-View-Controller Architecture

At the heart of the `Commander` lies a model-view-controller (MVC) architecture, first described in [102]. The MVC architecture consists of three parts as shown in Fig. 4.7:

- `Model` Manages the application data and processing of data. Responds to requests about its state (typically from a `View`), and requests to change state (typically from a `Controller`).

- `View` A visual representation of the `Model` for visualisation and interaction – a user interface. A web page rendered in a browser or an OpenGL window are typical examples.

- `Controller` The link between the user and the `Model` and the `View`. Takes input (key press, mouse movement, etc.) and communicates these to the model and the `View`.

This modularisation and clear separation of responsibility can result in a highly flexible system. One can imagine a situation where an OpenGL `View` of a 3D scene would be useful for debugging and testing, but which would involve unnecessary overhead and computation when batch processing a large number of test cases. If the interfaces and communication protocols in the MVC architecture are well defined, then an alternative `View` which prints `Model` state messages to a terminal could be a drop-in replacement for the OpenGL `View`. In fact, a `View` could be omitted entirely if no visualisation of the `Model` or `Controller` is needed.

4.4.1.2 Implementation: The pipeline `Model`

The `Commander` is a pipeline of operations beginning with the acquisition of 3D data and ending with a sequence of desired poses being published to `MOOSDB`. This pipeline can be queried through a well defined interface and can be considered to be the `Model` in this application. The various stages in this pipeline are shown in Fig. 4.8.

- **Sensor** Receives and processes 3D sensor data, directly from hardware or via `MOOSDB`

- **Filter** Filters 3D data by, for example, applying a distance threshold

- **Map** Integrates filtered data into existing map structure

```
<Scenario name="Simulator" type="manual" version="1">
    <Sensor type="Simulator" PoseEstimator="none">
        <MOOSHost>        localhost  </MOOSHost>
        <MOOSPort>        9000       </MOOSPort>
        <MOOSPoseVarName> SIM_SENSOR </MOOSPoseVarName>
    </Sensor>

    <Filter type="Distance">
        <Threshold> 10.0 </Threshold>
    </Filter>

    <Map type="OctoMap">
        <Res> 0.1 </Res>
    </Map>

    <Explorer type="Potential">
        <Mid> 0.0; 0.0; 0.0 </Mid>
        <Dim> 5.7; 4.3; 4.5 </Dim>
        <Res> 0.1           </Res>
    </Explorer>

    <Controller type="MOOSSim">
        <MOOSHost> localhost </MOOSHost>
        <MOOSPort> 9000      </MOOSPort>
    </Controller>
</Scenario>
```

Listing 2: Commander configuration - An example of a configuration file for the `Commander` application. This XML is parsed when `Commander` starts and appropriate object instances are created. The correspondence between these XML blocks and the structure of the `Commander` can be seen in Fig. 4.8.

- **Explorer** Identifies exploration targets and paths in the map
- **Controller** Communicates target poses to hardware, either directly or via `MOOSDB`

All five of these modules are defined as *abstract base classes*. This is object-oriented programming terminology for a class which cannot be instantiated as it has not been fully defined – it contains method declarations with no method body for example. An example in the Explorer base class is the `FindPath` method which does the work of selecting a path to an exploration target:

```
virtual void FindPath( Pose pose, Path & path ) = 0
```

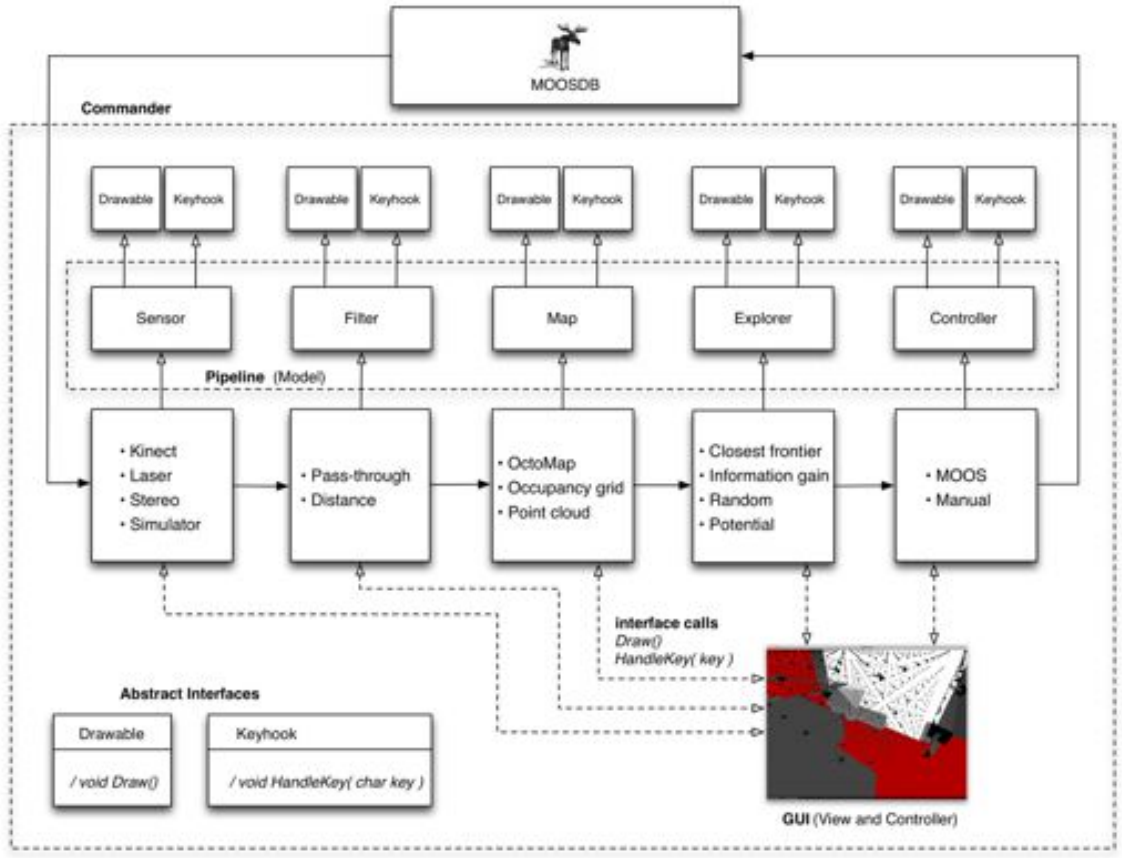


Figure 4.8: `Commander` structure. The `Commander` is a pipeline of operations beginning with the acquisition of 3D data from either a local sensor or through the `MOOSDB`. Data is filtered, integrated into a map, an exploration algorithm is run using the updated map, and the resulting trajectory is sent back to the `MOOSDB` as a sequence of desired poses. This image shows the inner structure of the Commander process shown in Fig. 4.6.

Each of the derived classes such as the Closest Frontier explorer, or the Potential explorer implement `FindPath` differently, but the common base class interface means that the derived class objects can be used interchangeably by the `Commander`.

Some concrete classes implementations are shown underneath (and inheriting from) the pipeline in Fig. 4.8. The pipeline stages and their interfaces mean that extending the code base with a new class implementation is straightforward.

The Map is the most important of these from the perspective of maintaining a model of the world but the other pipeline stages maintain their own state (such as last path selected) and can also be thought of as part of the `Model`.

4.4.1.3 Visualisation: OpenGL window as `View` and `Controller`

```
1  class Drawable
2  {
3      public:
4          Drawable(){};
5          virtual ~Drawable(){};
6          virtual void Draw() = 0;
7  };
8
9  class Keyhook
10 {
11     public:
12         Keyhook(){};
13         virtual ~Keyhook(){};
14         virtual void HandleKey( bool print, unsigned char key = 0 ) = 0;
15 };
```

Listing 3: Drawable and Keyhook abstract base class interfaces as used in the `Commander` processing pipeline

The `Commander` interface implements both the `View` and the `Controller`. This is common as there is usually a tight-coupling between these two components as both need access to the `Model` and (in an application such as ours) the user interaction takes place through the same user interface element as the visualisation.

Figure 4.8 shows each section of the pipeline inheriting from two base classes: `Drawable` and `Keyhook`, shown in Listing 3. These are simple interfaces which each define a single method: `Drawable::Draw()` and `Keyhook::HandleKey( bool print, unsigned char key )`.

The `Drawable::Draw()` method is used to draw some representation of each pipeline stage into a 3D world which the user can interact with. To do this we make use of the Open Graph Library (OpenGL), a freely available cross-platform and cross-language platform for developing 2D and 3D visualisations. OpenGL handles window creation, keyboard and mouse handling, and has function calls for drawing various primitive shapes. We have wrapped the OpenGL library into

our own `OpenGLWindow` class which acts as both the `View` and `Controller` in the `Commander` application. A screenshot of the window is shown at the bottom right of Fig. 4.8.

On startup an OpenGL application enters a main control loop which iterates continuously and calls a user-defined display callback function on each iteration. This callback is the `OpenGLWindow::Draw()` function shown in Listing 4. This function iterates over all `Drawable` objects which the window has knowledge of and calls the `Drawable::Draw()` function of each.

The visualisation and implementations of pipeline modules are therefore cleanly separated: a module implementing the `Drawable` interface can be added to an OpenGLWindow and it will be drawn, or it can be run without a GUI.

```
1  void OpenGLWindow::AddObject( Drawable &obj )
2  {
3      m_vObjects.push_back( &obj );
4  }
5
6  // OpenGL display callback
7  void OpenGLWindow::Draw()
8  {
9      std::vector<Drawable*>::iterator it = m_vObjects.begin();
10     while( it != m_vObjects.end() )
11     {
12         (*it)->Draw();
13         it++;
14     }
15 }
```

Listing 4: Two functions in the OpenGL window interface: `AddObject` allows a `Drawable` object (see Listing 3) to be added to the `View`. The display callback function `Draw()` is called on each iteration of the main OpenGL control loop and it calls the `Draw()` function of each `Drawable` object which the window knows about.

Communication in the other direction – user input being fed back to the pipeline modules – happens in a similar manner. A `Keyhook` object is added to the OpenGL window and when a key press or mouse movement occurs these objects are iterated over and have their `HandleKey` methods called. It is the responsibility of the

implementations not to use keybindings which conflict with other modules – a shortcoming which could be addressed by forcing modules to declare to the OpenGL window which keys they wish to deal with.

4.5 Concluding Remarks and Tools

This chapter has provided an overview of the various layers of software and hardware used in the Mobile Robotics Group and in this thesis. A lot of engineering effort has been invested in these systems to ensure reliable and robust operation and this chapter has given some justification and explanation of various design decisions along the way. Consider that Fig. 4.8 is an exploded diagram of a single process block – the `Commander` – to get an idea of the complexity of the full system.

4.5.1 Tools used

Implementing and managing this code base is a big task, but helped enormously by the use of high quality third-party libraries to perform various functions:

- **Subversion** Version control software which stores source code history in a central repository – vital for sharing code across a group and reverting to previous versions. `http://subversion.tigris.org/`
- **CMake** Cross platform build system which dramatically improves the experience of adding code to the existing code tree. `http://www.cmake.org/`
- **Boost** Peer-reviewed C++ libraries implementing a wide range of algorithms: concurrency primitives (e.g. scope locked mutex), timers, regular expressions, etc. `http://www.boost.org/`
- **Eigen** C++ libraries for linear algebra, providing widely used matrix and vector primitives with similar syntax to Matlab. `http://eigen.tuxfamily.org/`
- **OpenGL** Cross-platform, cross-language, 2D and 3D visualisation libraries. Used to display 3D sensor data, maps, exploration paths, and to handle user input. `http://www.opengl.org/`

Chapter 5

Graph Based Exploration of Workspaces

5.1 Introduction

This chapter is about the automated discovery and mapping of surfaces using a depth sensor, primarily a stereo pair. We begin with the observation that for any workspace which is topologically connected (i.e. does not contain free flying islands) [39] then there exists a single surface that covers the entirety of the workspace. We call this surface the covering surface. We assume that while this surface is complex and self intersecting every point on it can be imaged from a suitable camera pose and furthermore that it is locally smooth at some finite scale – it is a manifold. We show how by representing the covering surface as a non-planar graph (an *embedded* graph [60]) of observed pixels we are able to plan new views and importantly fuse disparity maps from multiple views. The resulting graph can be lifted to 3D to yield a full scene reconstruction.

We consider the problem of workspace surface discovery using an actuated stereo camera. Our goal is to generate a sequence of control commands which will servo

the camera to take views of the workspace which guarantee that we image every point on every visible surface of every object in the workspace. With this complete coverage we can construct a single covering surface $\mathcal{S}$ of the entire workspace. This work is motivated by a desire to construct complete and detailed maps of unknown workspaces – a task which has an important role to play in automated inspection. We emphasise that the aim is to obtain surface coverage of an arbitrarily complex workspace using a stereo camera – not the exploration of free-space.

5.1.1 Summary

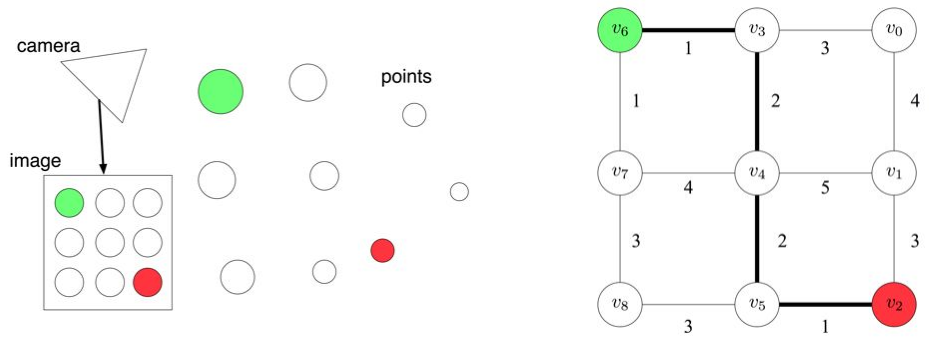
The approach in this Chapter can be summarised as follows. We construct a non-planar graph consisting of observed pixels as vertices. Neighbouring pixels in a stereo disparity image will be connected with an edge weighted in proportion to the Euclidean distance in $\mathbb{R}^3$ between vertices – spatial positions having been triangulated from stereo. As new views are taken the graph is grown such that at any time the edge between two vertices will correspond to the smallest observed Euclidean distance between corresponding pixels in any disparity image to date. By detecting and ranking the severity of one dimensional rifts in this graph (collections of edges with large weights in close proximity) we can plan a continuous trajectory through space to guide the camera to a new pose, the view from which will enable some if not all of the scene detail hidden by the rift to be filled in. The process can continue until no significant rifts remain. In what follows we will carefully explain the detail behind this scheme and cover some of the intricacies required to maintain a consistent graphical structure.

We make few assumptions in this work but they need to be made explicit. Firstly we require that we have accurate information on the pose of the camera at all times – a reasonable assumption given we are using a stereo pair and the state of the art in structure from motion and SLAM techniques [112]. Although the surface $\mathcal{S}$ will be

complex and self intersecting we place just two restrictions on the workspace itself. Firstly we require that it must be topologically connected. Secondly, for practical reasons we require that there exists at least one camera pose for any point $\mathbf{p} \in \mathcal{S}$ such that $\mathbf{p}$ is four connected when projected into a stereo disparity map. Put simply this implies that there are no prominent knife edges which resolutely remain one dimensional however close the camera gets to them.

Our work has interesting parallels with the Gap Navigation Tree [126] in which it is shown that a scheme that requires the chasing down of discontinuities perceived depth in a 2D environment provably enforces complete exploration. One could see the graphical approach we adopt here as a re-factorisation of this approach and a generalisation to 3D.

Part of this work was presented as "Discovering and Mapping Complete Surfaces With Stereo" at the IEEE International Conference on Robotics and Automation (ICRA) in Anchorage, Alaska, in May 2010 [109].



(a) **(b)**

Figure 5.1: Graph notation. An illustration of the conversion from an image and point cloud shown in (a), to a graph structure in (b). Each point from the point cloud in (a) is represented by a vertex in (b), and edges connect points which were adjacent in the source image. The weight of edge (u, v) corresponds to the Euclidean distance in $\mathbb{R}^3$ between the end points u and v. The shortest path, $\delta v_6, v_2$, from the green start vertex to the red goal vertex is shown in bold.

5.2 Graph Notation

We begin by defining the notation which will be used throughout this Chapter.

A graph, G, is a data structure which is an abstract representation of some objects and the connectivity between them [21]. It is defined by set of vertices V, and a set of connecting edges E. Vertices (or nodes) are the abstract representations of objects, for example we might create a vertex for each city in a country. An edge connects two vertices and can have a weight associated with it. If vertices are cities, then edges may be roads connecting them with weights corresponding to travel times.

We denote a graph as an ordered pair

$$G = (V, E) \tag{5.1}$$

where the first element, V, is a set of vertices

$$V = \{v_0 \dots v_i\} \tag{5.2}$$

and the second, E is a set of edges

$$E = \{(u, v) : u \in V, v \in V\} \tag{5.3}$$

In this definition an edge $(u, v) \in E$ is an unordered pair of vertices. A graph with this type of edge is an *undirected* graph, that is an edge has no implicit direction. As mentioned each edge $e \in E$ can have a weight or cost associated with it. We define a weight function

$$w(e) : E \to \mathbb{R} \tag{5.4}$$

which maps from an edge e to a real-valued weight. We can use this weight function to measure the cost of a path through the graph. A path $p = (v_0, v_1, \ldots, v_k)$ is defined as an ordered set of vertices with the property that for each neighbouring pair of vertices (v_i, v_{i+1}) in the set there exists an edge connecting them $(v_i, v_{i+1}) \in E$. The weight of a path $w(p)$ is a summation of the weights of each edge in the path

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \tag{5.5}$$

Later we will be looking at the problem of finding the *shortest path* from a start vertex to a goal vertex. Consider the simple graph of Fig. 5.1(b). We can see that there are many possible paths from the start vertex v_6 to the goal vertex v_2, of varying total cost. The shortest path is highlighted in bold – it has a total cost of 6. The weight of the shortest path is defined as

$$\delta(u, v) = \begin{cases} min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from u to v,} \\ \infty & \text{otherwise.} \end{cases} \tag{5.6}$$

The notation $u \overset{p}{\rightsquigarrow} v$ means that there exists a path p from vertex u to vertex v. Correspondingly the *shortest path* from u to v is defined as a path p with $w(p) = \delta(u, v)$.

5.3 Graph Construction

We now look at the construction of a graph given a stereo disparity image (and resulting point cloud).

5.3.1 Disparity from stereo

Stereo was chosen as the sensor modality for this research as it provides dense point clouds and high resolution source images at high speed. A stereo camera provides left and right images $\mathcal{I}^L$ and $\mathcal{I}^R$. Corresponding pixels are identified using a local window-based matching technique resulting in a *disparity image* $\mathcal{D}$.

As described in Chapter 3 every pixel of a disparity image stores a floating point value $d = \mathcal{D}_{r,c}$, the *disparity* of $\mathcal{I}^L_{r,c}$. This disparity is the horizontal displacement between pixels $\mathcal{I}^L_{r,c}$ and $\mathcal{I}^R_{r,c-d}$ which are images of the same point in $\mathbb{R}^3$.

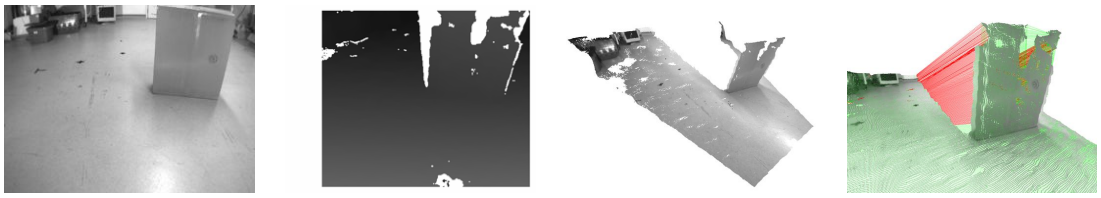
Some pixels do not have a disparity score due to poor matching confidence or occlusions and are assigned an error value. We will use the notation $\mathcal{D}_{valid}$ to refer to the subset of pixels in $\mathcal{D}$ which have valid disparities. A disparity image captured at time i will be referred to as ${}_i\mathcal{D}$.

Figure 5.2(a) and Fig. 5.2(b) show an example $\mathcal{I}^L$ and the resulting $\mathcal{D}$. Pixels $\notin \mathcal{D}_{valid}$ are coloured white. The implementation details of the stereo algorithm are described in Chapter 3.

5.3.2 Point cloud from disparity

With knowledge of the camera parameters (importantly focal length f and baseline b) we can back project a pixel in $\mathcal{D}$ and triangulate the position in $\mathbb{R}^3$. The back projection function π at camera pose $\mathbf{c}_i$ is:

$$\{\pi_i(r,c) : \mathbb{R}^2 \rightarrow \mathbb{R}^3\} = \left[\frac{b.c}{{}_i\mathcal{D}_{r,c}}, \frac{b.r}{{}_i\mathcal{D}_{r,c}}, \frac{f.b}{{}_i\mathcal{D}_{r,c}}\right]^T \tag{5.7}$$



(a) Left image **(b)** Disparity image **(c)** Point cloud **(d)** Graph detail

Figure 5.2: Construction of a graph surface from stereo. (a) is the input image $\mathcal{I}^L$ (the corresponding $\mathcal{I}^R$ is not shown). (b) shows the result of applying stereo processing to $\mathcal{I}^L$ and $\mathcal{I}^R$, the disparity image $\mathcal{D}$. Darker pixels are further from the camera. (c) the point cloud obtained by triangulation of valid disparity values. (d) detail of the graph structure – edges have been coloured according to weight: low weight edges are green, while high weight edges are red.

where (r, c) is a pixel location. Applying this function to all pixels in ${}_i\mathcal{D}_{valid}$ gives us a point cloud

$$\mathbf{P} = [\pi_i(r, c) : \forall [r, c] \in {}_i\mathcal{D}_{valid}] \tag{5.8}$$

This resulting point cloud is unorganised – it is simply a list of $[x, y, z]^T$ points in space. However, the source images which gave rise to this point cloud encode a great deal of information about the relationship between pixels and we aim to include this in the construction of our graph structure.

Two neighbouring pixels in an image with similar disparities suggest that the two pixels are images of neighbouring points on a surface in the world. Areas of a disparity image with smooth gradients indicate smooth surfaces in the world while discontinuities – edges in the disparity image – highlight physical discontinuities in space.

5.3.3 From RGB+D to Graph

An undirected graph, $G = [V, E]$, consists of a set of vertices, V, and a set of edges, E, which describe the connectivity between vertices. In the case of a single disparity

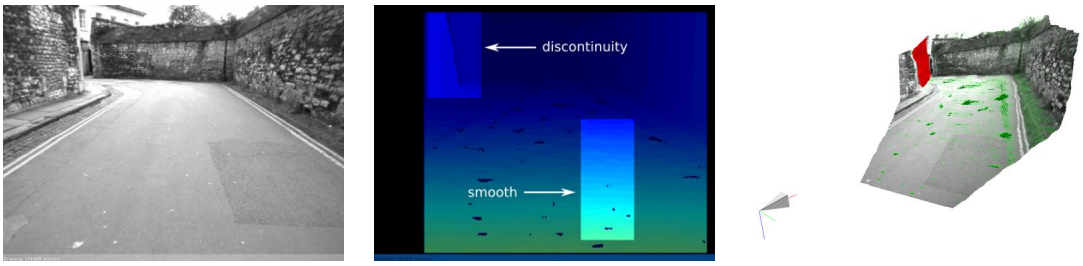
map from camera pose $\mathbf{c}_i$, every pixel $\mathbf{p} \in \mathbf{P}$ results in a graph vertex v.

Edges are constructed by considering the pixels which surround v in ${}_i\mathcal{D}$. If v was seen at ${}_i\mathcal{D}_{r,c}$, then we consider the neighbouring pixels ${}_i\mathcal{D}_{r\pm1,c\pm1}$. If one or more of these pixels has a valid disparity and hence a corresponding vertex u, then an edge is created connecting v and u. Every edge e has a weight $w(e)$ associated with it – in this case it is simply the Euclidean distance in $\mathbb{R}^3$ between the two end vertices:

$$w(e) = ||v_{\mathbf{x}} - u_{\mathbf{x}}||_2 \tag{5.9}$$

where $v_{\mathbf{x}}$ and $u_{\mathbf{x}}$ are the 3D positions of the end vertices.

Figure 5.2 shows the stages of constructing such a graph from a typical stereo image pair. $\mathcal{I}^L$ and $\mathcal{I}^R$ are used to create $\mathcal{D}$, Fig. 5.2(b). $\mathcal{D}$ is converted to a point cloud, Fig. 5.2(c), and is used to create the graph structure, Fig. 5.2(d). The graph edges have been coloured according to weight – low weight edges are green, while high weight edges are red.



(a) (b) (c)

Figure 5.3: Depth discontinuities. The identification of rifts in the graph corresponds to finding areas in the original depth map which contain depth discontinuities. (a) is the left image of a stereo pair and (b) is the corresponding disparity map. The highlighted areas illustrate the difference between smoothly varying disparity values on the flat road and discontinuous jumps such as at the end of the wall in the upper left. (c) shows the resulting graph structure overlaid on the point cloud – note that the high weight edges, shown in red, are those corresponding to the depth discontinuity highlighted in (b).

5.4 Identifying Rifts

As alluded to in the introduction to this Chapter, we want to identify exploration targets by finding collections of edges in the graph with high average edge weights. Figure 5.3(b) highlights two areas in a stereo disparity image. One area, labelled "smooth", is on the flat road surface which corresponds to an area of smoothly varying disparity values. This is uninteresting from an exploration perspective – there are no gaps or occlusions which may conceal unexplored terrain, the surface here contains no holes.

The second area labelled "discontinuity" is more interesting and we can see that this corresponds to the end of a wall and the corner of the road in Fig. 5.3a. The resulting graph is shown in Fig. 5.3c as per Section 5.3. The "smooth" area in the graph consists only of short edges as the smoothly varying disparity has resulted in spatially close points in $\mathbb{R}^3$, while the "discontinuity" area consists of edges with large weights – shown here in red. We now look at how we can programatically identify such regions, or rifts, in the graph.

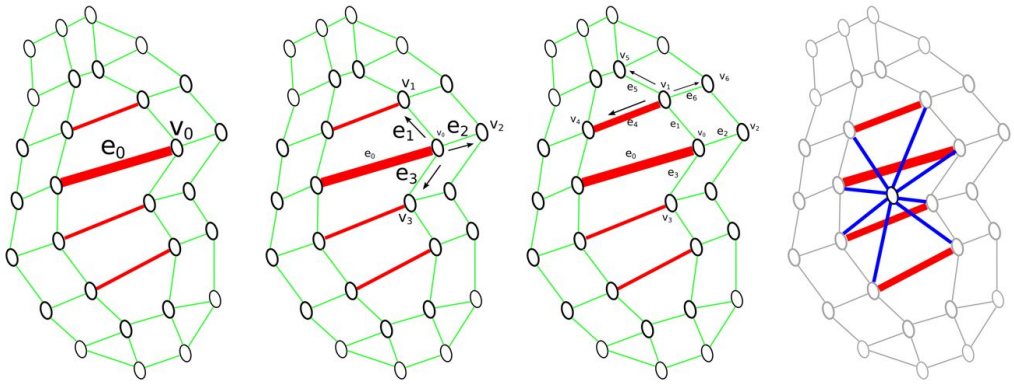


Figure 5.4: Rift generation. Starting with a high weight edge (bold red edge in (a)), we recursively follow high weight edges connected to the neighbours of its end vertices. Explored edges are indicated with arrows. The detected rift is shown in (b).

5.4.1 Expansion Algorithm

We have established that groups of edges with high weights and in close spatial proximity are indicative of discontinuities on the surface of the world explored so far. We call these groups of edges *rifts*, and our method for identification of rifts in a graph is as follows.

First a threshold, δ_m, is chosen which specifies the minimum edge cost before an edge is considered for rift membership. This is dependent on a number of factors; environment, sensor accuracy, robot characteristics. In choosing a value we are making a decision about the resolution we want from the final explored surface map. In the examples which follow we use a threshold of $\delta_m = 0.1m$. Effectively this says that exploration is not complete until the longest edge in the surface graph is shorter than $0.1m$.

The set of all edges which have a weight greater than or equal to δ_m is now found

$$R_m = \{e \in E : w(e) \geq \delta_m\} \tag{5.10}$$

The rift identification algorithm is initialised by choosing the edge with the

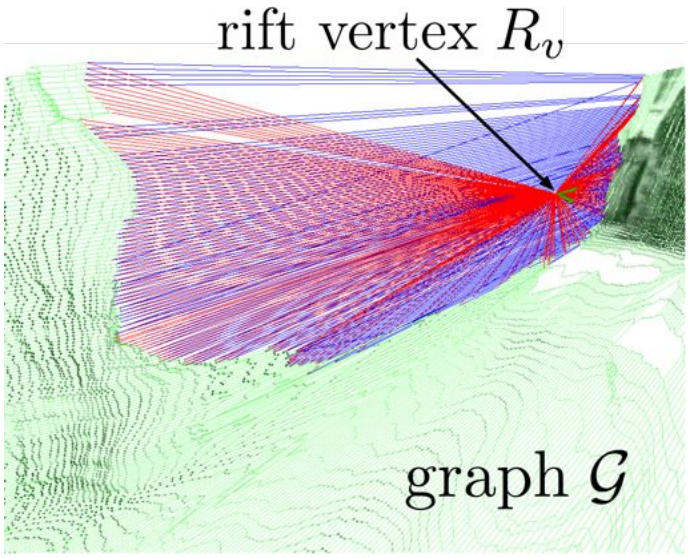


Figure 5.5: Identified rift. A rift having been identified by Algorithm 2 is created by initialising a new graph vertex at the geometric mean of the constituent vertices. New graph edges (shown here in red) connect this new vertex to the existing graph. This is the rift resulting from the depth discontinuity shown in Fig. 5.3.

highest cost $e \in R_m$ with end vertices (u, v). We recursively expand u and v looking for neighbouring vertices which themselves are connected to high weight edges. If a neighbouring vertex has no high weight edges attached to it then it is discarded. The list of high weight edges encountered in this graph traversal is stored as a new rift R. A *rift vertex* is created at the geometric centre of R and is connected to the constituent vertices. This is described in pseudo-code in Algorithm 2.

5.4.1.1 Resulting Rifts

The process is shown graphically in Fig. 5.4 on a simple graph containing one rift. Green edges are edges which have a cost below the high weight threshold δ_m, while red edges are high cost edges.

The emboldened graph edge e_0 in Fig. 5.4(a) is the edge with the highest weight and is the starting point for rift generation. A list of vertices to expand, the $ToVisit$ set is created, and a new rift R is initialised to contain just e_0.

Beginning with the vertex v_0 we inspect its neighbours $\{v_1, v_2, v_3\}$. v_2 is discarded

as it has no high weight edges, but v_1 and v_3 are added to the $ToVisit$ set for further expansion. In Fig. 5.4(c) we see v_1 being expanded and edge e_4 is added to the rift R. This process continues until we have exhausted the $ToVisit$ list.

Finally the new rift vertex is created at the geometric centre of the rift, and connected to all the vertices of the constituent edges. The result is shown in Fig. 5.4(d)

Any edge which is contained in this rift is removed from the set R_m, and the algorithm is repeated until $R_e = \emptyset$. The result is a set of rifts $\mathbf{R} = [R_0, R_1, ..., R_n]$.

Figure 5.5 shows the result on some real data. The blue edges are the high weight edges which constitute the rift R. Red edges are new edges which connect the rift vertex to the graph.

Algorithm 2: Rift identification in a graph

Input: Seed edge e_s
Edge weight threshold δ_m
Output: Rift R

$ToVisit = [e_s]$
while $ToVisit \neq \emptyset$ **do**
  $e = pop(ToVisit)$
  foreach $n \in Neighbours([e_u, e_v])$ **do**
    foreach $d \in Edges(n)$ **do**
      if $d_w > \delta_m$ **then**
        $ToVisit = ToVisit \cup d$
      end
    end
  end
end

5.5 Planning Next View

At this stage we have a graph structure which describes the workspace surface explored so far. In this graph we have identified one or more rifts – groups of high weight edges.

We want to plan a continuous camera trajectory through space from the current pose $\mathbf{c}_i$ to a new pose $\mathbf{c}_{i+1}$. The view from $\mathbf{c}_{i+1}$ should enable the visibility based resolution of a rift, and thus increase coverage of the workspace surface.

5.5.1 Rift selection

When faced with a set of rifts $\mathbf{R}$, we order them according to severity and distance from the current camera position $\mathbf{c}_i$ to the rift center:

$$weight(R_i) = \alpha(|\mathbf{c}_i - R_i|) + \beta(\#R_i) \tag{5.11}$$

where $|\mathbf{c}_i - R_i|$ is the distance through the graph G from the camera pose $\mathbf{c}_i$ to the rift vertex at the centre of R_i. Severity, $\#R$, is a measure of rift size and is taken to be the number of edges in the rift. α and β are weightings which are chosen depending on the application and the graph scale. Further experimentation could lead to optimising these constants, but in this work we use $\alpha = 0.5$ and $\beta = 0.5$. The rift with the highest weight is named the *dominant* rift

$$R_d = \max_i \{weight(R_i)\} \tag{5.12}$$

and we plan a camera trajectory such that the end camera pose is oriented along the normal vector of R_d, which we will now describe how to estimate.

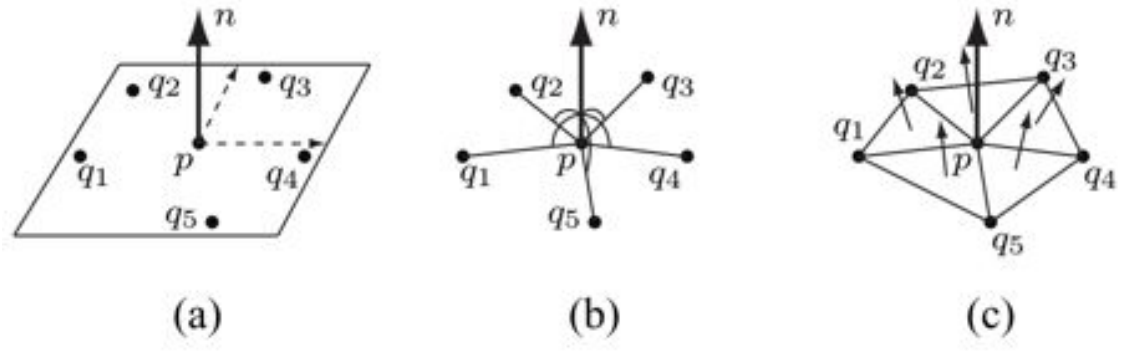


Figure 5.6: Surface normal estimation. Different approaches for estimating normal vectors. (a) shows plane fitting to a neighbourhood of points around p. (b) is maximising the angle between the normal vector of p and the tangential vectors to neighbours of p. (c) finds triangles formed from pairs of neighbours and p and averages their normal vectors. Image from [53].

5.5.2 Surface Normal Estimation

For a given vertex, v, we calculate the local surface normal by considering its immediate neighbours in the graph and fitting a plane to this point set. Inherent in this method is a tradeoff between data smoothing and sensitivity to noise – a large neighbourhood leads to noise insensitivity but poor resolution. We have found that using vertices within one graph edge of v gives acceptable results.

A typical vertex will have four immediate neighbours, but after graph merging (to be discussed in Section 5.6) may have more. Following the notation conventions of [53] we have the vertex point $\mathbf{p_i}$ and its immediate neighbours $\mathbf{q}_{i1}, \mathbf{q}_{i2}, \ldots, \mathbf{q}_{ik}$ which we will store in two vectors: $\mathbf{Q}_i$ which contains only the neighbours, and $\mathbf{Q^+}_i$ which contains the neighbours and the point $\mathbf{p}_i$

$$\mathbf{Q}_i = [\ \mathbf{q}_{i1},\ \mathbf{q}_{i2},\ \ldots,\ \mathbf{q}_{ik}\ ]^T \qquad \mathbf{Q^+}_i = [\ \mathbf{p_i},\ \mathbf{q}_{i1},\ \mathbf{q}_{i2},\ \ldots,\ \mathbf{q}_{ik}\ ]^T \tag{5.13}$$

Generally speaking we are trying to calculate the normal of $\mathbf{p}_i$ by solving the

optimisation problem

$$\min_{\mathbf{n}_i} J(\mathbf{p}_i, \mathbf{Q}_i, \mathbf{n}_i) \tag{5.14}$$

where $J(\mathbf{p}_i, \mathbf{Q}_i, \mathbf{n}_i)$ is a penalising cost function which varies depending on the surface normal algorithm being used.

The result will be a normal $\mathbf{n}_i = [n_{ix}, n_{iy}, n_{iz}]^T$ of a local plane

$$S_i = n_{ix}x + n_{iy}y + n_{iz}z + d \tag{5.15}$$

A number of surface normal estimation methods for point clouds are described and evaluated in [53]. Figure 5.6 shows some of these: Figure 5.6(a) shows plane fitting to a neighbourhood of points around p. In this case $J(\mathbf{p}_i, \mathbf{Q}_i, \mathbf{n}_i)$ would penalise distance of points from the plane. Figure 5.6(b) maximises the angle between the normal vector of p and the tangential vectors to neighbours of p. $J(\mathbf{p}_i, \mathbf{Q}_i, \mathbf{n}_i)$ will penalize agreement between tangential vectors and the normal vector. Figure 5.6(c) finds triangles formed from pairs of neighbours and p and averages the normal vectors of these triangles.

The conclusion of [53] is that for reliability, quality, and speed, a generally good choice is the technique dubbed PlanePCA This is a modification of the standard singular value decomposition (SVD) method of [46].

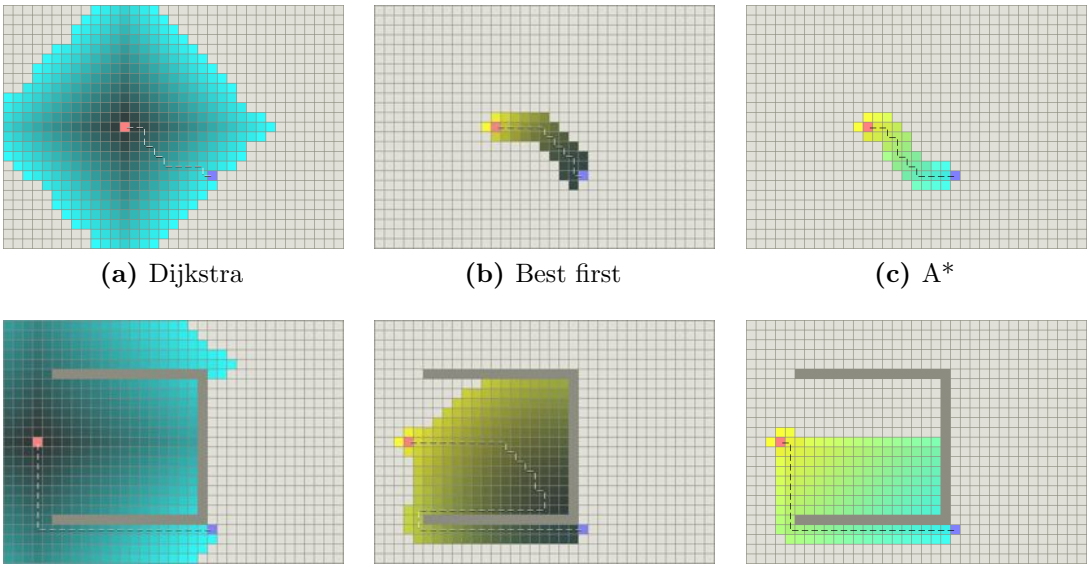
First we centre the data matrix $\mathbf{Q}_i^+$ on the origin by subtracting the empirical mean, and solving

$$\min_{\mathbf{n}_i} \left|\left|[\mathbf{Q}_\mathbf{i}^+ - \overline{\mathbf{Q}}_i^+]\mathbf{n}_i\right|\right|_2 \tag{5.16}$$

where $\mathbf{n}_i$ is the plane normal and $\overline{\mathbf{Q}}_i^+$ is the mean-centred matrix

$$\overline{\mathbf{Q}}_i^+ = \left[\mathbf{Q}_i^+ - \frac{1}{k}\sum_{j=1}^{k}\mathbf{Q}(j)\right]^T \tag{5.17}$$

This is solved by taking the SVD ($\mathbf{U}\Sigma\mathbf{V}^T$) of $\mathbf{Q}_\mathbf{i}^+ - \overline{\mathbf{Q}}_i^+$ The vector in $\mathbf{V}$ corresponding to the smallest singular value in Σ is the normal of the plane.



(a) Dijkstra **(b)** Best first **(c)** A*

(d) Dijkstra **(e)** Best first **(f)** A*

Figure 5.7: Graph search algorithms. A comparison between three shortest path search algorithms. The aim is to find the shortest path across a graph from a start vertex to a goal vertex. In this example the graph is constructed from a 2D grid – each grid cell is a graph vertex and edges connect neighbouring grid cells. Dijkstra's algorithm expands outwards in all directions from the start vertex until it finds the goal. Best-first tries to guide exploration towards the goal vertex, but is easily shown to be sub-optimal if we include a concave obstacle. A* employs a heuristic to guide the search in the general direction of the goal. Both Dijkstra and A* are guaranteed to find the shortest possible path, but A* will typically do so while examining far fewer vertices. Images from [95].

5.5.3 Choosing a Camera Path

We are now in a position to plan a camera trajectory which will move the depth sensor to view the chosen rift R_d down the normal vector and thus expand the explored surface. In other words we want to find a path from the current camera position on the surface graph to the graph vertex at the centre of the rift. This is framed as a graph search problem – specifically finding the shortest path between two vertices in a weighted undirected graph. The notation used is described previously in Section 5.2.

A simple example graph is shown in Fig. 5.1(b). Vertices are drawn as circles with the start vertex v_6 in green and the goal vertex v_2 in red. Edges have weights

which are some measure of how expensive it is to traverse that edge – in this case the Euclidean distance in $\mathbb{R}^3$ between the end points. We want to find a *minimum cost path* p through the graph from v_6 to v_2.

Equation (5.6) defines what we mean by a minimum cost path but it gives no indication how to find such a path. A simple approach might be to enumerate all possible paths but this is inefficient. Consider a typical graph from a single point cloud as shown in Fig. 5.3(d) where $|V| \approxeq 200000$ and $|E| \approxeq 400000$. Excluding cyclic paths (paths which revisit vertices) still leaves us with millions of paths most of which are not worth considering. A number of algorithms exist which guide the graph search towards optimal paths, without the need to examine every possibility.

5.5.3.1 Dijkstra's algorithm

The prototypical solution is the algorithm developed in 1959 by Dijkstra [22]. Dijkstra's algorithm finds the shortest paths from a single source vertex on a weighted, directed graph $G = (V, E)$ with non-negative edge weights.

The algorithm maintains two sets of vertices: the *closed* set which contains vertices to which the shortest path has been found, and an *open* set which contains vertices lying on the 'fringe' of exploration. New vertices to expand are selected from the open set based on the total path length from start to chosen vertex. Neighbours of this chosen vertex are added to the open set if they've never been seen before, and the algorithm repeats this process until the goal vertex is found.

Dijkstra's algorithm is guaranteed to find the shortest path but it comes at the cost of inspecting and expanding a large number of vertices. Figure 5.7(a) and (d) show the typical behaviour of Dijkstra – expanding outwards almost uniformly in all directions until it finds the goal.

5.5.3.2 Best First Search

Dijkstra's algorithm is guaranteed to find a shortest path $\delta(s, g)$, but given that we know the position of the goal and the start can we use this knowledge to guide the graph search to find $\delta(s, g)$ more efficiently?

Consider the regular grid map of Fig. 5.7. An estimate of the distance from a given vertex v to the goal g could be the Manhattan distance between their two grid positions $\mathbf{x}_v = [x_v, y_v]^T$, $\mathbf{x}_g = [x_g, y_g]^T$

$$h(v, g) = |x_g - x_v| + |y_g - y_v| \tag{5.18}$$

Best First Search [95] (BFS) is an algorithm which attempts to exploit this knowledge. Inspect the neighbours of the current vertex and move to the one with the lowest $h(v, g)$ estimate. Any neighbours which have a higher h are stored in case of back-tracking later if a dead-end is reached. This is repeated until the goal vertex is found. In very simple environments with few or no obstacles like Fig. 5.7(a)-(c) this performs very well and leads us straight to the goal. However, as Fig. 5.7(e) shows, as soon as obstacles are introduced BFS produces sub-optimal paths. This is not unexpected – a greedy algorithm which only considers the locally optimal solution is always prone to being trapped in dead-ends.

5.5.3.3 A* algorithm

The A* search algorithm [40, 105] lies somewhere in between Dijkstra and Best First Search. It uses an *heuristic* estimate h of the distance from each vertex to the goal to improve performance but it does so in conjunction with a priority queue which takes into account not only the heuristic estimate at each vertex but the cost required to get to that vertex in the first place. Figure 5.7(c) and (d) show that A* will find optimal paths but in doing so inspects far fewer vertices than Dijkstra.

The priority queue Q used in Dijkstra's algorithm is ordered by the shortest path distance from s to each vertex $v \in Q$. A* uses the same priority queue but modifies the ordering by introducing the heuristic h weight

$$f[v] = d[v] + h[v] \tag{5.19}$$

where $d[v]$ is the shortest distance estimate from $s \rightarrow v$ as used previously in Dijkstra's algorithm, and $h[v]$ is the heuristic distance estimate from $v \rightarrow g$.

How do we calculate the heuristic function h? If we are working on a regular grid like Fig. 5.7 we can use the Manhattan distance, see Eq. (5.18). In general though we will be working with irregular graphs in $\mathbb{R}^3$ and so we use the Euclidean straight line distance between two points

$$\delta(s, g) \geq |\mathbf{x}_g - \mathbf{x}_s| \tag{5.20}$$

$$|\mathbf{x}_g - \mathbf{x}_s| = \sqrt{(x_g - x_s)^2 + (y_g - y_s)^2 + (z_g - z_s)^2} \tag{5.21}$$

where $\mathbf{x}_s$ is the position of s in $\mathbb{R}^3$, $[x_s, y_s, z_s]^T$ and $\mathbf{x}_g$ is the position of g, $[x_g, y_g, z_g]^T$.

Equation (5.20) is a lower bound on $\delta(s, g)$. This is true if we have constructed our graph carefully – ensuring that it maps to the real world with edge weights which correspond to Euclidean distances in $\mathbb{R}^3$. In other words if there was an edge directly connecting s and g then it must have a weight exactly equal to Eq. (5.20).

The A* algorithm is only guaranteed to produce global shortest paths if the heuristic function used is *admissible*. An admissible heuristic has the properties that it will never underestimate the true cost to reach the goal. The Euclidean distance between two points is an admissible heuristic as the straight line distance is a strict lower bound on the true path distance.

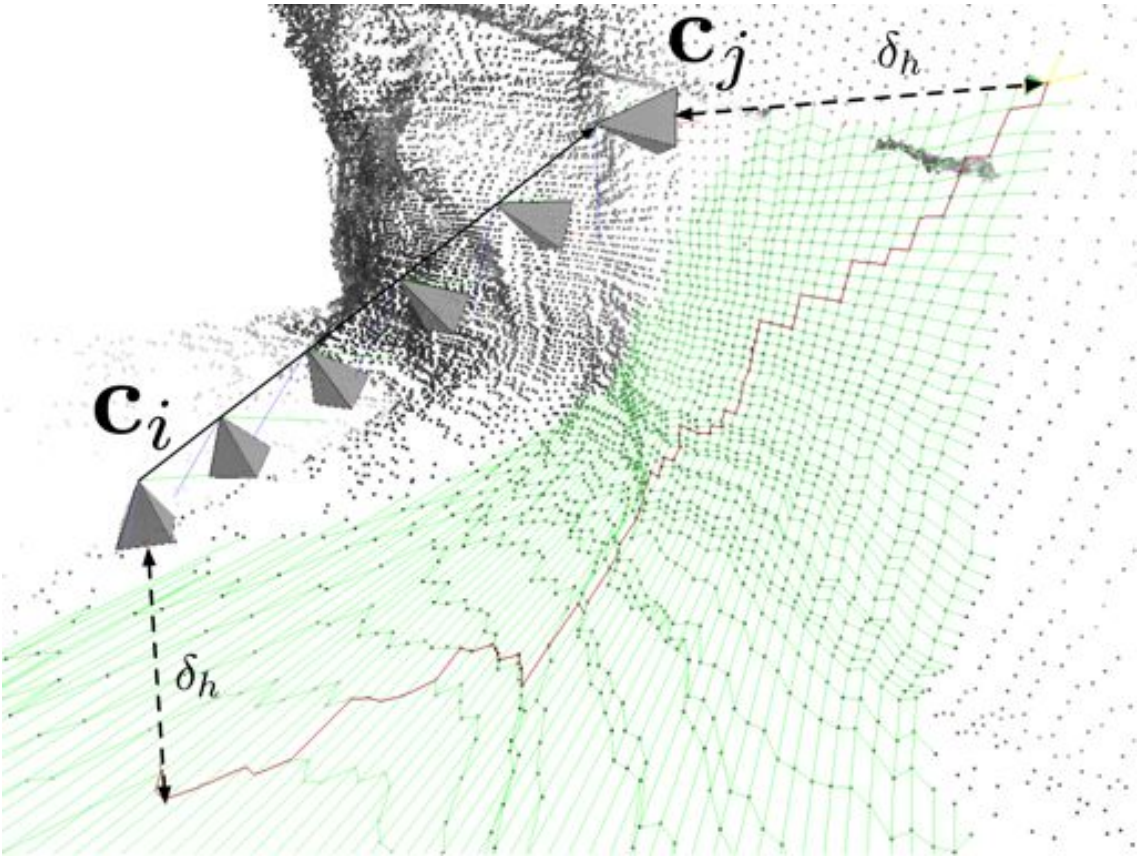


Figure 5.8: A^*over a real graph. The result of applying the A^*graph search algorithm to a graph constructed during exploration. A path is planned across the graph from the vertex associated with $\mathbf{c_i}$ to the vertex associated with $\mathbf{c_j}$. A camera trajectory is then created such that the camera views the graph from a height δ_h. Note that the camera path shown here is an idealized interpolation between the endpoints of the true camera path.

5.5.4 Planning a Camera Trajectory

The camera trajectory is found by planning a path from the current camera position $\mathbf{c}_i$ to a pose $\mathbf{c}_{i+1}$ which is looking along the normal of the dominant rift R_d. The path is planned using the A^*algorithm and results in an ordered set of vertices describing the shortest path p from $\mathbf{c}_i$ to $\mathbf{c}_{+1}$

Each vertex on this path has a normal $v_\mathbf{n}$ – calculated using the SVD plane fitting algorithm described in Section 5.5.2. The final camera trajectory consists of a ordered set of poses $\mathbf{c}_{path} = [\mathbf{c}_s, \ldots, \mathbf{c}_g]$ each oriented such that they view the workspace surface along the normal at a fixed height.

Figure 5.8 shows a camera trajectory being planned across the surface of the graph, from $\mathbf{c}_i$ to $\mathbf{c}_j$. The camera stays at a constant distance δ_h from the surface.

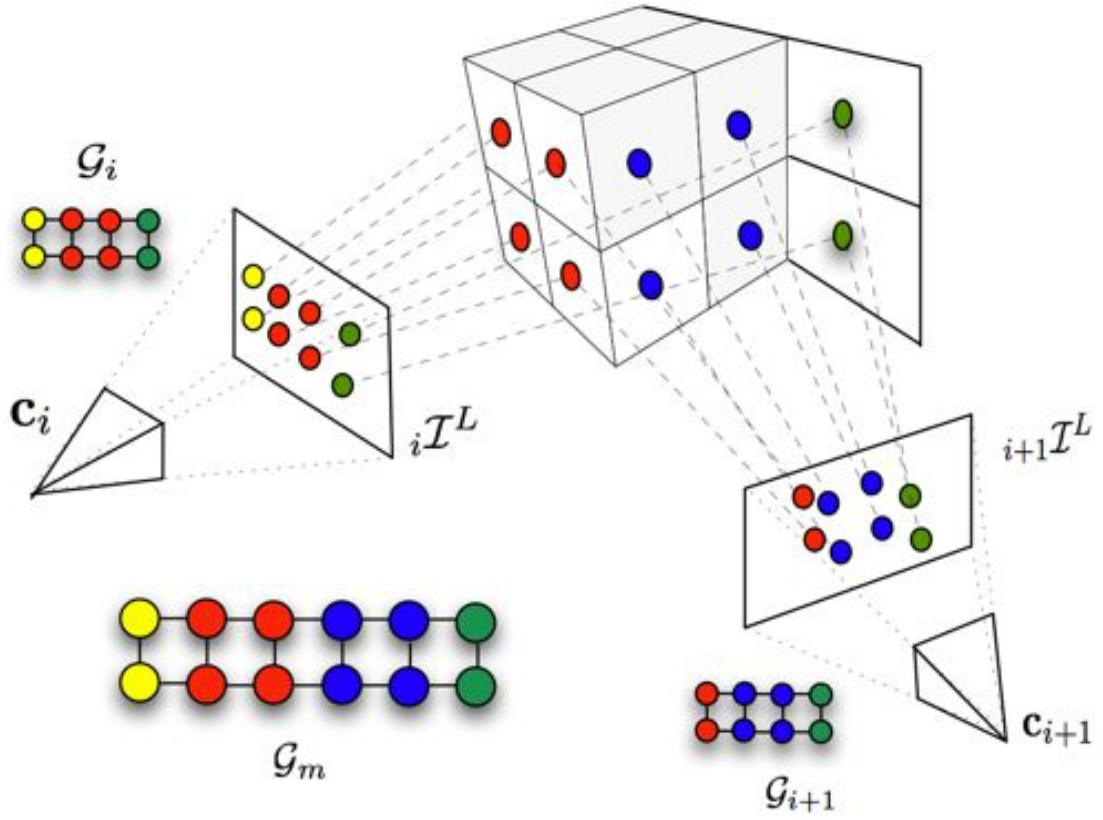


Figure 5.9: Graph merging. Each vertex v, in G_{i+1} is projected into ${}_i\mathcal{I}^L$. If the pixel it projects onto is valid ($[r,c]^T \in \mathcal{D}_{valid}$) then the outgoing edges of v are added to the vertex in G_i corresponding to ${}_i\mathcal{D}_{r,c}$, and v is discarded. Otherwise v is added to G_i as a new vertex. The result is the merged graph G_m.

5.6 Merging Graphs

Exploration of an environment necessarily involves moving the sensor to capture data from a new camera pose. On capturing new data we construct a new graph as per Section 5.3 and then look to integrate this with the existing graph.

Any given rift seen from one pose may not be visible from the next. To ensure complete surface coverage all rifts must be inspected at some point, and so we do not want to discard the old graph structure after a change in pose. The merging process aims to combine vertices which are representations of the same point in $\mathbb{R}^3$ but which are from independent graphs.

Figure 5.9 shows a simplified version of the situation. We have two camera poses

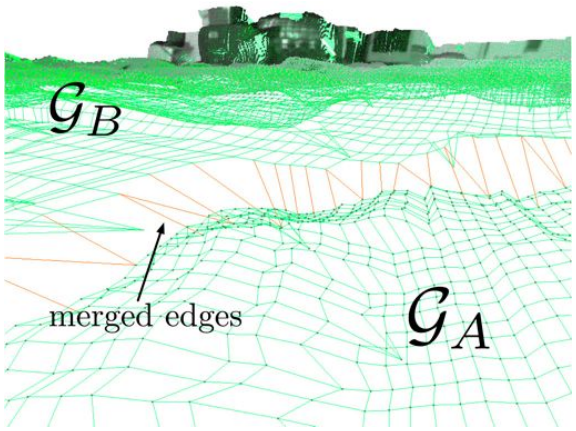


Figure 5.10: Graph merging result. Detail of the graph merging operation on real data – orange edges are new and join two graphs, G_A and G_B.

$\mathbf{c}_i$ and $\mathbf{c}_{i+1}$, and associated graphs G_i and G_{i+1}. The aim is to merge these two graphs into a single graph G_m.

First we define a projection function from a point $\mathbf{p}$ to image coordinates $[r, c]^T$

$$\{\pi^{-1}(\mathbf{p}) : \mathbb{R}^3 \rightarrow \mathbb{R}^2\} = \left[\frac{f.p_x}{p_z}, \frac{f.p_y}{p_z}\right]^T \tag{5.22}$$

The idea is to reproject the point represented by each vertex in G_{i+1} into the disparity image captured at the previous timestep ${}_i\mathcal{D}$

$$[r, c]^T = \pi^{-1}(\mathbf{p}) \tag{5.23}$$

Recall that we defined the set $\mathcal{D}_{valid}$ as the set of pixels in the disparity image which had a strong enough correlation in the stereo matching process to be considered good matches. For a vertex v to be merged into the existing graph two conditions must be met

- the vertex $v \in G_{i+1}$ must reproject onto a valid pixel, i.e. $[r, c]^T \in {}_i\mathcal{D}_{valid}$
- $\mathbf{p}_v = [p_x, p_y, p_z]^T$ must at the same position in $\mathbb{R}^3$ as the projection of $[r, c]^T$, i.e. $\left|\mathbf{p}_v - \pi([r, c]^T)\right| \leq \delta_{3d}$

Algorithm 3: Graph merge operation

Input: $G_i = [V_i, E_i]$
$G_{i+1} = [V_{i+1}, E_{i+1}]$
Minimum distance threshold γ
Output: Merged graph: $G_m = [V_m, E_m]$

$G_m = G_i$
foreach $v \in V_{i+1}$ **do**
  $[r, c]^T = \pi^{-1}(v_{\mathbf{x}})$
  $ValidPixel = [r, c]^T \in {}_i\mathcal{D}_{valid}$
  $Distance = ||\pi_{i+1}(r, c) - v_{\mathbf{x}}||_2$
  if *ValidPixel && Distance* $< \gamma$ **then**
    foreach $[u, v] \in Edges(v)$ **do**
      $E_m = E_m \cup Edge(u, {}_i\mathcal{D}_{r,c})$
    end
  end
  else
    $V_m = V_m \cup v$
    $E_m = E_m \cup Edges(v)$
  end
end

Less formally this is a check to see if a vertex from G_{i+1} is close enough in 3D to a vertex from G_i that we can consider them to be the same point.

This check for spatial proximity is a threshold on Euclidean distance between $\mathbf{p}_v$ and $\pi_{i+1}(r, c)$. This threshold is chosen based on the scale of the environment and our confidence in the sensor accuracy – we use a threshold of $0.01m$ in the lab environment.

If a vertex fails either condition then we add it as a new vertex to G_m along with any edges it might have in G_{i+1}. If however it passes both tests then we add the edges of v in G_{i+1} to the vertex corresponding to ${}_i\mathcal{D}_{r,c}$ in G_i, and discard v. Pseudocode for this operation is given in Algorithm 3. The process is shown graphically in Fig. 5.9 for a simple synthetic scene.

5.7 Edge Visibility

The problem which now arises is that of determining when an edge rift has been resolved, i.e. the unknown area of the workspace which exists behind the rift has been explored. Consider an edge e seen from pose $\mathbf{c}_i$ and a new pose $\mathbf{c}_{i+1}$. This is shown in Fig. 5.11.

If there exists a vertex in G_{i+1} which is collinear with a point on e and the camera centre $\mathbf{c}_{i+1}$ then we perform a visibility test. The projection of edge point $\mathbf{p}_e$ in ${}_{i+1}\mathcal{I}^L$ is

$$[r, c]^T = \pi_{i+1}^{-1}(\mathbf{p}_e) \tag{5.24}$$

If $[r, c]^T \in \mathcal{D}_{valid}$, we can retrieve the $\mathbb{R}^3$ position of the corresponding vertex v:

$$\mathbf{p}_{r,c} = v_{\mathbf{x}} \tag{5.25}$$

The two points $\mathbf{p}_e$ and $\mathbf{p}_{r,c}$ are collinear with the camera center $\mathbf{c}_{i+1}$ as they both project onto the same pixel. The visibility test is a comparison of the distances from the camera center to both points:

$$d_{\mathbf{p}_e} = ||\mathbf{p}_e - \mathbf{c}_{i+1}||_2 \tag{5.26}$$

$$d_{\mathbf{p}_{r,c}} = ||\mathbf{p}_{r,c} - \mathbf{c}_{i+1}||_2 \tag{5.27}$$

If $d_{\mathbf{p}_e} < d_{\mathbf{p}_{r,c}}$ then point $\mathbf{p}_{r,c}$ is visible behind $\mathbf{p}_e$ and passes the test. The end points of e are projected into ${}_{i+1}\mathcal{D}$ and the set of pixels to test is chosen by the Bresenham line drawing algorithm. Edge e is deleted from G if the fraction of points passing the visibility test is greater than a threshold, typically 75% of the pixels.

Figure 5.11 shows a graphical example. The red edges e and e' were created at

pose $\mathbf{c}_i$. The camera has now moved to $\mathbf{c}_{i+1}$ and we use the visibility test on e and e' to determine whether they survive. Edge e will be deleted as the camera can now see pixels $\mathbf{p}_{r,c}$ behind every point on e. Edge e' will survive the visibility test because all points $\mathbf{p'}_{e'}$ are occluded by pixels $\mathbf{p'}_{r,c}$.

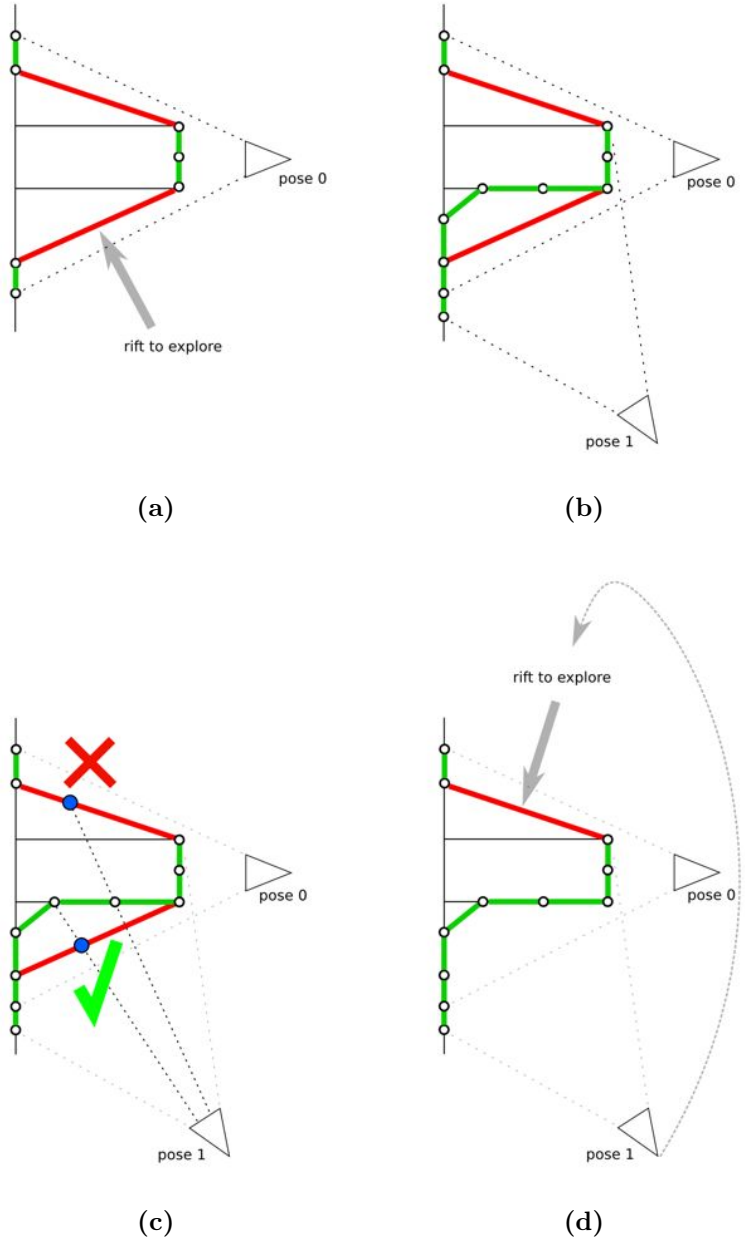
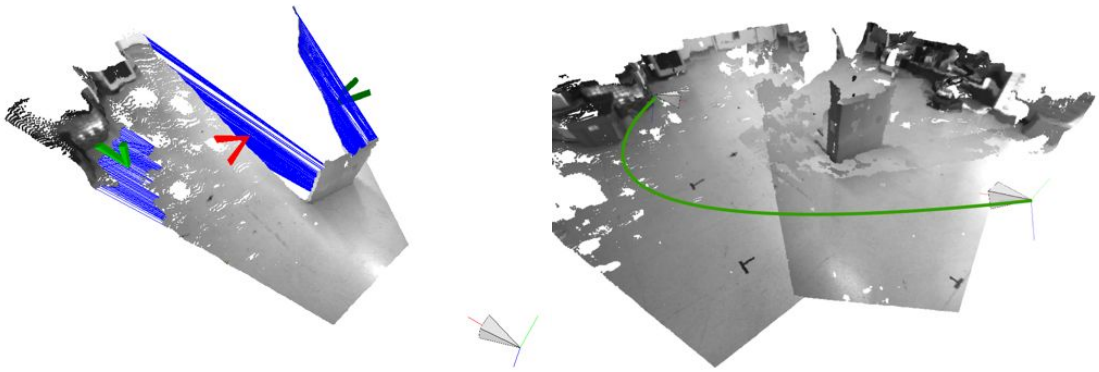


Figure 5.11: Visibility constraint testing. The depth image captured at pose 0 results in the graph shown in (a). A rift is chosen as the exploration target and the camera is repositioned at pose 1 and captures a new depth image, the merged graph is shown in (b). The visibility test is performed by reprojecting rifts into the latest image and comparing depths of reprojected pixels. If a significant fraction of reprojected rift pixels are found to be closer to the camera than the pixels in the latest depth image then the rift is resolved and is deleted from the graph, as shown in (c). Exploration then continues in (d).



(a) Rifts detected **(b)** Final point cloud

Figure 5.12: Lab results. (a) shows the initial rifts detected in the point cloud with the dominant rift, R_d, in red. (b) shows the point cloud after moving the camera to inspect R_d and performing merging and visibility checks. Note that the camera path shown here in green is an idealized interpolation between the endpoints of the true camera path.

5.8 Results

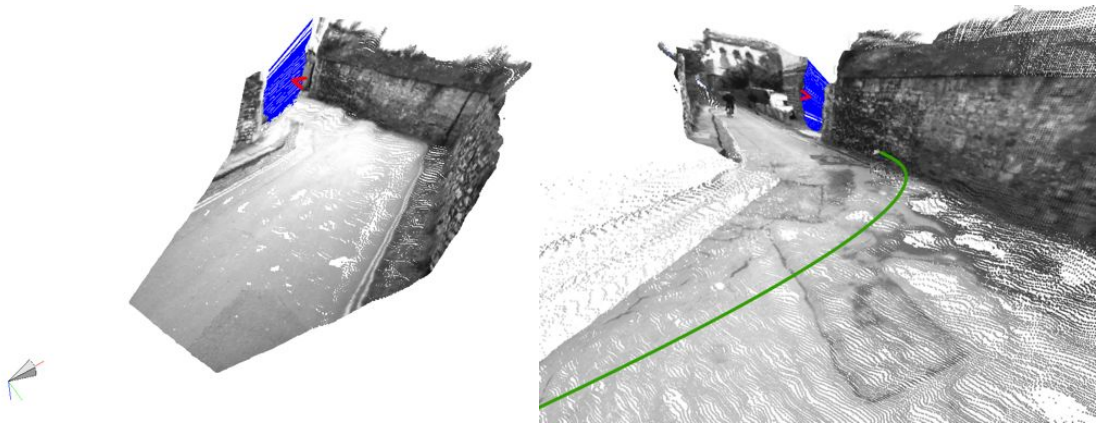
5.8.1 Experimental setup

A detailed description of the acquisition of 3D data can be found in Chapter 3. A handheld PointGrey Bumblebee stereo camera was used to capture mono images $\mathcal{I}^L$ and $\mathcal{I}^R$ at a resolution of 512×384 pixels. The camera poses were accurately estimated with a robust visual odometry system [113].

5.8.1.1 Real data

Figure 5.12 and Fig. 5.13 show the results of applying our method to real data. Figure 5.12 is data taken from a sequence of images of a lab environment, another image of which is seen in Fig. 5.2(a).

Figure 5.12(a) shows the graph generated from a single camera pose. Rifts have been identified and highlighted with green arrows, the dominant rift, R_d, with red. Fig. 5.12(b) is the result of a change in camera pose to point along the normal of R_d.



(a) Rifts detected **(b)** Final point cloud

Figure 5.13: New College results. (a) shows the initial rifts detected in the point cloud with the dominant rift, R_d, in red. (b) shows the point cloud after moving the camera to inspect R_d and performing merging and visibility checks. Note that the camera path shown here in green is an idealized interpolation between the endpoints of the true camera path.

	Fig. 5.12(a)	**Fig. 5.12(b)**	**Fig. 5.13(a)**	**5.13(b)**
Raw points	140789	548978	15124	605831
Graph vertices	140789	422060	151214	447503
Graph edges	280130	902108	300032	884977
Surface area	$13m^2$	$52m^2$	$177m^2$	$452m^2$

Table 5.1: Exploration data in real-world environments

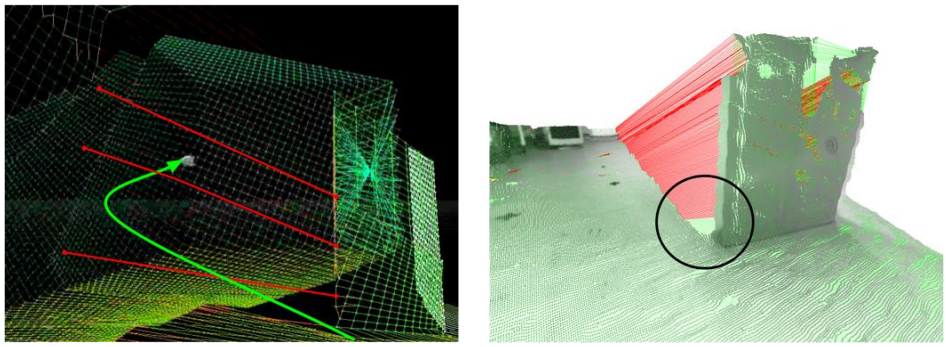
Figure 5.13(a) is a graph generated from an image of a street. Note the high walls ahead and to the right meaning that the obvious place to explore is the corner to the left. The dominant rift is indeed oriented such that the new camera pose will be pointing into this unknown area. Figure 5.13(b) shows a view of the previously unseen workspace. The original dominant rift has been deleted due to the visibility check of Section 5.7, and a new dominant rift has been chosen.

The camera paths resulting in both Fig. 5.12(b) and Fig. 5.13(b) consisted of 4 intermediate poses. The graphs were merged and visibility checks were done at each pose.

Numerical data for these figures is given in Section 5.8.1.1.

- **Raw points** is the number of points without any graph merging – concatenated point clouds.
- **Graph vertices** is the reduced point set after merging (this is the same as raw points for single poses).
- **Graph edges** is the number of edges after merging.
- **Surface area** is a rough estimate of the world surface covered by the graph.

Figures 5.12 and 5.13 and Section 5.8.1.1 show that chasing down the dominant rift has improved the completeness of the surface of the workspace. This is shown both in terms of surface area covered and from views of the resulting point clouds.



(a) (b)

Figure 5.14: Problems with the approach. Two images which show specific problems with the approach described in this chapter, and help to motivate the approach described in the next chapter.
(a) highlights edges which fall below the threshold for rift inclusion, but nevertheless are not lying on the surface of the world. This is one of many arbitrary thresholds which are required in this approach, and are undesirable.
(b) shows stray edges which have not been resolved by the visibility check, even though the camera trajectory suggests that they should have been. This is a result of the limited precision reprojection step, which attempts to find the intersection between two idealised lines.

5.9 Discussion and Conclusions

This chapter has introduced the idea of viewing the world as a single continuous surface and of exploration as "filling in" holes in this surface. Representing the surface as a graph has allowed us to identify such holes, or rifts, and to efficiently plan a path across the surface to the next such rift. Results have been presented which show this algorithm working on a small scale – pushing back unexplored frontiers and expanding the map of the workspace.

There are however a number of problems with this approach which became apparent when trying to scale to larger environments:

Unrealistically High Precision This approach relies on the reprojection of points and lines into captured images and identifying matches. This is hugely susceptible to failure due to noisy measurements. Even if we had perfect pose and sensor data, the merging step can be seen as the problem of finding the union

of two arbitrary polyhedra – shown to be NP-hard [124].

Scaling The visibility check of Section 5.7 has to happen for every previous disparity map. This requires reprojection of every edge in the viewing frustum of the camera to be reprojected into an ever growing set of images. We could perhaps mitigate this by identifying the subset of previous views which could possibly appear in the viewport of the current view, and then only reproject into this subset.

To perform this then, requires the storage of all previous disparity images along with the associated camera poses. The memory requirements therefore scale linearly with number of scans. The problem is that this does not directly map to exploration progress – if a robot drives in circles the storage requirements will continue to increase. Contrast this to the occupancy grid methods in which a scan can be discarded after integration into the map.

Path planning A robot or actuated sensor doesn't live on the surface of the world as is assumed in this work. Simply planning a path across the surface can be problematic given the physical dimensions of a robot – the graph surface representation does not take into account feasibility of planned paths

The problem is that there is no notion of freespace explicitly encoded in this representation, and it's therefore difficult to guarantee a collision free path by just considering the graph surface. We realised that the entire structure would need to be lifted into 3D space at each planning step, negating somewhat the benefits of the graph structure.

Arbitrary thresholds Thresholds are set somewhat arbitrarily throughout – how close do points have to be in space before we merge them? How high above the graph should the camera path be? What values for α and β should be set when weighing up which rift to explore? What edge length classifies it as an

exploration target? Throughout these have been set by adjusting them to fit the environment but this practice certainly doesn't feel very rigorous.

If the threshold for rift edge is too low then we will waste time exploring areas which require no further observation. The alternative, setting the threshold too high, will result in us missing areas of importance. No matter how this threshold is set, there will always be edges in the graph which lie in free-space and will therefore result in inefficient path planning. Figure 5.14(a) highlights one such area – the green edges fall below the threshold but are not lying on the surface of the world.

Dense 3D data required The graph merging process described in Section 5.6 relies on matching pixels from image to image. If there is noise in the disparity data, or gaps due to poorly textured areas, then this matching is prone to failure. The occupancy grid maps used in the next chapter overcome this by maintaining a probabilistic measure of occupancy in a voxel grid which is more robust to noisy measurements.

Stray edges Figure 5.14(b) is an image taken from a simulated environment. The camera has followed the green trajectory to resolve a rift and perform exploration. The red edges indicate high weight edges which remain after the sensor has been in a position where it was in front of them, viewing a surface behind them. Stray edges such as these occur because the visibility check being performed is effectively trying to intersect two idealised lines – the high weight edge, and the ray from the sensor.

With limited precision reprojection we can catch most of these, and many of the others can be removed using heuristics – if a high weight edge has no neighbouring high weight edges then perhaps we should remove it. This is not a satisfying or robust solution.

5.9.1 Conclusions

Viewing the world as consisting of a single continuous surface and storing our map as a graph structure overlaid on this surface is a compelling image in terms of identifying exploration targets.

The results in this section are not exhaustive because although the idea was crisp, various difficulties (as enumerated above) motivated a parameter-free approach which is more robust. Specifically we need to explicitly model free-space and occupied space for camera trajectory planning, and we need a map structure which can cope with noisy sensor readings and pose estimates.

The idea of filling in holes in the surface of the world, and of using these rifts to drive exploration will be continued in the next chapter with the development of a parameter-free exploration algorithm based on solving a partial differential equation across the freespace of the map.

Chapter 6

Workspace Exploration with Continuum Methods

6.1 Introduction

The implementation of the graph-based exploration algorithm in Chapter 5 was not robust to noise in pose estimates, sensor data, or to sparse 3D data. It relied on setting a number of arbitrary parameters, and thus motivated the development of a more robust algorithm which is parameter-free.

This chapter describes the development and implementation of an exploration algorithm based on solving a boundary value constrained partial differential equation (PDE). This solution yields a scalar field which is guaranteed to have no local minima and is trivially transformed to a vector field in which following streamlines causes provably complete exploration of the environment. The algorithm is parameter-free and is applicable to any 3D sensor modality (laser, stereo, Kinect, etc.) with accurate pose estimates. It unifies the two stages of exploration, goal selection and path planning, in full 6 degrees of freedom (6DoF), and runs in real-time. We present empirical data showing that it outperforms (produces shorter paths) than the oft-used

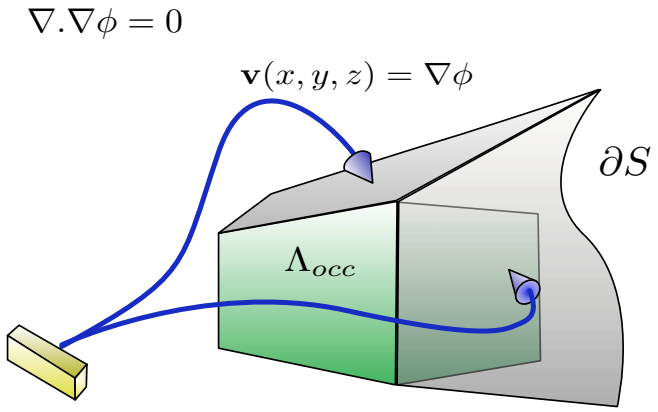


Figure 6.1: Moving a camera to explore the world. A solution to Laplace's equation, a harmonic function ϕ, is calculated over the domain of known free space. The boundary of free-space – ∂S – is either occupied by an obstacle or is a target for exploration as beyond it lies an unexplored region. The camera is guided to view these unexplored regions by following the gradient of $\nabla\phi$ – two such streamlines are shown.

information gain frontier exploration methods.

The success of the algorithm depends on solving Laplace's equation over a domain defined by the environment explored thus far. Solutions to Laplace's equation are known as harmonic functions and have no minima (or maxima) and we use this fact to guarantee continued and complete exploration. Applying appropriate boundary conditions to the problem yields a scalar field ϕ which satisfies Laplace's equation and we can use gradient descent in $\nabla\phi$ to reach exploration frontiers.

This chapter is structured as follows. First we present an informal analogy to a physical fluid process which would give rise to the exploratory behaviour we will demonstrate. We continue by describing the theoretical foundations of this work, namely the link between Laplace's equation and the properties of an inviscid, irrotational and incompressible fluid flow and how this relates to the analogy presented.

The underlying map structure, a 3D occupancy grid, is introduced, and we discuss how to map Laplace's equation, a continuous partial differential equation, onto a discrete voxel grid. We then solve the discretised version of Laplace's equation using

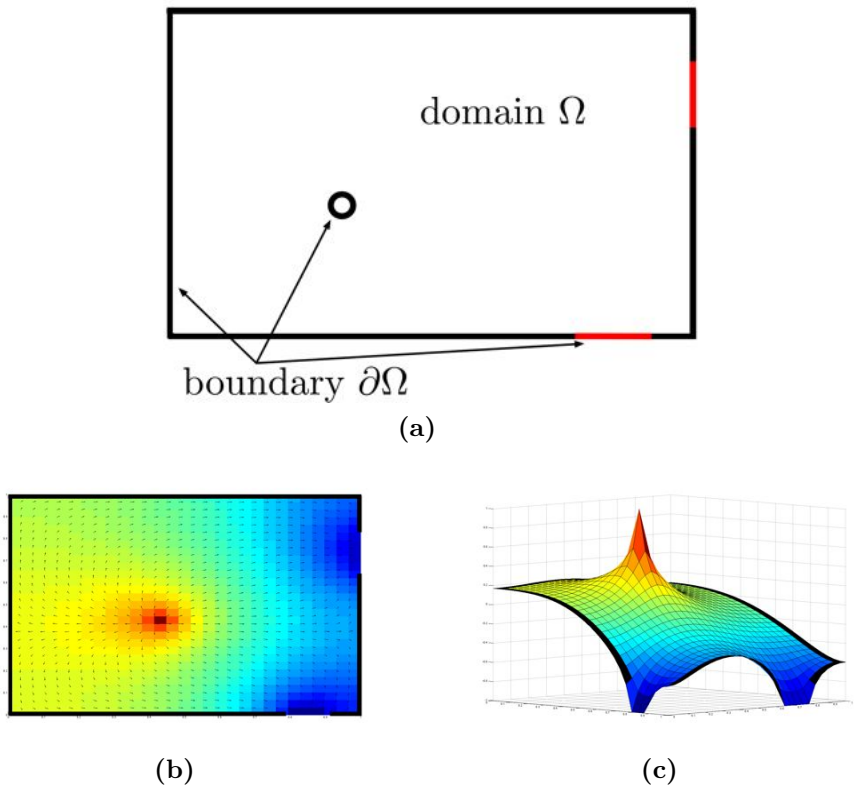


Figure 6.2: PDE Boundaries. An example environment is shown in (a). Black lines are walls: obstacles impenetrable to fluid flow. The red sections are exploration frontiers: boundaries between known free-space and unexplored space. The PDE domain Ω covers this free-space and is surrounded by a closed surface, the boundary $\partial\Omega$. Applying appropriate boundary conditions (for details see Section 6.2.5) and solving for ϕ (see Section 6.4.1) we obtain the scalar fields shown in (b) and (c).

iterative finite differences, and show how following streamlines through the resulting scalar potential field will lead us to an unexplored frontier.

Results are then presented showing the performance of our implementation and we show empirically that it performs better than the oft-used frontier-based information gain approaches. We have performed experiments both in simulated environments (which allow quick iterative testing and repeatability) and real-world results gathered using the Europa robot, described earlier in Chapter 4.

Part of this work was presented as "Choosing Where To Go: Complete Exploration With Stereo" at the IEEE International Conference on Robotics and Automation (ICRA) in Shanghai, China, in May 2011 [110].

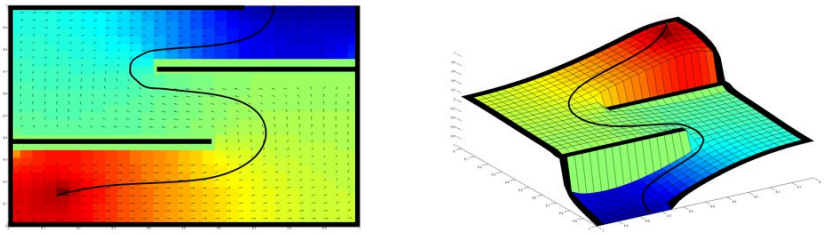


Figure 6.3: Solved field. Laplace's equation solved in a zig-zag environment. The red peak is the current robot position and the curved black streamline leads from this position down to the frontier, which is fixed at a low potential (shown in dark blue).

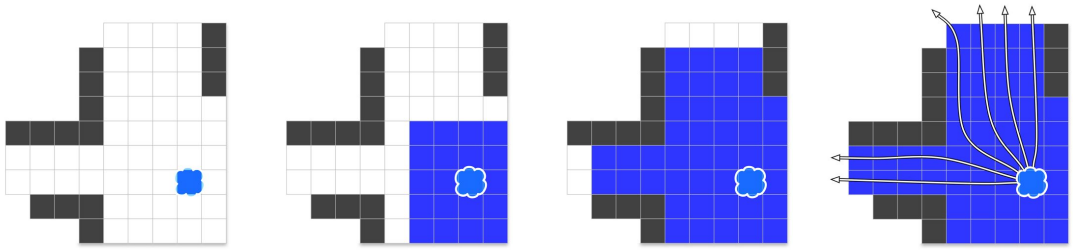


Figure 6.4: Informal analogy to exploration. An informal analogy to the exploration algorithm presented. Imagine that the robot is a source of fluid – a tap which continuously pumps water into the environment. Obstacles are impenetrable to this water and it can only leave the workspace by dissipating out across frontiers. In the steady state this will result in a flow field with streamlines emanating from the robot position and terminating at frontiers.

6.1.1 Physical Analogy

Our method leverages the properties of harmonic solutions to Laplace's equation but before describing this formalism in detail it is helpful to present an informal analogy to what we are proposing.

Consider a sealed room which contains a tap – a fluid source which pumps out water, filling the room. With nowhere for the water to go the internal the pressure inside the room will grow, but if we introduce an open door the water will flow out through it (the pump increases the flow rate to deal with the water being lost). Add a window to one of the walls and we now have two exit paths for the water. It will flow out through both, but at different rates depending on the size of the aperture

through which it is leaving the room. Imagine following a single water molecule from the tap as it follows the streamline with largest initial velocity. This streamline will invariably terminate at one of the exits – the door or the window. In the steady state flow field we can visualise many such streamlines emanating from the fluid source.

Figure 6.4 show this idea pictorially. In this example the black squares can be thought of as the walls of the room, and the edges of the grid as fluid sinks (where the fluid can exit the system). Figures 6.4a to 6.4c show the water spreading into the environment, and Fig. 6.4d shows some example streamlines in the steady state flow field.

We would like to take this idea and apply it to the problem of robot exploration. Consider a robot with a known pose in a partially explored environment. There are some areas which have been fully explored, some areas which are marked as obstacles, and some areas which are yet to be seen. The boundaries between explored and unexplored areas are known as frontiers. If we replace our robot with a fluid source, and the frontiers with fluid sinks (and specify that obstacles are impenetrable to water), then we can see how following a streamline in the steady state flow field will lead the robot smoothly to a new frontier.

This informal analogy provides us with an intuition as to how our exploration algorithm will behave. We can see that there can be no local minima or maxima in the flow field as the water must exit the system and cannot accumulate or stagnate. Secondly, the effect of frontier boundaries on the streamlines emanating from the robot position are a function of both size and distance – a large boundary close to the robot will have a greater effect than a small boundary at a large distance.

This analogy makes us think of trying to solve a steady state flow problem over the known workspace. In the absence of an analog to viscosity, this collapses down to finding a solution to Laplace's equation [1]. All such solutions are known as harmonic

[1] the inclusion of a viscosity term would further amplify the effect of known obstacles

functions and have properties that are advantageous to our cause: they have no local minima and hence streamlines emanating from the sensor position are guaranteed to lead us to exploration boundaries.

6.2 From fluid flow to Laplace's equation

"When a flow is both frictionless and irrotational, pleasant things happen"

Frank White [133]

The informal fluid flow analogy leads us to the field of fluid mechanics. When analysing fluid motion there are broadly speaking two approaches The **control volume approach** seeks to determine gross effects over a finite bounded region e.g. forces acting on a structure or flow rate through a cross section. Alternatively the approach of **differential analysis** is to obtain a detailed flow pattern at every point in a region of interest e.g. the pressure distribution inside a container It is this second approach, differential analysis, which we will look at in more detail.

This fluid problem will be solved across three-dimensional space, $\mathbb{R}^3$. Specifically we are interested in finding a vector field which describes fluid velocity. A vector field over $\mathbb{R}^3$ is a function which associates a 3-element vector to every point in the function domain, $f_v : \mathbb{R}^3 \to \mathbb{R}^3$. In the case of fluid flow the range of f_v describes the velocity of the fluid at each point in $\mathbb{R}^3$.

In general this velocity field is a function of position and time, and has three components, u, v, and w each of which is a scalar field (a function which maps from $\mathbb{R}^3$ to a scalar value, $f_s : \mathbb{R}^3 \to \mathbb{R}$):

$$V(x, y, z, t) = \mathbf{i}u(x, y, z, t) + \mathbf{j}v(x, y, z, t) + \mathbf{k}w(x, y, z, t) \tag{6.1}$$

The velocity field is arguably "the most important variable in fluid mechanics" [133] and solving for the flow field is in many cases equivalent to solving the full fluid flow problem as many other properties (pressure, temperature) can be derived from it.

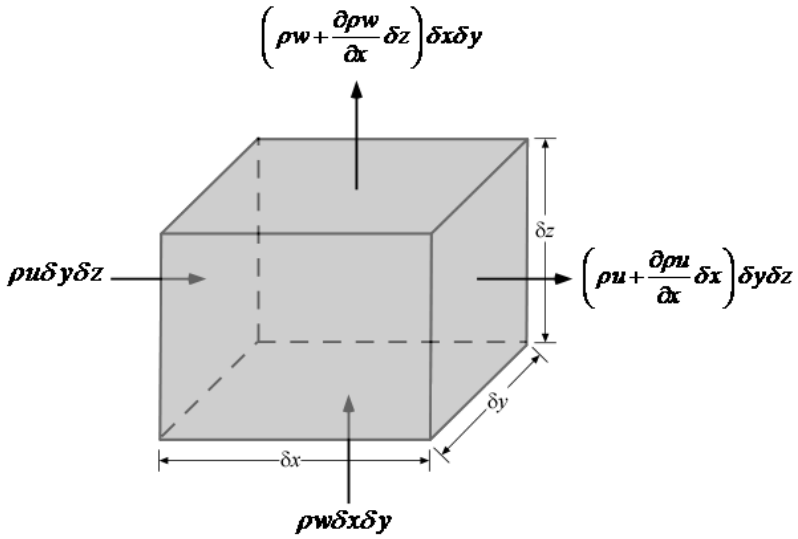


Figure 6.5: Conservation of mass in fluid flow. An infinitesimally small cube of dimensions $[dx, dy, dz]$ used to demonstrate the conservation of mass principle which leads to the important continuity equation. The velocity flow field components u and w are shown (flow in the y direction is omitted for clarity). ρ is the fluid density. Image from [106].

6.2.1 Continuity Equation

Consider Fig. 6.5. This is an infinitesimally small cube of dimensions $[\delta x, \delta y, \delta z]$ which is used as a control volume in this analysis. The velocity flow field components in the x and z directions are shown (flow in the y direction is omitted for clarity). ρ is the fluid density.

A fundamental conservation principle in fluid mechanics is that of mass conservation which states that the mass of a closed system must remain constant over time. Relating this principle to Fig. 6.5 we can say that the net mass flow through the volume must be zero or in other words that flow in must equal flow out. The mass contained in the volume at some time t is

$$m = \rho \delta x \delta y \delta z \tag{6.2}$$

and the rate of change of mass is simply

$$\frac{\partial m}{\partial t} = \frac{\partial \rho}{\partial t}\delta x \delta y \delta z \tag{6.3}$$

The net rate of change of mass must be zero to satisfy the conservation of mass principle. We can calculate this by summing Eq. (6.3) with the spatial derivatives $\frac{\partial m}{\partial x}$, $\frac{\partial m}{\partial y}$, and $\frac{\partial m}{\partial z}$.

We assume that all fluid properties are uniformly varying functions of time and position (for justification see [133]) and thus if the left face of the control volume has a value of ρu then the right face will have the value $\rho u + (\frac{\partial \rho u}{\partial x})\delta x$, as shown in Fig. 6.5. The net flow along the x axis is therefore

$$\frac{\partial m}{\partial x} = \left[\rho u + \frac{\partial \rho u}{\partial x}\delta x\right]\delta y \delta z - \rho u \delta y \delta z \tag{6.4}$$

$$= \frac{\partial \rho u}{\partial x}\delta x \delta y \delta z \tag{6.5}$$

The conservation of mass principle says that summing Eq. (6.4) with the equivalent terms in the y and z directions along with Eq. (6.3) must result in a net change of zero:

$$\frac{\partial m}{\partial t} + \frac{\partial m}{\partial x} + \frac{\partial m}{\partial y} + \frac{\partial m}{\partial z} = 0 \tag{6.6}$$

$$\frac{\partial \rho}{\partial t}\delta x \delta y \delta z + \frac{\partial \rho u}{\partial x}\delta x \delta y \delta z + \frac{\partial \rho v}{\partial y}\delta x \delta y \delta z + \frac{\partial \rho w}{\partial z}\delta x \delta y \delta z = 0 \tag{6.7}$$

and the element volume $\delta x \delta y \delta z$ cancels leaving us with

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} + \frac{\partial \rho w}{\partial z} = 0 \tag{6.8}$$

Equation (6.8) is known as the *continuity equation* because of the assumption that density and velocity are continuous functions. The continuity equation is more

commonly seen in the more compact form

$$\frac{\partial \rho}{\partial t} + \nabla.(\rho \mathbf{V}) = 0 \tag{6.9}$$

where $\mathbf{V}$ is the vector velocity field (Eq. (6.1)) and ∇ is the vector gradient operator, which in $\mathbb{R}^3$ is

$$\nabla = \frac{\partial}{\partial x} + \frac{\partial}{\partial y} + \frac{\partial}{\partial z} \tag{6.10}$$

6.2.2 Incompressibility

A very common assumption in fluid mechanics is that of *incompressible flow* – that is density variation is negligible inside an infinitesimal control volume, or that ρ is some constant value c. Under this assumption we see that Eq. (6.9) simplifies further as the density differential disappears and ρ cancels, leaving

$$\left.\frac{\partial \rho}{\partial t}\right|_{\rho=c} + \nabla.(\rho \mathbf{V}) = 0 \tag{6.11}$$

$$\nabla.\mathbf{V} = 0 \tag{6.12}$$

6.2.3 Irrotationality

The second assumption we make is that the flow field is *irrotational*. An irrotational flow is a flow in which the local rotation of fluid elements is zero – such a field is conservative and arises in the scalar potential problems we are considering [133].

A vector field is irrotational if it has zero curl, that is

$$\nabla \times \mathbf{V} = 0 \tag{6.13}$$

From vector calculus we can make use of the identity

$$\nabla \times (\nabla \phi) = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ \partial_x & \partial_y & \partial_z \\ \partial_x \phi & \partial_y \phi & \partial_z \phi \end{vmatrix} = \mathbf{0} \tag{6.14}$$

to conclude that every irrotational vector field $\mathbf{V}$ can be written as the gradient of a scalar potential field

$$\mathbf{V} = \nabla \phi \tag{6.15}$$

Substituting this knowledge into Eq. (6.11) leads us to

$$\nabla . \mathbf{V} = 0 \tag{6.16}$$

$$\nabla . \nabla \phi = 0 \tag{6.17}$$

$$\nabla^2 \phi = 0 \tag{6.18}$$

which expanded in $\mathbb{R}^3$ is

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2} = 0 \tag{6.19}$$

Equation (6.19) is Laplace's equation and by solving for ϕ and taking the gradient we will be able find streamlines in the velocity flow field – this is the key to mathematically reproducing the situation described by analogy in Section 6.1.1.

6.2.4 Laplace's equation

We have established that solving Laplace's equation is key to finding the fluid velocity field that we will show will govern our exploratory behaviour. Let us examine this

equation in some more detail before describing the method used to solve it.

The general form of a 2nd order partial differential equation with two independent variables

$$A\frac{\partial^2\phi}{\partial x^2} + 2B\frac{\partial^2\phi}{\partial x\partial y} + C\frac{\partial^2\phi}{\partial y^2} + D\frac{\partial\phi}{\partial x} + E\frac{\partial\phi}{\partial y} + F = 0 \tag{6.20}$$

where A, B, C, D, E, F are functions of the variables x and y. PDEs can be classified according to the properties of the matrix of coefficients

$$Z = \begin{bmatrix} A & B \\ B & C \end{bmatrix} \tag{6.21}$$

and are classed as *elliptic* if $det(Z) > 0$. Laplace's equation is one of the simplest non-trivial forms of Eq. (6.20) and can be reached by setting $A = 1$, $C = 1$, and $B = D = E = F = 0$. It is then clear that

$$det(Z) = \begin{vmatrix} A & B \\ B & C \end{vmatrix} = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix} = 1 \tag{6.22}$$

and thus we conclude that Laplace's equation is indeed an elliptic PDE. This is a useful result as it means we can be assured that any solution to Laplace's equation will have certain properties common to elliptic PDEs. One such property is the maximum principle which we can use to guarantee no minima in our solution.

6.2.4.1 The Maximum Principle

One family of solutions to Laplace's equation is simply $\phi = c$ where c is some constant value. By setting our boundary conditions appropriately we will ensure that these never occur in practice.

The so called *weak maximum principle* [74] is a property of solutions to certain elliptic equations which states that maxima or minima in ϕ must appear on the

boundary $\partial\Omega$ (but could equalled by values in the interior).

The *strong maximum principle* [74] takes this further and states that minima and maxima must appear on $\partial\Omega$, and that if either of these extrema appears in the interior as well then the function must be uniformly constant, $\phi = c$.

6.2.4.2 Harmonic functions

Solutions to Laplace's equation are known as *harmonic functions* [5]. Harmonic functions are a class of function to which we can apply the *strong maximum principle*. Stated formally in [5]:

> Suppose Ω is connected, ϕ is real valued and harmonic on Ω, and ϕ has a maximum or a minimum in Ω. Then ϕ is constant.
>
> *Corollary:* Suppose Ω is bounded and ϕ is a continuous real-valued function that is harmonic on Ω. Then ϕ attains its maximum and minimum values over Ω on $\partial\Omega$.

The intuition as to why this is true for Laplace's equation can be obtained by considering the three differential terms: $\frac{\partial^2\phi}{\partial x^2}$, $\frac{\partial^2\phi}{\partial y^2}$, and $\frac{\partial^2\phi}{\partial z^2}$. At a local extremum in Ω, the three terms would share the same sign – positive at a minimum, negative at a maximum – and thus not sum to zero.

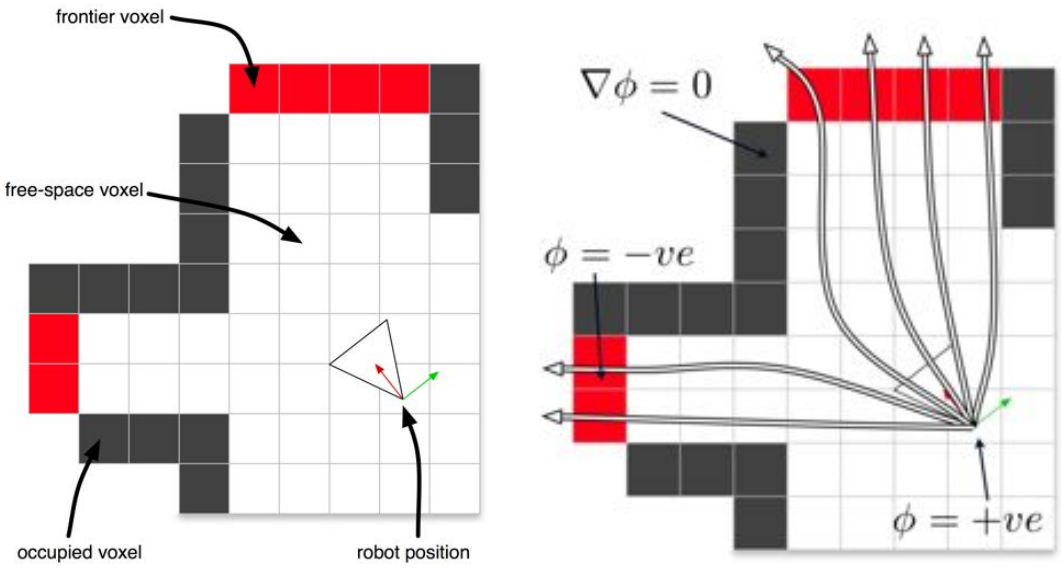
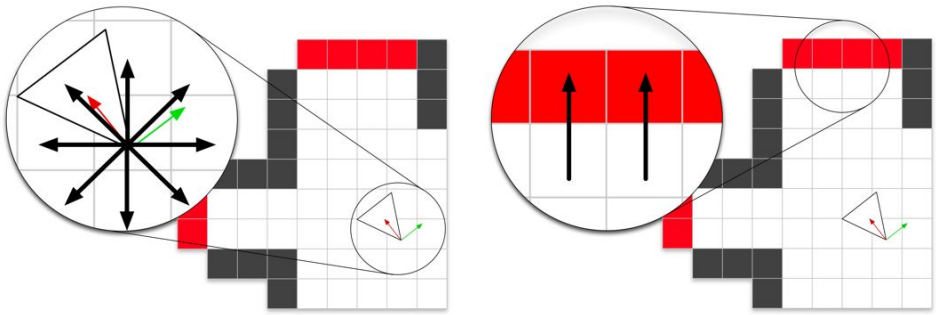


Figure 6.6: From occupancy grid to fluid flow. If we have framed the problem correctly and boundary values are specified at the robot position, at frontiers, and at obstacles, then on solving Laplace's equation across the occupancy grid we expect to find streamlines in $\nabla\phi$ which follow roughly the same pattern as those seen in our informal analogy from Section 6.1.1.



(a) Robot boundary **(b)** Frontier boundary

Figure 6.7: Dirichlet boundary conditions. The value of ϕ at the robot position is set with a Dirichlet boundary condition to a fixed positive value, while the frontiers are set to fixed negative values. This ensures streamlines will flow down from the robot to frontiers.

6.2.5 Boundary Conditions

The solution to Laplace's equation is constrained by a set of *boundary conditions* which specify the values ϕ or $\nabla\phi$ must take on the boundary of the problem domain. We set three different boundary conditions on different parts of the domain: the robot position, frontiers, and obstacles.

6.2.5.1 Dirichlet boundary conditions

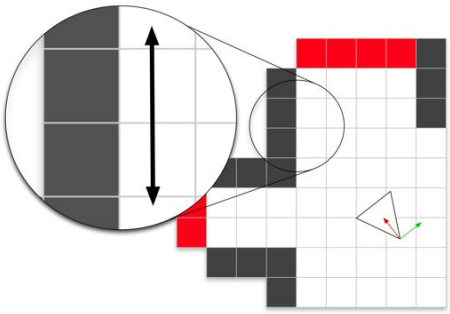
A Dirichlet, or first-type, boundary condition specifies that ϕ must take a specific constant value on the boundary

$$\phi_{\delta\Omega} = c \tag{6.23}$$

Such boundary conditions are used to guarantee that a streamline starting at the robot position will terminate at a frontier voxel and are set as follows:

Robot position ϕ is set to a fixed positive value $\phi_r = +1$

Frontiers ϕ is set to a fixed negative value $\phi_f = -1$



(a) Obstacle boundary

Figure 6.8: Neumann boundary conditions. Obstacle boundaries are set using Neumann boundary conditions: the derivative of ϕ is set to zero across obstacles to ensure that the velocity field runs parallel to walls.

6.2.5.2 Neumann boundary conditions

A Neumann, or second-type, boundary condition specifies that the first derivative of ϕ must take a specific value on the domain boundary

$$\nabla\phi_{\delta\Omega} = c \tag{6.24}$$

We use Neumann boundary conditions to force the vector field to run parallel to obstacles. To do this, the gradient normal to an obstacle boundary is set to zero $\nabla\phi = 0$.

6.2.5.3 Application of boundary conditions

The 2D exploration of [100] makes use of Dirichlet boundary conditions only. As discussed in both [35] and [50], using only Dirichlet boundaries results in very flat fields with no appreciable gradient except for the local regions surrounding start and goal points. Neumann boundaries, or a combination of Neumann and Dirichlet, result in significant gradients throughout the solution domain. This is shown in Fig. 6.9.

Figure 6.10 illustrates the effect of these boundary conditions when applied to real data. It is difficult to illustrate this in 3D and so we show a 2D slice through

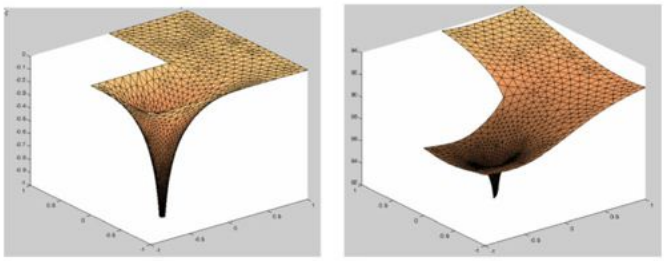
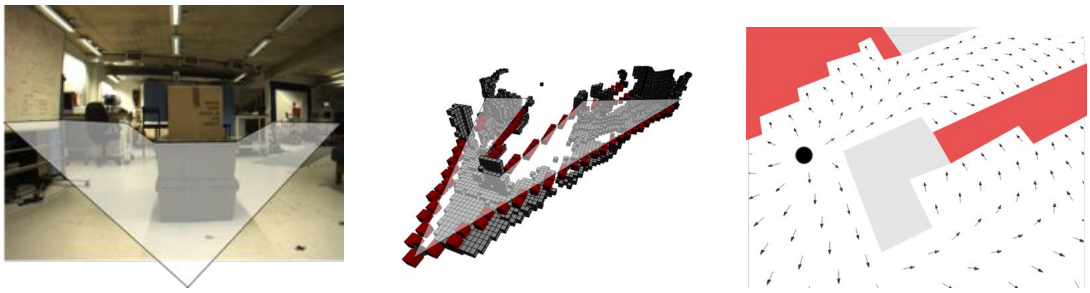


Figure 6.9: Dirichlet and Neumann boundaries. A comparison of the potential fields produced when using (a) Dirichlet boundaries only and (b) Neumann boundary conditions. As discussed in both [35] and [50], using only Dirichlet boundaries results in very flat fields with no appreciable gradient except for the local regions surrounding start and goal points. Neumann boundaries, or a combination of Neumann and Dirichlet, result in significant gradients throughout the solution domain. Image from [35].



(a) Input image from stereo camera

(b) Occupancy grid and exploration boundaries

(c) Harmonic function gradient, $\nabla\phi$

Figure 6.10: Exploration in 2D. An example left image from the stereo camera is shown in (a). This is processed, producing a depth image which is used to fill an occupancy grid, shown in (b). 2D exploration is performed in a single plane of this occupancy grid – exploration boundaries are shown in red. (c) shows the gradient of a harmonic function satisfying $\nabla^2\phi = 0$ over this domain. The sensor position is a black circle, unexplored space in red, and occupied space in grey.

the map. Fig. 6.10(a) and Fig. 6.10(b) show the input image and a horizontal slice through the resulting occupancy grid. The exploration boundaries are shown in red – all voxels lying within this boundary are free space, those outside are unknown. After applying boundary conditions and calculating a harmonic solution over this grid we can display $\nabla\phi$ as a vector field, shown in Fig. 6.10(c).

6.3 Implementation

The formulation of Laplace's equation as described previously in Section 6.2.4 is applicable to a continuous problem domain. There are a couple of issues which must be addressed before we can apply it to directly controlling exploration of a mobile robot.

Any map implementation we choose will be, at some level, a discrete model of the world. Ultimately this is enforced by the digital nature of computers, but at a more practical level we sacrifice resolution for real-time performance and the result is a quantification of the real world into discrete containers. We must therefore find a method to apply a continuous PDE to a discretised problem domain – specifically how do we map Laplace's equation to a partially explored occupancy grid map?

Secondly we must be able to explicitly differentiate between free-space, unexplored regions, and obstacles in the map. The boundary values of the PDE are dependent on this and so our options for map representations are limited to those which can make this distinction easily – as previously discussed we will use a 3D occupancy grid.

6.3.1 Occupancy grid to Laplace

Our knowledge about the state of the world is encoded in an octree based occupancy grid. This is described in detail in Chapter 3, but will be recapped briefly: An occupancy grid is a regular discretisation of space into voxels (volumetric pixels), each of which has an associated binary random variable. This random variable maintains an occupancy probability – what is the probability that a given voxel has an obstacle in it?

Voxels can be classified as one of three states by thresholding this variable: occupied, free space, or unknown and we refer to the sets of these voxels as $\mathbf{\Lambda}_{occupied}$,

$\mathbf{\Lambda}_{free}$, and $\mathbf{\Lambda}_{unknown}$ respectively. A frontier voxel is one which lies on the boundary between explored and unexplored space – a free space voxel with a neighbouring unknown voxel. The set of frontier voxels is denoted $\mathbf{\Lambda}_{frontier}$. The volume of explored space, $\mathbf{\Lambda}_{free}$, is enclosed by a boundary $\partial S = \mathbf{\Lambda}_{occupied} + \mathbf{\Lambda}_{frontier}$. Beyond this boundary exist only unknown voxels: $\mathbf{\Lambda}_{unknown}$.

The criterion for exploration termination is the elimination of all frontier voxels, *i.e.* $\mathbf{\Lambda}_{frontier} = \emptyset$. Assuming that our sensor can resolve 3D points at a sufficient density over the surface of the world, then this equates to complete exploration, up to the resolution of the occupancy grid. The key to successful exploration is to reposition our sensor such that ∂S will be expanded outwards into unknown space, increasing $\mathbf{\Lambda}_{free}$ and reducing $\mathbf{\Lambda}_{unknown}$.

6.3.1.1 Identifying Frontiers

Frontiers, the boundaries between explored and unexplored space, can now be identified in our occupancy grid. A voxel is defined to be a frontier voxel if

- it is free-space – obstacles cannot be frontiers
- it lies on the outer edge of a set of explored voxels and is a neighbour of at least one unexplored voxel

To identify all frontiers we inspect each free-space voxel and examine its 6 immediate neighbours $(x \pm 1, y \pm 1, z \pm 1)$ – if at least one is an unexplored voxel then this voxel is a frontier.

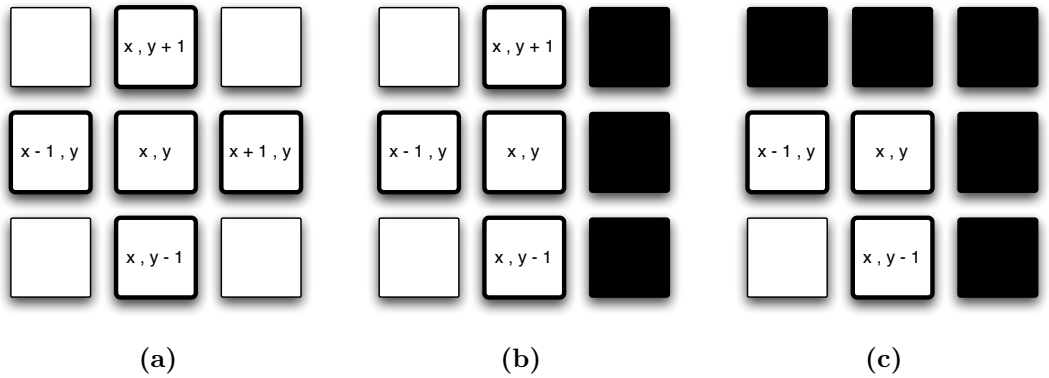


Figure 6.11: Finite difference method in 2D. FDM in two dimensions: The five-point stencil for the finite difference calculations is shown in (a). A couple of special cases are shown in (b) and (c). These situations require fictitious mirror values to enforce the Neumann boundary conditions.

6.4 Solving Laplace

So far we have described a process which begins with the acquisition of 3D data; the integration of this data into an occupancy grid map; and the identification of frontier voxels in this map. We now consider the problem of mapping Laplace's equation to this voxel grid, and solving for ϕ, the scalar potential field which will be analysed to yield exploration targets.

As we saw in Section 6.2, the potential flow equations reduce to solving Laplace's equation with assumptions of irrotational and incompressible fluid flow. To solve Laplace's equation and obtain the field ϕ, we turn to the field of computational fluid dynamics (CFD). CFD is concerned with using numerical analysis to compute approximate solutions to fluid flow problems, and we will use one of its many methods to tackle our situation, namely the method of iterative finite differences.

6.4.1 Iterative methods

To analytically solve Laplace's equation is not feasible for the large 3D irregular domains we are dealing with, but the occupancy grid structure lends itself perfectly

to the application of the finite difference method (FDM) [2, 75]. In FDM the function ϕ is represented by a discrete value in each grid cell, and derivatives are approximated by differences between neighbouring cells.

To apply FDM to Laplace's equation a finite difference approximation to the second derivative of ϕ is required. Consider Fig. 6.11a. It shows a point in a 2D grid with its four neighbours in the x and y directions, separated by h (the distance between voxel centres). The value of ϕ at this point is denoted by $\phi_{x,y}$. Taking a Taylor expansion around $\phi_{x,y}$ in the x direction leads us to the second derivative approximation:

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi_{x+1,y} - 2\phi_{x,y} + \phi_{x-1,y}}{h^2} + O(h^2). \tag{6.25}$$

Combining Eq. (6.25) with the corresponding equation in y and the two dimensional version of Laplace's equation Eq. (6.19) we get:

$$\phi_{x,y} \approx \frac{1}{4}(\phi_{x+1,y} + \phi_{x-1,y} + \phi_{x,y+1} + \phi_{x,y-1}). \tag{6.26}$$

This tells us that if ϕ satisfies Laplace's equation then at any point in the problem domain, i.e. at the centre of any free-space voxel, then the value of ϕ at that point is simply the average of the 4 surrounding voxels.

Equation (6.26) is trivially extended into three dimensions, and it is this sum over 6 neighbouring values which is used in our work:

$$\phi_{x,y,z} \approx \frac{1}{6}(\phi_{x\pm1,y,z} + \phi_{x,y\pm1,z} + \phi_{x,y,z\pm1}). \tag{6.27}$$

Consider again Fig. 6.11. In (b) and (c) we see that we cannot use Eq. (6.26) as is – required cells such as $\phi_{x+1,y}$ lie outside of the PDE domain. The solution is to use fictitious grid points lying beyond the boundary which, if they existed, would result in the specified derivative. The zero derivative Neumann boundary conditions

mean these fictitious values are simply a reflectance of the interior values and so the update equation for $\phi_{x,y}$ in Fig. 6.11(c) becomes:

$$\phi_{x,y} = \frac{1}{4}(2\phi_{x-1,y} + 2\phi_{x,y-1}), \tag{6.28}$$

and similarly in higher dimensions.

The solution to this bounded PDE is now calculated by iteratively updating each cell. At each timestep Eq. (6.27) is applied to each voxel. When updating $\phi_{x,y,z}$, do we use the neighbours' values from the previous timestep, or the values from this iteration if already computed? Jacobi iteration uses values from the previous timestep, but faster convergence can be achieved if we use the latest available values – the Gauss-Seidel method.

We terminate based on the maximum relative error at each timestep. The relative error for voxel x, y, z at timestep i is defined as:

$$\epsilon(x, y, z) = \left| \frac{\phi^i_{x,y,z} - \phi^{i-1}_{x,y,z}}{\phi^{i-1}_{x,y,z}} \right|. \tag{6.29}$$

Termination occurs when the largest relative error drops below a given precision threshold. A precision of $\epsilon_t = 10^{-4}$ was found to be acceptable, and this process is shown in Fig. 6.12.

6.4.2 Guiding Exploration in ϕ

Once Laplace's equation has been solved we wish to use the solution to drive our robot exploration algorithm. The scalar potential field, ϕ, surrounds the current robot position – the robot position itself is part of $\partial\Omega$, the boundary of the domain. The Dirichlet boundary condition at the robot position means that ϕ is fixed to a constant positive value, and we have a guarantee that the field values will monotonically

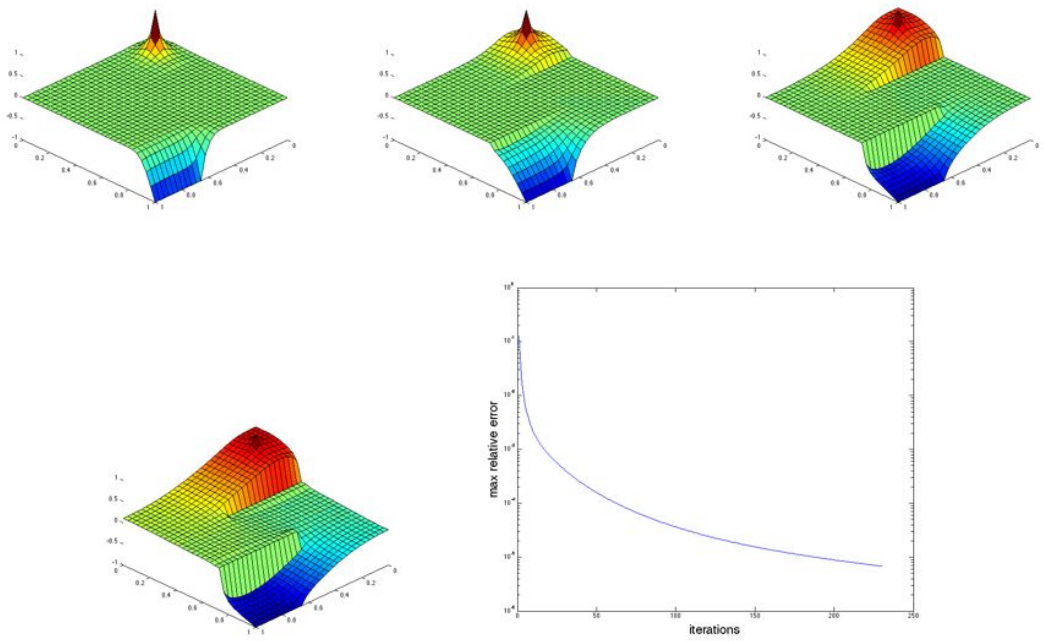


Figure 6.12: FDM termination. Termination of the iterative finite difference method occurs when the maximum relative error has dropped to a specified precision, in this case 10^{-5} which occurred after around 230 iterations on a 100x100 grid.

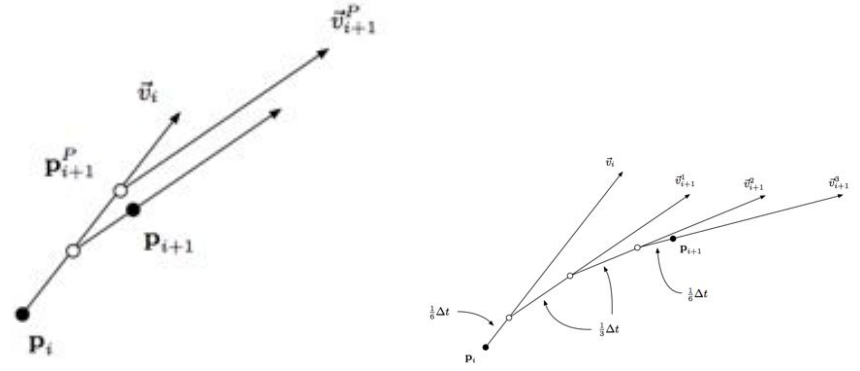
decrease with any small step $\delta\mathbf{x}$ away from this initial position.

If we consider the values of the vector gradient field, $\nabla\phi$, immediately surrounding the robot position and drop an imaginary water particle into the vector field at the point of maximum gradient it will flow through the vector field and exit the system at an exploration frontier. It is this streamline following behaviour that we will now capture algorithmically.

6.4.2.1 Streamlines

A streamline is defined as a curve through a fluid flow velocity vector field which is at all points tangent to the velocity vector

$$\frac{d\mathbf{p}_S}{ds} \times \nabla\phi(\mathbf{p}_S) = 0 \tag{6.30}$$



(a) **(b)**

Figure 6.13: Euler's method and 4th order Runge-Kutta. (a) depicts the improved version of Euler's method. Starting at $\mathbf{p}_i$, the standard Euler's method is used to predict $\mathbf{p}^P_{i+1}$ and the interpolated gradient at this point is used to better predict $\mathbf{p}_{i+1}$. This predictor-corrector method is bounded by $\mathrm{O}(\delta t^3)$ (compared to $\mathrm{O}(\delta t^2)$ for the standard Euler's method). (b) shows the 4th order Runge-Kutta method which is also a predictor-corrector method but which performs 3 predictions which are then combined to obtain the estimate for $\mathbf{p}_{i+1}$. Although slightly more computationally expensive, the 4th order Runge-Kutta method has errors bounded by $\mathrm{O}(\delta t^4)$. Both figures are adapted from [49].

where $\mathbf{p}_S(s)$ is a parameterised streamline, which originates at some point $\mathbf{p}_S(0) \in \phi$. Figure 6.3(b) shows one such streamline in which $\mathbf{p}_S(0)$ is the current robot position.

Given a vector field $\nabla\phi$ and a point $\mathbf{p}_S(0)$, how do we find the points which lie on the streamline originating here? It is worth remembering that the vector field is not continuous, rather it is defined at the voxel centres only.

6.4.2.2 Euler's Method

Consider a particle being dropped into the vector field $\nabla\phi$ at a point $\mathbf{p}$ as shown in Fig. 6.14(a). The values of $\nabla\phi$ are only defined at the grid points (shown as $\vec{v_{0,1}}$, $\vec{v_{1,1}}$ etc.). We can approximate the gradient, f, at a point $\mathbf{p} = [u, v]$ by performing a bilinear interpolation of the four surrounding values

$$f(\mathbf{p}) = (1-v)(1-u)\vec{v}_{0,0} + (1-v)u\vec{v}_{1,0} + v(1-u)\vec{v}_{0,1} + vu\vec{v}_{1,1} \tag{6.31}$$

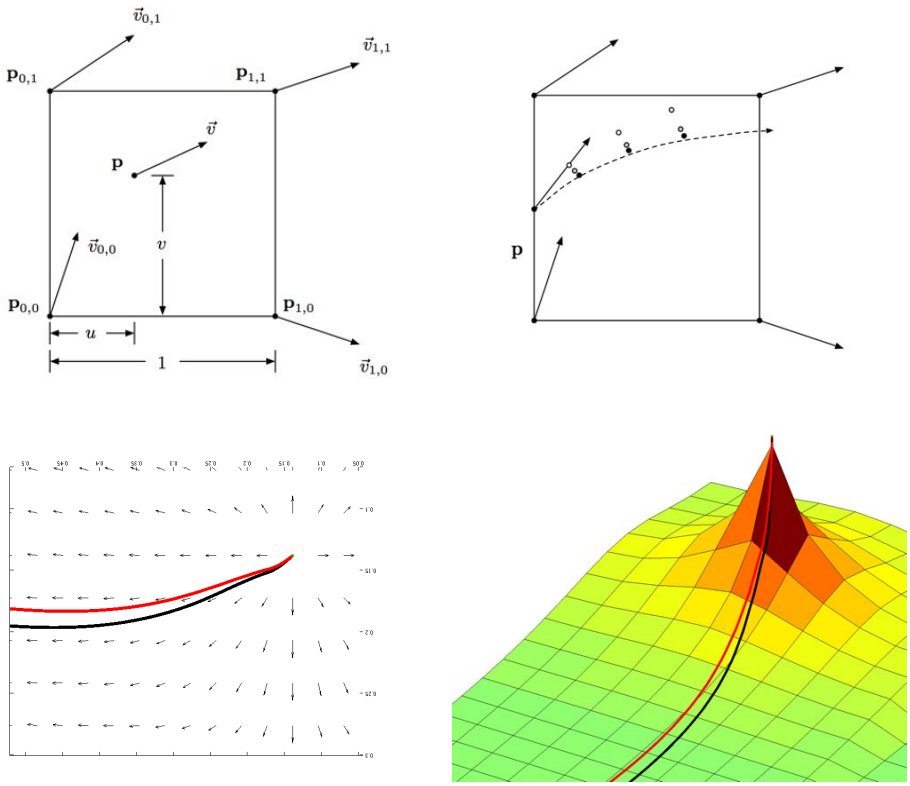


Figure 6.14: Finding streamlines by numerical integration. Tracing streamlines through a vector field by numerical integration. The field consists of a discrete grid of vectors and we wish to find a continuous curve through this field. (a) shows the starting point $\mathbf{p}$ of such a curve. Through bilinear interpolation (trilinear in 3D) we can obtain an estimate of the gradient at $\mathbf{p}$. (b) shows typical results for three numerical integration methods: Euler's method in white, improved Euler's method in grey, and black for 4th order Runge-Kutta. The dotted line is the true solution. (c) and (d) show the divergence of the improved Euler's method (black) and 4th order Runge-Kutta (red) on real data. (a) and (b) are adapted from [49].

where (u, v) is the position of $\mathbf{p}$ relative to $\mathbf{p}_{0,0}$.

In a single timestep h the particle will move a distance $hf(\mathbf{p})$ and so we can update

$$\mathbf{p}_{i+1} = \mathbf{p}_i + hf(\mathbf{p}_i) \tag{6.32}$$

This is Euler's method for numerical integration, and can give reasonable results locally. However it has an error bounded by $O(\Delta t^2)$ which results in quite rapid divergence. Figure 6.14(a) shows, in white, the values given by Euler's method

(compare to the dashed line which represents the true solution).

6.4.2.3 Improved Euler's Method

Euler's method can be improved quite easily by introducing the idea of the predictor-corrector method. Figure 6.13(b) shows the idea. We predict the position of the new point $\mathbf{p}_{i+1}^P$ using Euler's method, and then use the gradient at the predicted point to *correct* the original estimate. This improved estimate can be calculated as follows

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \frac{1}{2}hf(\mathbf{p}_i) + \frac{1}{2}hf(\mathbf{p}_{i+1}^P) \tag{6.33}$$

$$= \frac{1}{2}h(f(\mathbf{p}_i) + f(\mathbf{p}_{i+1}^P) \tag{6.34}$$

This method is substantially better, with an error bounded by $O(\Delta t^3)$, but we can go further with this predictor-corrector method and get better results by looking at the Runge-Kutta family of algorithms.

6.4.2.4 4th order Runge-Kutta

The Runge-Kutta family of algorithms uses a weighted combination of multiple predicted values and can be seen as a generalisation of the Euler methods. The most commonly used is known as the 4th order Runge-Kutta method or RK4 and uses 3 predicted values

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \tag{6.35}$$

$$\tag{6.36}$$

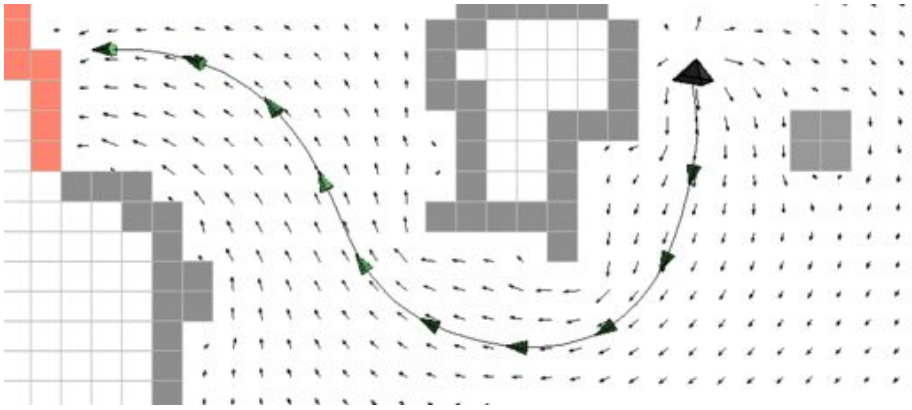


Figure 6.15: Streamline in 2D. An example of using the Runge-Kutta algorithm to find a streamline leading from the robot position to a frontier.

where the coefficients k_i are

$$k_1 = hf(\mathbf{p}_i) \tag{6.37}$$

$$k_2 = hf(\mathbf{p}_i + \frac{1}{2}k_1) \tag{6.38}$$

$$k_3 = hf(\mathbf{p}_i + \frac{1}{2}k_2) \tag{6.39}$$

$$k_4 = hf(\mathbf{p}_i + k_3) \tag{6.40}$$

RK4 has a total accumulated error bound of $O(\Delta t^4)$. The relative simplicity of implementation and the improved error bound leads us to use this algorithm to find streamlines through $\nabla\phi$. Figure 6.14(b) shows how RK4 much more closely matches the true streamline compared to the Euler or improved Euler methods.

6.4.3 Completion

How do we decide we've finished exploration? This is easy – there are no more frontier voxels in the occupancy grid. We can trivially and cheaply update the set of frontier voxels by iterating over the grid and inspecting immediate voxel neighbours. A free voxel with an unknown voxel neighbour is marked as a frontier, and if none exist in the map then exploration is complete.

6.5 Results

To evaluate the performance of our exploration algorithm we have performed a number of experiments both in simulation and on a real-world robot platform: the Europa robot described in Chapter 4.

6.5.1 Implemented exploration algorithms

We implemented a number of commonly used algorithms and these will now be described briefly. All of these algorithms are run over the same 3D occupancy grid maps. Most of the exploration code is common to all the algorithms – they differ in their method of selecting an exploration target, and of planning a path to that target. The pseudocode for the general exploration algorithm is as follows:

1. **Initialisation** The robot (real or simulated) is placed in a starting pose, and the various data structures are initialised

2. **Grab sensor data** Data from the sensor(s) is captured, processed, and integrated into the occupancy grid map

3. **Identify exploration target** Choose an exploration target using whichever algorithm is being used (see below)

4. **Plan path to target** Calculate the path to the chosen target using A^*across the occupancy grid, or the streamline method if using the potential algorithm

5. **Move robot along path** Robot or simulated sensor follows the chosen path to target

6. **Iterate** Repeat steps 1-5 until no frontier voxels remain in occupancy grid map

The various exploration algorithms used in step 3 are now described.

6.5.1.1 Random Frontier

One of the simplest approaches to exploration is to choose, at random, a frontier voxel to visit. Once selected, a shortest path to the target voxel is planned using the A* algorithm described in Section 5.5.3.3. This is an inefficient algorithm which would never be used in practice, but it provides a baseline against which we can measure the performance of superior algorithms. As would be expected, this algorithm produces long paths which zig-zag randomly across the map.

6.5.1.2 Closest Frontier

An improvement over random exploration is to choose the frontier voxel which is *closest* to the robot, as described in [137]. Again the A* path planner is then used to find the shortest path from the current pose to the target voxel.

The paths produced by this algorithm can be described as frontier-following – it will move to the closest frontier area and "chase it down" until completely resolved.

6.5.1.3 Information-gain

A number of random candidate poses, $[p_0, \ldots, p_i]$, are hypothesised in the known freespace of the map, and are ranked according to a cost function $f(p_j)$.

This cost function will include some measure of *information-gain* – that is, how much information might we expect to obtain if we moved our sensor to a given pose p_j? This is described in more detail in Chapter 2. We use the number of frontier voxels which we expect to image from a given pose as the measure of information gain, and hypothesise a number of poses equal to 5% of the number of freespace voxels currently in the occupancy grid.

The A* path planner is once again invoked to provide a shortest path through the occupancy grid.

6.5.1.4 Weighted Information-gain

This is identical to the information-gain approach, except that the cost function is modified to take into account path length to candidate poses. As introduced by [38] we score the candidate poses using the following function

$$g(q) = A(q) \exp(-\lambda L(q)) \tag{6.41}$$

where A(q) is a measure of the information gain expected at the q (the number of frontier voxels seen from a candidate pose), L(q) is the path length from the current pose to the candidate pose, and λ is a weighting factor which balances the cost of motion against the expected information gain. We set $\lambda = 0.2$ as a suitable weight as suggested by [38].

6.5.1.5 PDE Exploration

The algorithm described in this chapter. We solve Laplace's equation over a domain specified by the known freespace of the map, with boundaries defined by obstacles, frontiers, and the robot position. Streamlines emanating from the robot position lead us to frontiers so there is no need to invoke a separate path planner.

sensor resolution	100×100
sensor fov	75° horz
voxel dimension	$0.1 \times 0.1 \times 0.1$ m
number of runs per environment	30

Table 6.1: Common settings used to gather simulated exploration results

6.5.2 Simulated environments

We ran a number of experiments in simulated environments using the `Simulator` application described in Chapter 3.

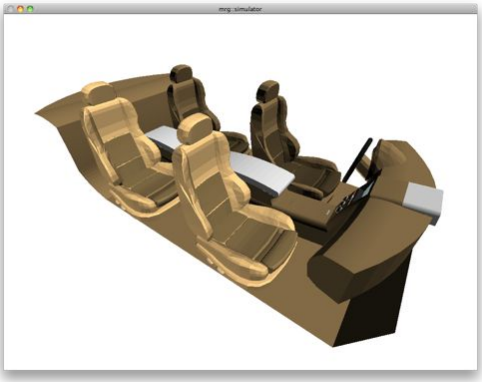
Each of the five exploration algorithms – random frontier, closest frontier, information gain (IG), weighted information gain (IG), and our potential method – were tested in six different simulation environments. The environments were chosen to provide a range of different obstacles, shapes, and sizes. They range from the interior of a car to an apartment.

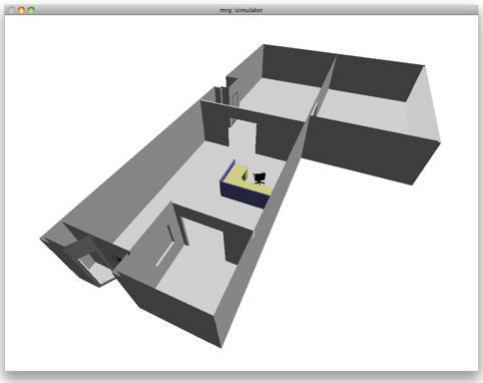
The five exploration algorithms were run 30 times per environment using the common settings shown in Table 6.1. Each run began with the starting pose being chosen at random from within the freespace of the environment, and exploration continued until there were no frontier voxels remaining in the occupancy grid map.

The pseudocode given previously at the start of this section describes the steps performed as exploration progressed. It should be noted that the initial pose is randomised each run (position and orientation) with the condition that it lie in free-space. The results of these experiments, and images of the test environments used, are shown over the next three pages.

Scenario 1: $7 \times 4 \times 4$ m

algorithm	mean path	std dev
random	22.34	4.1
closest	9.78	1.72
frontier	12.99	3.66
weighted IG	11.07	2.8
potential	9.3	2.39





Scenario 2: $10 \times 10 \times 5$ m

algorithm	mean path	std dev
random	87.02	24.86
closest	22.98	6.08
frontier	56.9	14.56
weighted IG	37.02	7.13
potential	21.43	3.96

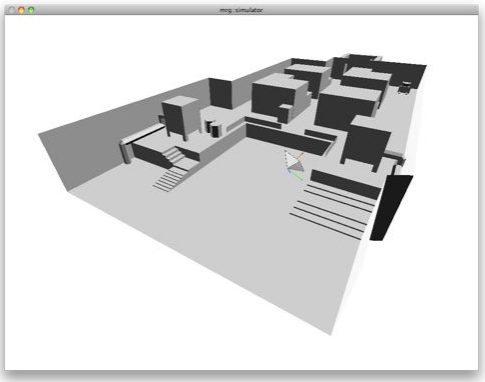
Scenario 3: $8 \times 8 \times 3$ m

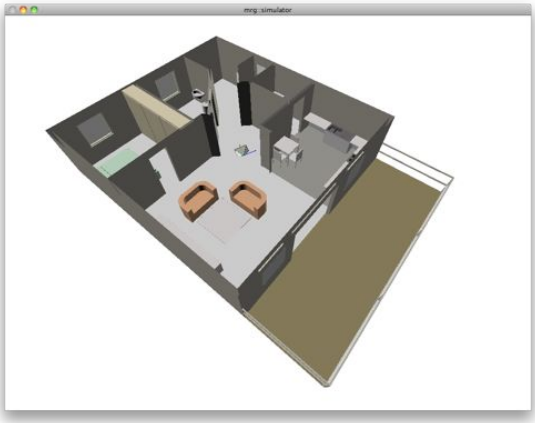
algorithm	mean path	std dev
random	36.44	10.8
closest	16.99	3.37
frontier	20.29	7.88
weighted IG	18.98	4.83
potential	15.29	4.36

Figure 6.16: Simulated exploration results

Scenario 4: $8 \times 18 \times 5$ m

algorithm	mean path	std dev
random	257.75	98.31
closest	57.7	18.5
frontier	108.99	31.85
weighted IG	65.04	24.18
potential	50.79	11.96





Scenario 5: $6 \times 8 \times 2$ m

algorithm	mean path	std dev
random	71.45	18.9
closest	34.0	7.97
frontier	40.84	10.55
weighted IG	38.23	7.16
potential	30.06	6.06

Scenario 6: $7 \times 9 \times 2$ m

algorithm	mean path	std dev
random	37.16	12.46
closest	23.5	6.65
frontier	25.79	7.67
weighted IG	24.88	10.09
potential	21.98	6.53

Figure 6.17: Simulated exploration results

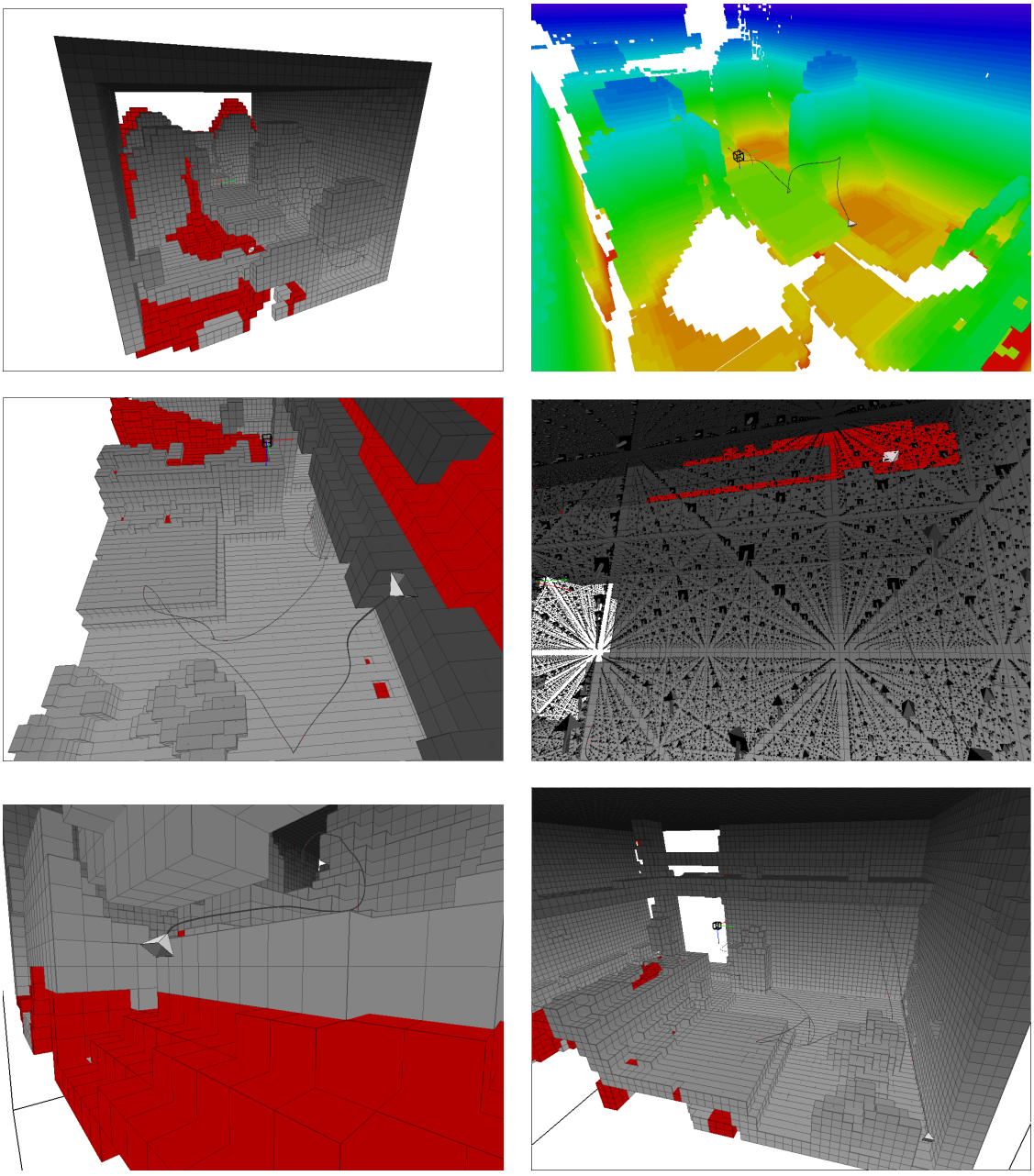


Figure 6.18: Snapshots of 3D exploration in simulated environments. Various examples of exploration taking place in 3D occupancy grids. The top right image shows the OctoMap rendering with colours mapped to height values. The other images show grey or black obstacle voxels, and red frontiers. The sensor path is shown as a black line.

6.5.3 Simulated environments with sensor noise

To demonstrate the robustness of the probabilistic occupancy grid mapping to sensor noise we performed experiments in simulation.

In [68] it is shown that the error distribution of stereo sensor noise can be approximated by a 3D Gaussian. In stereo vision, the measurement of range is only made indirectly by triangulating from matched correspondences in two images. One option is to add noise directly to the $[xyz]^T$ points produced. An obvious and easy way to do this would be to sample from a 3D Gaussian centred on each point. This method will produce noisy point clouds, and could be used to validate our approach, but a more realistic simulation can be achieved if we add Gaussian noise to the actual measurements – the 2D position in the left image which has been matched to a point in the right image.

Once noise has been added to this position, we can then perform the triangulation and get something closer to the skewed 3D Gaussian described in [68]. In the following experiments we have, for each pixel position in the simulated sensor, sampled from a 2D Gaussian centred on the true pixel position, with $\sigma = 1px$. This produces point clouds which simulate the noisy point clouds produced from a real stereo camera. More research could be done here to build a more accurate sensor noise model, but this would depend on a great number of factors such as camera resolution, CCD noise, and the intricacies of whichever stereo correspondence algorithm was used. We feel that the model described captures the important properties of the noise distribution, and provides an acceptable platform for demonstrating the occupancy grid approach's resilience to corruption by noise.

The images at the top of Fig. 6.19 shows side by side comparison of the same scene. (a) shows a sensor positioned in a simulated environment, (b) shows the data captured by the sensor with no noise, and (c) shows the effect of adding 2D Gaussian noise to the disparity image.

6.5.3.1 Experiments

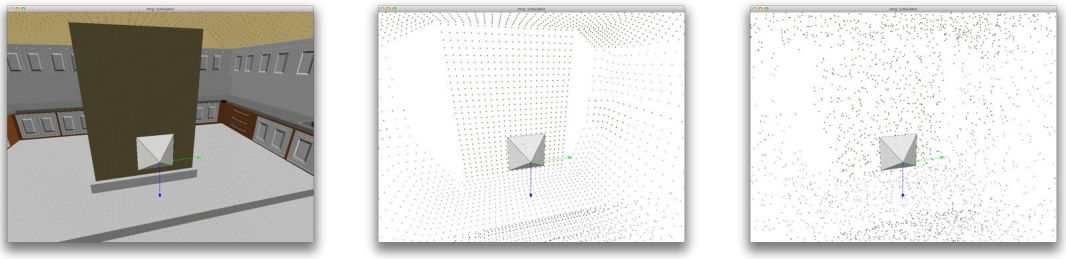
Our aim in this section is to demonstrate the robustness of the occupancy grid map approach to sensor noise, rather than to perform a thorough comparison of different exploration strategies under the influence of noise. We therefore have performed experiments using a subset of the exploration techniques introduced in Section 6.5: closest frontier, weighted information gain, and the potential method described in this chapter. We ran each of these in three of the simulated environments: the kitchen environment (Scenario 3), the apartment environment (Scenario 5), and the garage environment (Scenario 6). As before we performed 30 runs for each combination of algorithm and environment.

Due to the addition of noise we can expect that on termination of exploration the resultant map will not be identical to a map generated under the absence of noise. To measure the disparity between the two maps we compare the occupancy state of the voxels in the noisy map with a reference map generated by a sensor with no noise. The percentage of voxels which have the same occupancy state (occupied, free, or unknown) in both maps is recorded.

The tables in Fig. 6.19 show results from the experiments. In all cases complete exploration was achieved by all exploration algorithms – exploration terminated when there were no remaining frontier voxels. The mean path lengths are slightly longer to those recorded in the absence of noise, presumably due to misclassification of voxels. For example if a voxel is incorrectly deemed to be occupied, a route which avoids this voxel will necessarily be longer than a straight line path through it.

The results show that the occupancy grid map has proved to be robust to realistic amounts of noise in a simulated environment. With continued observation the accuracy values will approach 100% as the sensor noise is negated by the large number of measurements integrated into the probabilistic voxel grid. If this level of accuracy is required then multiple observations of each point in the world will

be required, and the robot behaviour would have to reflect this. Future work could include the use of occupancy probabilities to drive exploration – areas of the map which are classed as occupied, but with low confidence, could be marked as exploration targets alongside frontiers.



(a) (b) (c)

Scenario 3: Kitchen ($8 \times 8 \times 3$ m or 192000 voxels)

algorithm	mean path (m)	std dev	accuracy
closest	20.09	5.06	96.2%
weighted IG	24.22	7.28	96.6%
potential	19.69	5.14	97.0%

Scenario 5: Apartment ($6 \times 8 \times 2$ m or 96000 voxels)

algorithm	mean path (m)	std dev	accuracy
closest	34.8	8.2	95.9%
weighted IG	39.52	9.82	94.6%
potential	34.21	10.44	95.6%

Scenario 6: Garage ($7 \times 9 \times 2$ m or 126000 voxels)

algorithm	mean path (m)	std dev	accuracy
closest	26.65	8.85	96.7%
weighted IG	27.11	13.2	95.6%
potential	23.85	8.3	96.2%

Figure 6.19: Simulated results with noisy sensor. The three images show an environment, (a), being imaged by a sensor with no noise, (b), and imaged by a sensor with 2D Gaussian noise added to the disparity image, (c). The tables describe the results obtained when we performed simulated exploration in three environments with a noisy sensor. Path length statistics were obtained by performing 30 runs of each algorithm in each environment, with randomised starting positions. Accuracy is the percentage of voxels in the final map which match those in a reference map obtained by a sensor with no noise.

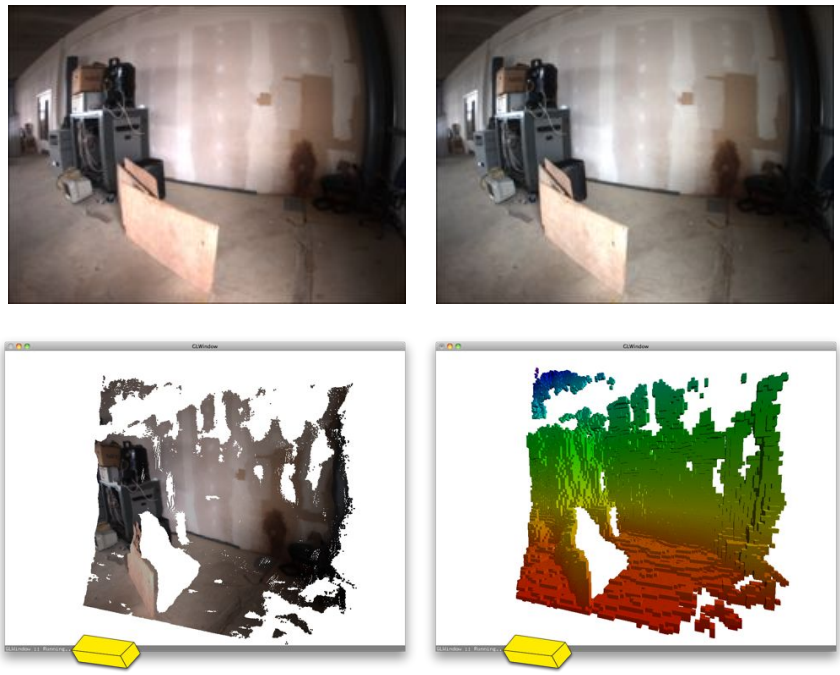


Figure 6.20: From stereo to occupancy grid. The upper images are the left and right images captured by the stereo camera. Lower left shows the resulting RGB point cloud, and lower right shows the result of integrating these points into the map.

6.5.4 Real environments

To demonstrate the ability of this algorithm to perform exploration on a real mobile platform we performed various trials in different environments. The Europa robot, previously described in Chapter 4, was used to explore a laboratory environment, and a warehouse environment. In both situations it was able to perform a complete exploration using visual odometry for pose estimation, dense stereo for map building, and the continuum method of this Chapter for goal selection and path planning. Figure 6.20 shows example data captured by the stereo camera, converted into a 3D point cloud, and integrated into an occupancy grid.

On the following pages we show results from exploration in various environments. As the robot is constrained to move on the ground plane we explore by taking a horizontal slice throughout the occupancy grid map and applying the continuum exploration algorithm to this layer.

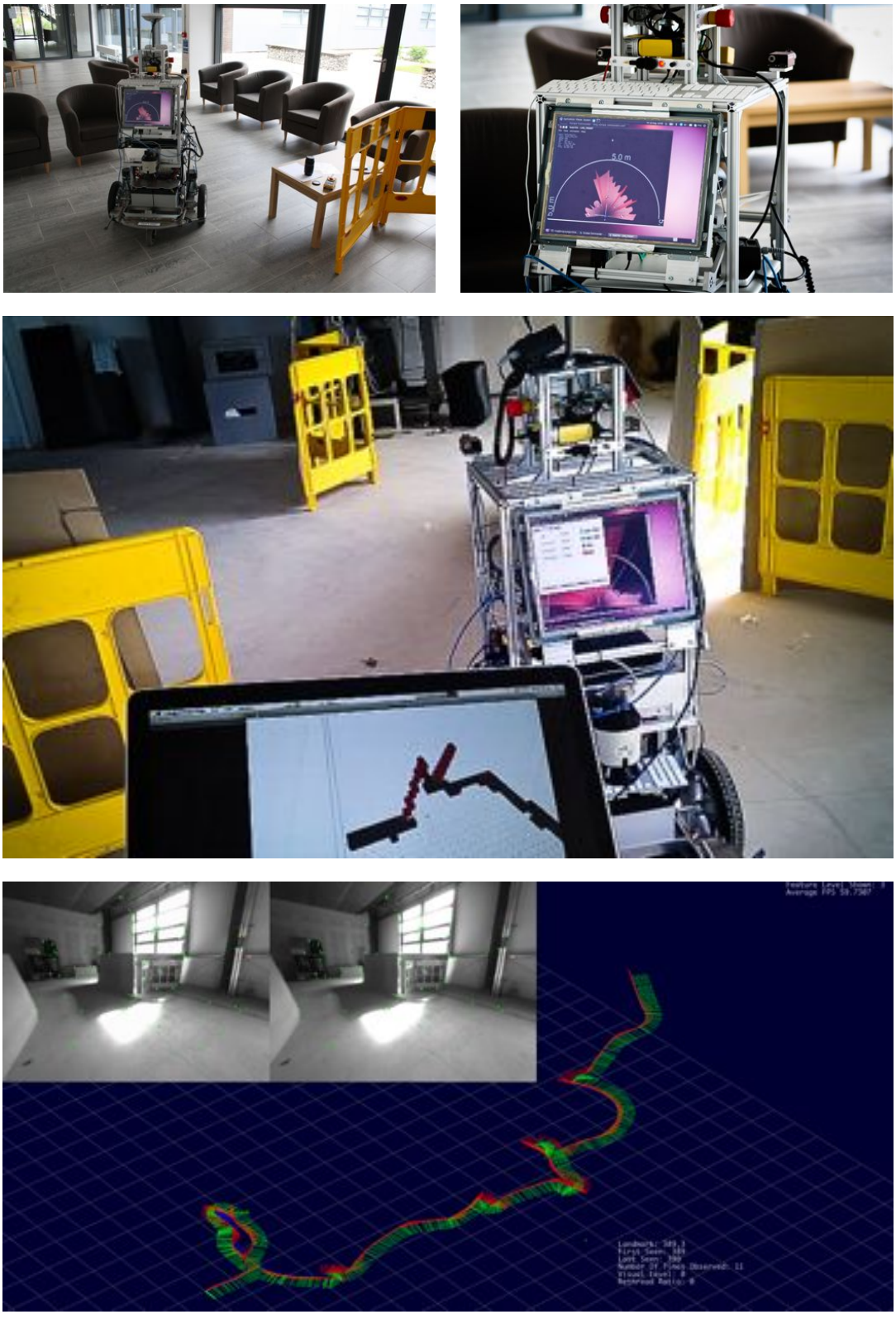


Figure 6.21: Europa exploration. The Europa robot performing autonomous exploration in various environments. The bottom image shows the visual odometry system in action.

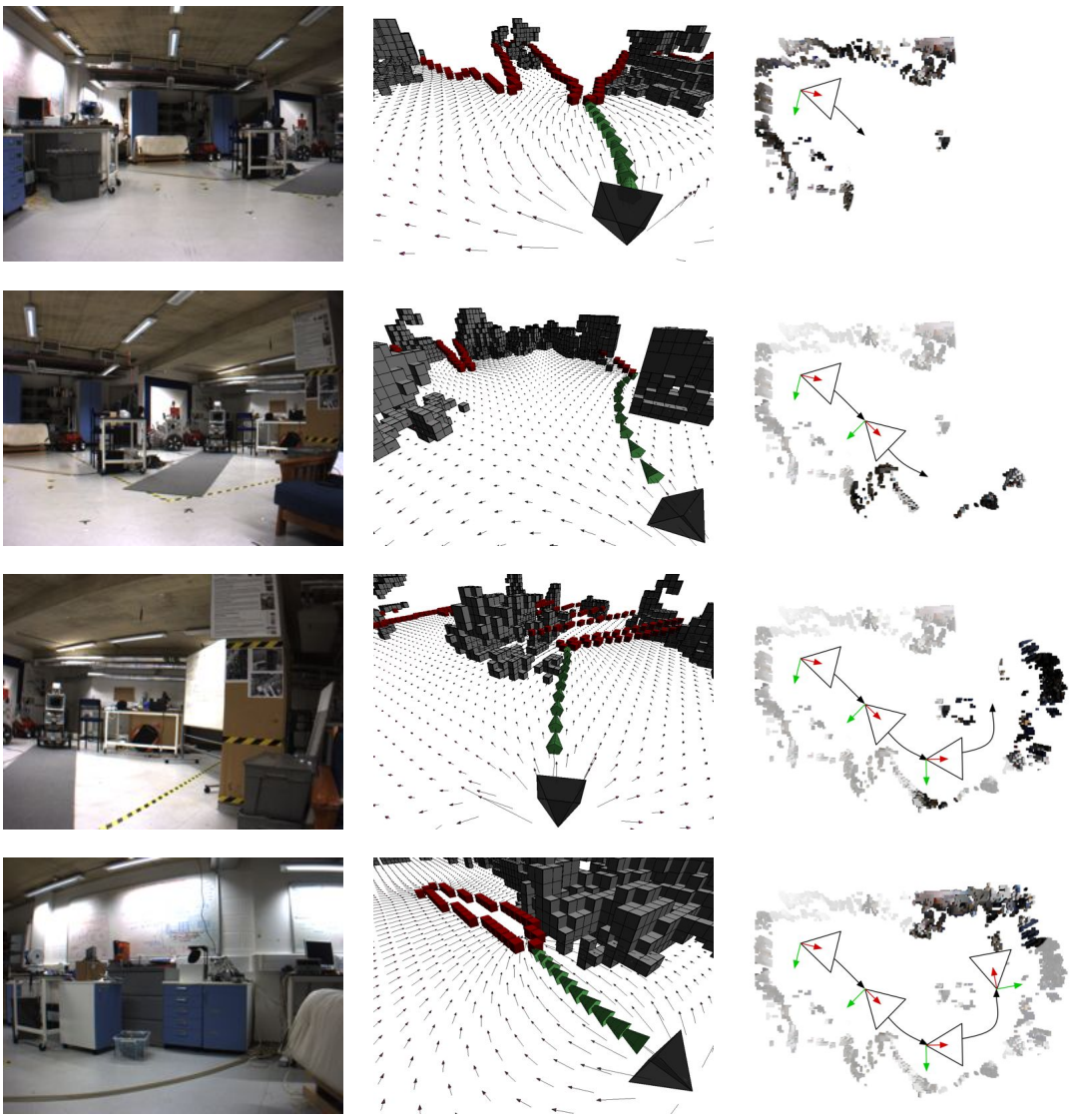


Figure 6.22: Exploration of lab environment. Representative images from a sequence of 66 scans taken while exploring a lab environment. The first column shows the left image from the stereo camera. The second column shows the gradient of the harmonic function at that pose, and the streamline chosen for exploration. The final column shows the updating map of the world – dark points are those seen from that particular pose.

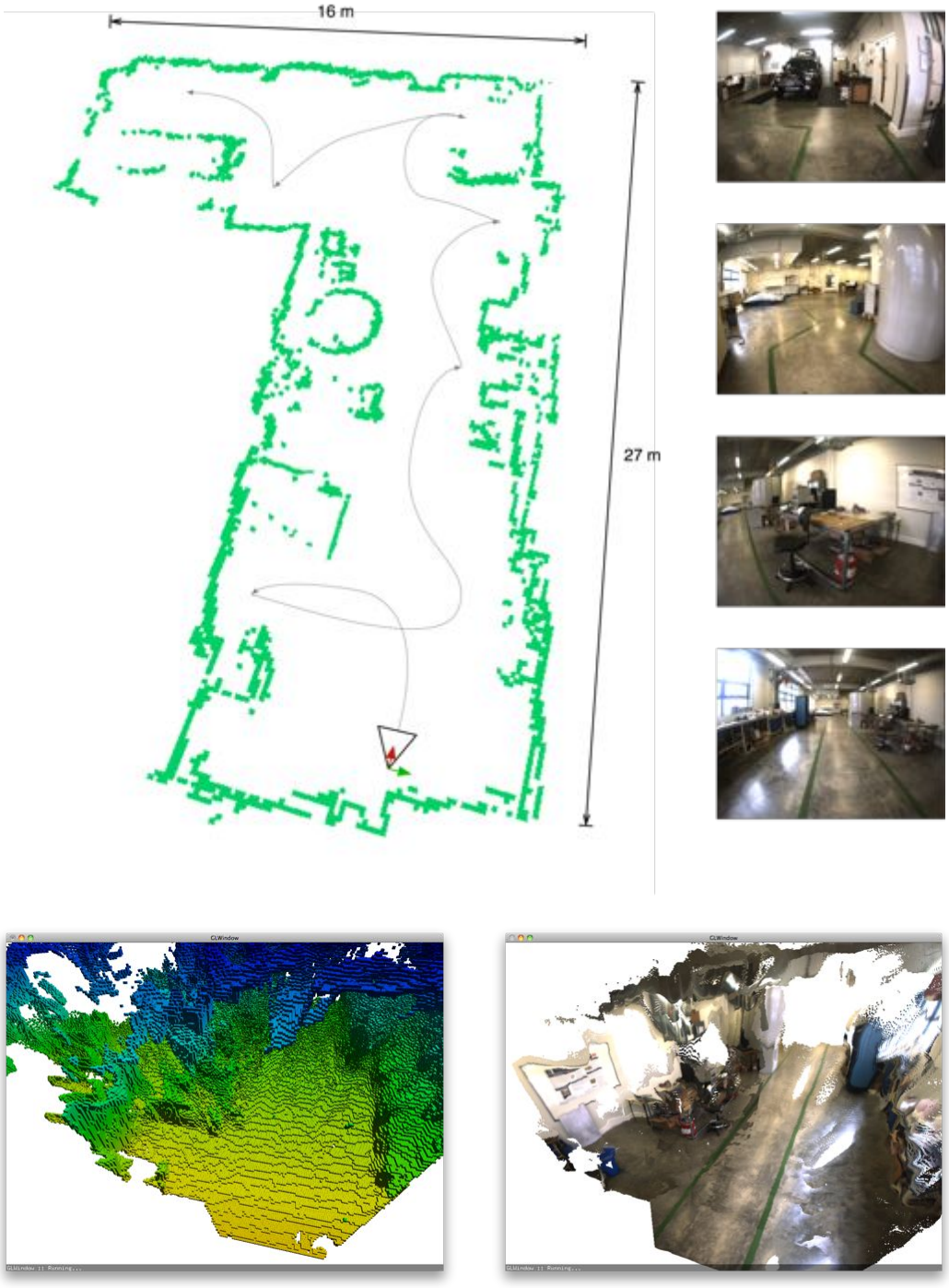


Figure 6.23: Exploration of industrial environment. The main image shows a horizontal slice through the occupancy grid map as used for exploration with the robot path superimposed in black. Representative images from the stereo camera are shown on the right. The bottom images show a section of the 3D occupancy grid map, and a section of the RGB point cloud as captured by the stereo camera.

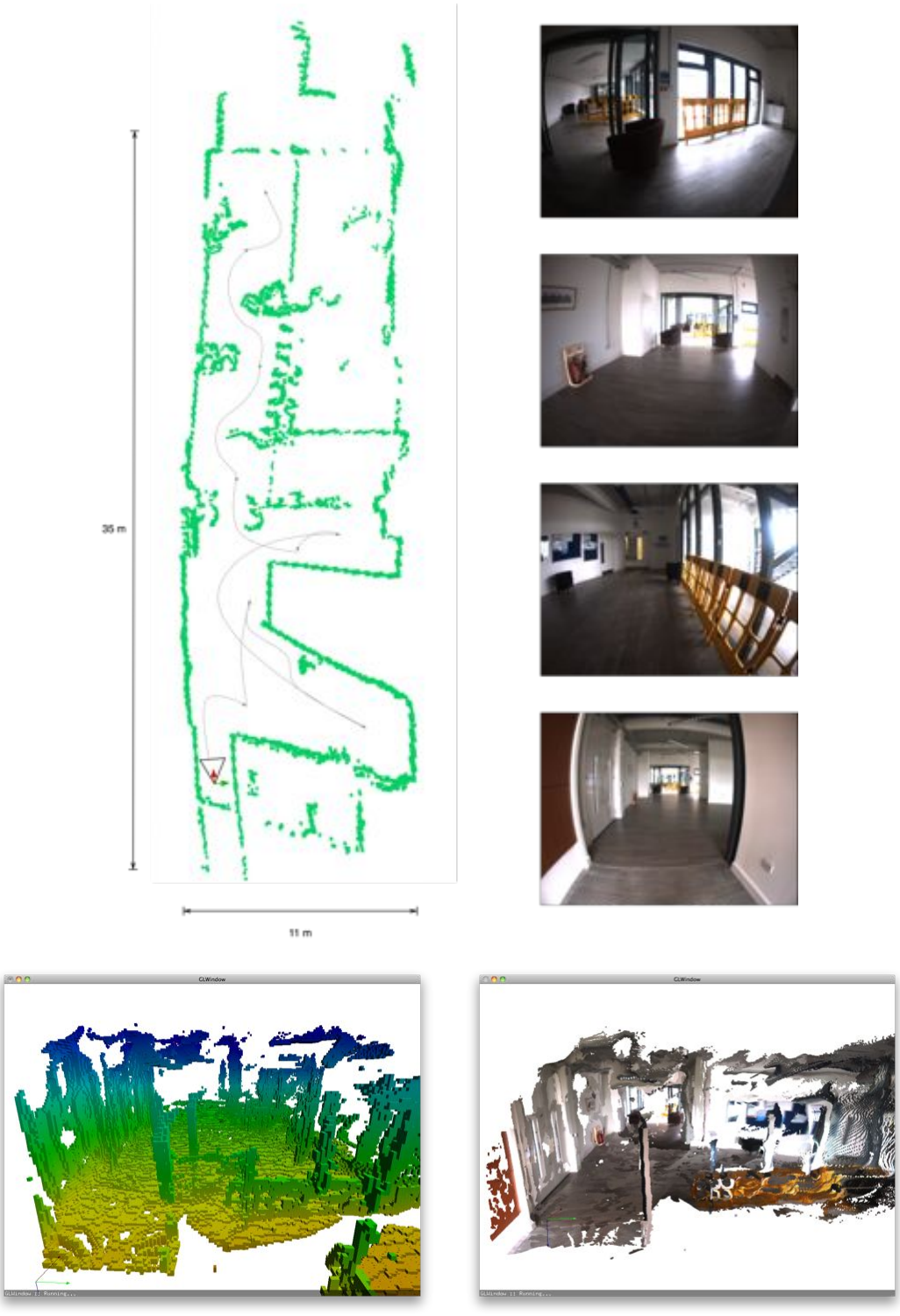
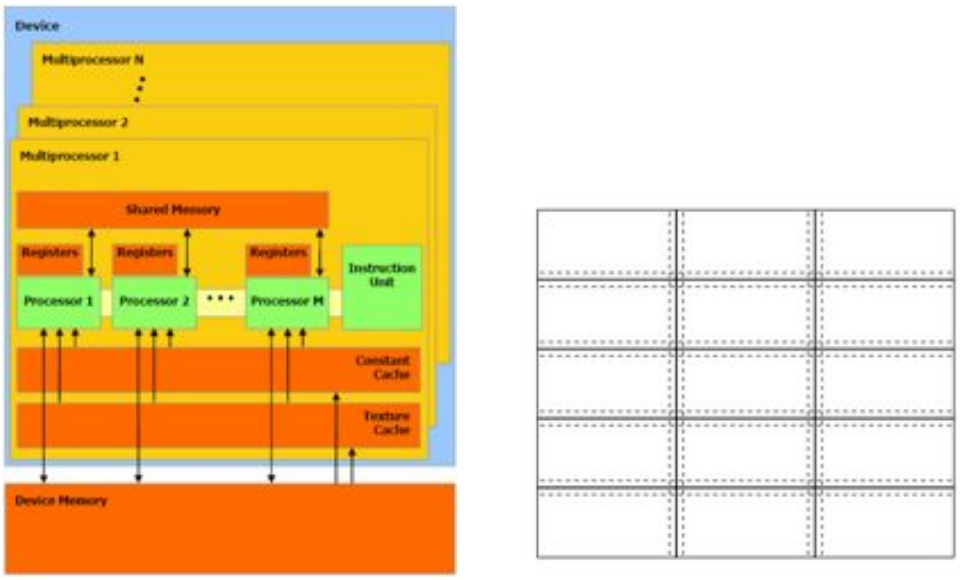


Figure 6.24: Exploration of office environment. The main image shows a horizontal slice through the occupancy grid map as used for exploration with the robot path superimposed in black. Representative images from the stereo camera are shown on the right. The bottom images show a section of the occupancy grid map, and a section of the RGB point cloud as captured by the stereo camera.



(a) (b)

Figure 6.25: CUDA architecture. The multiprocessor architecture on a modern GPU allows parallelisable algorithms to be run significantly faster than on a CPU. (a) shows the multiple processing units found in a typical GPU (image from [90]), and (b) shows the block structure used in the GPU implementation of the iterative solver of Laplace's equation (image from [36]).

6.6 Speed and scaling

We have demonstrated the exploratory abilities of our algorithm based on the solution to Laplace's equation but there are two further considerations:

Speed How do we calculate the PDE solution in real-time?

Scaling How can this system scale to larger environments?

The short answers are parallelisation of the problem on the GPU, which will be described in Section 6.6.1, and identification and elimination of fully explored areas from the search process, which will be described in Section 6.6.2.

6.6.1 Computing ϕ at high speed

Modern graphics cards such as those in the NVIDIA Geforce range [90] contain hundreds of processing cores, compared to the typical 1 to 4 cores found on a

standard desktop processor. The past few years have seen a rapid adoption of these GPU (graphics processing unit) devices to perform general purpose scientific and engineering computation as orders of magnitude speed-ups can be obtained for algorithms amenable to parallelisation across many cores.

6.6.1.1 CUDA

Originally the GPU architecture was designed for high speed processing of computer graphics – running the same transformation on millions of triangle primitives for example. Harnessing these capabilities for more general purpose computing was a laborious task, until NVIDIA introduced the Compute Unified Device Architecture (CUDA) in 2007. CUDA is the parallel computing architecture on NVIDIA GPUs and is exposed to programmers through standardised interfaces in common languages (C, python, etc.).

Figure 6.25 is a representation of the CUDA architecture on a typical GPU. The device consists of multiple *multiprocessors*, each of which contains a number of *processors*. Within a multiprocessor there is a block of shared memory which is accessible by all processors in that block. Code is passed to the GPU as a *kernel* – a block of code which will be executed in parallel on all the processing units. Each kernel can programatically access an ID which can be used to specify which parts of the shared memory a specific processor should operate on.

The execution of code on a CUDA GPU is done in 4 stages:

- Data is copied to the GPU shared memory
- A code kernel is transferred to the GPU
- GPU executes the kernel on all processors in parallel
- Results are copied back to the CPU

Figure 6.26: CUDA code comparison. Two snippets of code demonstrating the added complexity which CUDA requires. On the left is the CPU implementation of the central iterative FDM solver, on the right is the CUDA implementation. The added complexity is due to the three stage pipeline: copy data to the GPU, execute a kernel on multiple GPU processors, copy results back.

6.6.1.2 Solving Laplace's equation on the GPU

The grid based structure of the PDE domain and the solution through Jacobi iteration lend themselves well to parallelisation. However there are a number of considerations and potential pitfalls which must be taken into account to achieve the maximum speed-up. Some common pitfalls are listed in [36]:

Coalesced memory transfers: The highest bandwidth between the global GPU memory and the multiprocessors is only achievable if transfers are of contiguous blocks of memory, with each thread accessing memory at a 2-byte (or multiple of) offset

Keeping the hardware busy: Each processor should be occupied and working for as long as possible – it is inefficient if some must lie dormant waiting for others to complete their kernel execution

Data division into blocks: Figure 6.25(b) shows conceptually the division of mem-

ory into blocks (dotted lines) each of which will be processed independently by a multiprocessor on the GPU. There is some overlap between blocks as the results at the edge of one block will influence the results in the next. The block size chosen is dependant on the specifications of the GPU being used.

Our implementation is based on the freely available code provided by [36], modified somewhat to fit into our architecture. We use a block size of 32×4 and process three horizontal slices of the occupancy grid at once. This gives us a speed-up of $9\times$ on a 2010 Macbook Air with a Geforce 320M graphics card. The CUDA implementation allows us to solve Laplace's equation over a grid of 100000 voxels in less than a second.

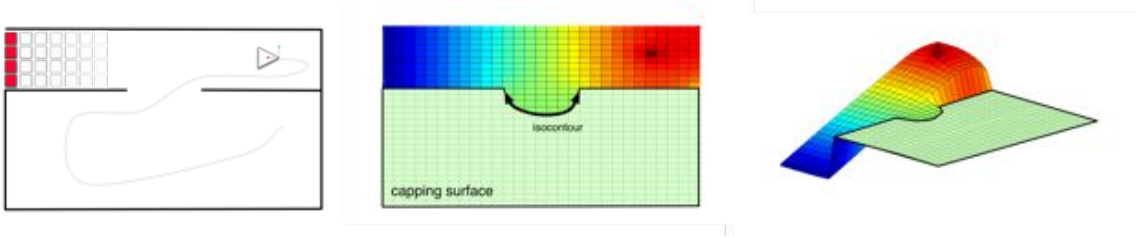


Figure 6.27: Capping surfaces. A robot has fully explored the large room at the bottom of (a) and is now positioned at the right hand end of the corridor. Unexplored space lies beyond the lefthand end of the corridor, shown here as red frontier voxels. Using the boundary conditions described in Section 6.2.5, and solving Laplace's equation over the freespace results in the scalar field shown in (b) and (c). Our aim is to algorithmically identify the large explored region, highlighted as a capping surface in (b), so that it can be excluded from further calculations.

6.6.2 Computing ϕ over larger scales

Implementing the PDE solver on the GPU has the result of computation time by a constant factor – it does not affect the computational complexity which continues to scale as $O(n^3)$. This is because the PDE domain is defined to be the entire freespace region of the map and as exploration progresses this area necessarily expands in three dimensions.

If we could identify regions in the map which have been fully explored and are therefore of no further interest, then we could exclude these areas from the calculation and potentially cap the growth of the PDE domain.

Specifically we want to find "stagnant" regions – volumes in which the gradient $\nabla\phi$ is uniform and zero. If such an volume exists in a solution to Laplace's equation then we can simplify future calculations by removing the interior points from the PDE domain and replacing them with a Dirichlet boundary on the surface of the volume (set to the fixed value found at the interior points). This section will discuss how we can identify these stagnant volumes algorithmically.

6.6.2.1 An Example

Consider a long corridor with a single fluid source at one end, the other end consisting of sinks. This situation arising from robot exploration is shown in Fig. 6.27: a robot has fully explored the large room at the bottom of Fig. 6.27(a) and is now positioned at the right hand end of the corridor. Unexplored space lies beyond the lefthand end of the corridor, shown here as red frontier voxels.

The resulting potential flow field shown in Fig. 6.27(b) and (c) is a steady downwards gradient from right to left. Our aim is to algorithmically identify the large explored region, highlighted as a capping surface in (b), so that it can be excluded from further calculations.

6.6.2.2 Gauss' Theorem

The divergence theorem, or Gauss' theorem, states that the outward flux of a vector field across a closed surface is equal to the integral of the divergence of the vector field in the volume contained by the surface

$$\int_V (\nabla.(\nabla\phi))\, dV = \int_S ((\nabla\phi).\mathbf{n})\, dS \tag{6.42}$$

where $\mathbf{n}$ is the outward pointing normal of the surface S. The volume contained within this closed surface is V.

$\nabla\dot{\phi}$ is the divergence of the vector field – the magnitude of outwards flux on an infinitesimal volume surrounding any point in the field. In other words it is a measure of the existence of sources or sinks: a positive divergence indicates a source, a negative divergence indicates a sink.

A volume which contains no sources and no sinks will have a divergence of zero,

and hence the surface integral will be zero

$$\int_S (\nabla\phi).\mathbf{n}\, dS = 0 \tag{6.43}$$

6.6.2.3 Isovolumes

The result of Eq. (6.43) is a necessary but not sufficient condition for identifying a stagnant volume. There are many possible volumes which satisfy this condition but which have internal gradients. We require not just zero divergence, but zero outwards flux at any point on the surface: we need to find an isovolume.

An isovolume is a volume $V \in \delta S$ within which $\nabla\phi = 0$ at every point

$$\nabla\phi_{x,y,z} = 0 : \forall(x, y, z) \in V \tag{6.44}$$

If we can find such a volume then we satisfy both Eq. (6.42) and Eq. (6.44) as zero internal gradient will result in zero divergence across the volume.

6.6.2.4 Implementation

The iterative solver described in Section 6.4.1 produces a close approximation to the true solution to Laplace's equation. If we could obtain an exact solution then we could easily identify isovolumes by starting at a given voxel and recursively adding neighbouring voxels with identical values of ϕ to the isovolume. There are no such exact matches in the iterative solution. It does however lend itself to finding isovolumes through a similar method of neighbour expansion, but neighbour expansion combined with gradient descent.

All voxels are initialised with $\phi = 0$. These values of course change as the iterative solver progresses, with the result that in an enclosed explored area such as the room in Fig. 6.27 the voxels furthest from the entrance have lower values of ϕ than those

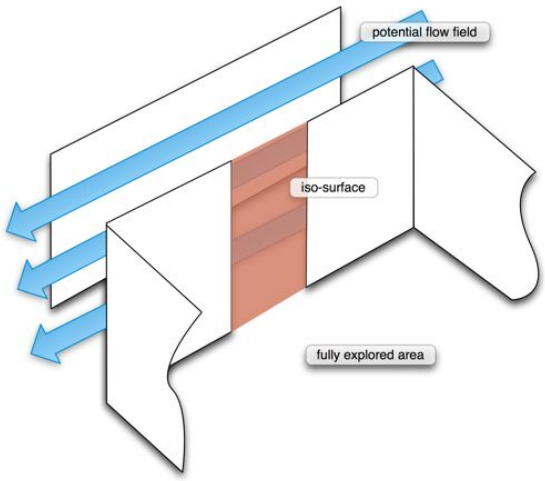


Figure 6.28: Isovolume. The fully explored area is capped with an isovolume which acts as a virtual obstacle boundary for the potential flow field. The result is that the iterative solution to Laplace's equation requires fewer calculations as we can disregard the voxels found in the explored area.

close to the entrance. An iso-surface of ϕ can be found lying across the entrance, as shown in Fig. 6.28.

The required stagnant volume will have a bounding surface consisting of a combination of the domain boundary δS and an iso-surface in ϕ. The fraction of the surface which is coincident with δS will only consist of occupied voxels (not sources or sinks).

Our algorithm progresses is presented as pseudo-code in Algorithm 4.

6.6.2.5 Results

Figure 6.29 shows typical results of applying Algorithm 4 in a 2D maze environment. In this instance we have identified two isovolumes which are marked in green as fully explored regions. The robot is now free to ignore these areas when calculating future solutions to Laplace's equation as it continues to explore the environment. The proposed trajectory is shown in black from the robot position to a frontier.

The algorithm scales to 3D with no changes, as shown in Fig. 6.30. Figure 6.30(a) shows an overview of the environment in the `Simulator` application. The sensor

Algorithm 4: Isovolume identification

Input: Seed vertex s
Output: Isovolume $\mathcal{I}$

```
I ← ∅ % set of vertices in isovolume
N ← [s] % set of vertices to explore
while N ≠ ∅ do
    n ← pop(N)
    if n == Sink then
        return ∅
    else
        for v ∈ n.Neighbours do
            if v ≠ Occupied and v ∉ N and v ∉ I and φ_v ≤ φ_s then
                % neighbour does not have higher φ than seed vertex
                % and so it is added to the isovolume
                I ← [I, v]
            end
        end
    end
end
return I
```

trajectory is shown as a black line. Figure 6.30(b) shows detail of the current map in the `Commander` application. The small room on the left has been fully explored as it contains no red frontier voxels. Finally Fig. 6.30(c) shows the result of running Algorithm 4 – the small room is correctly identified as fully explored, and the entrance can be "sealed off" from the exploration algorithm by adding a virtual obstacle boundary.

We have demonstrated an algorithm for identification of fully explored regions of the map which is reliant on the field ϕ. As the map expands we can seal off volumes which no longer influence exploratory behaviour, reducing the computational cost of calculating ϕ significantly – potentially reducing the cost from $O(n^3)$ to linear $O(m)$, where n is the number of voxels in the map and m is the number of voxels in non-stagnant regions.

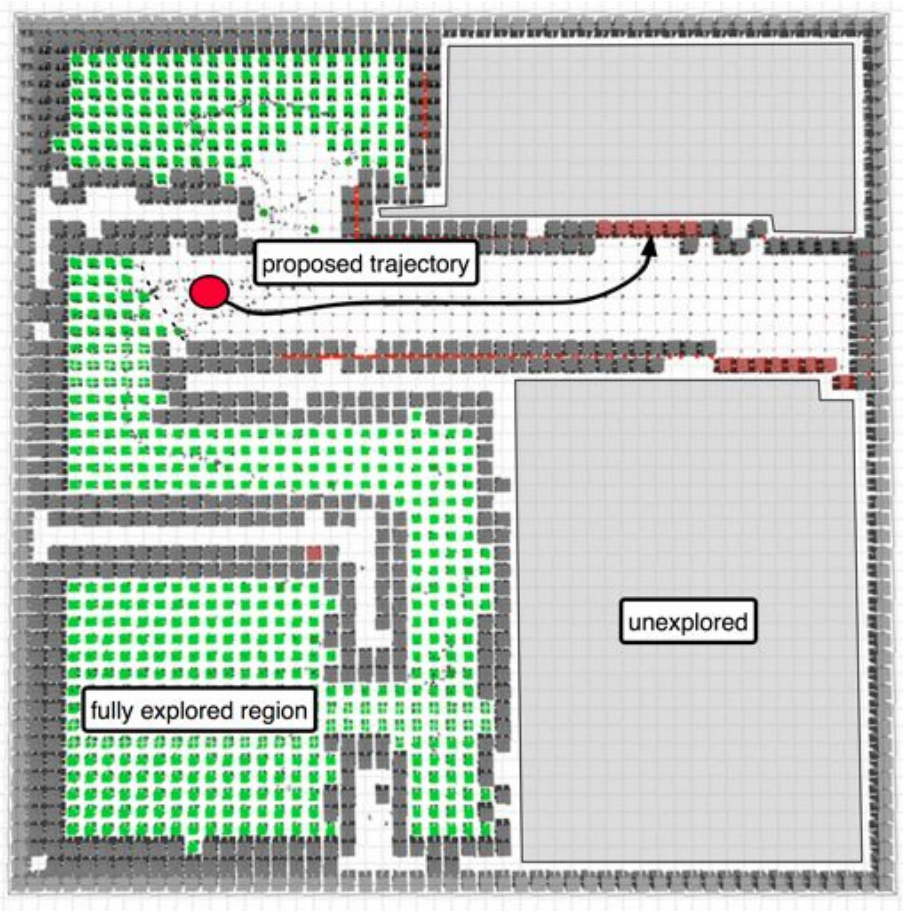
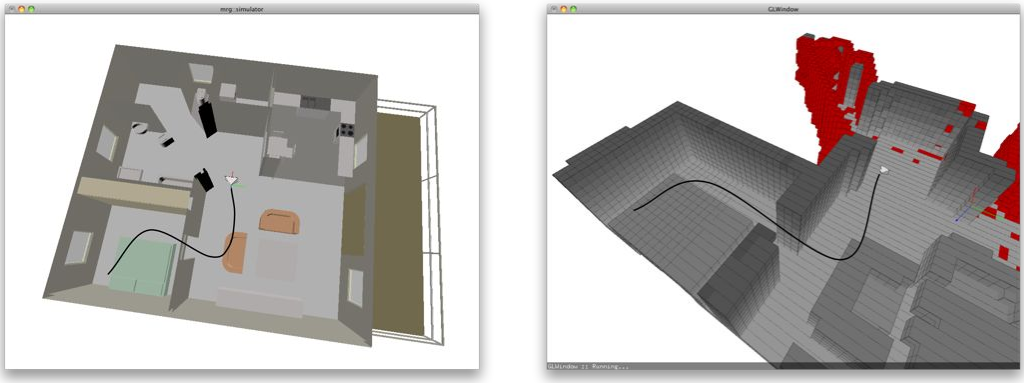
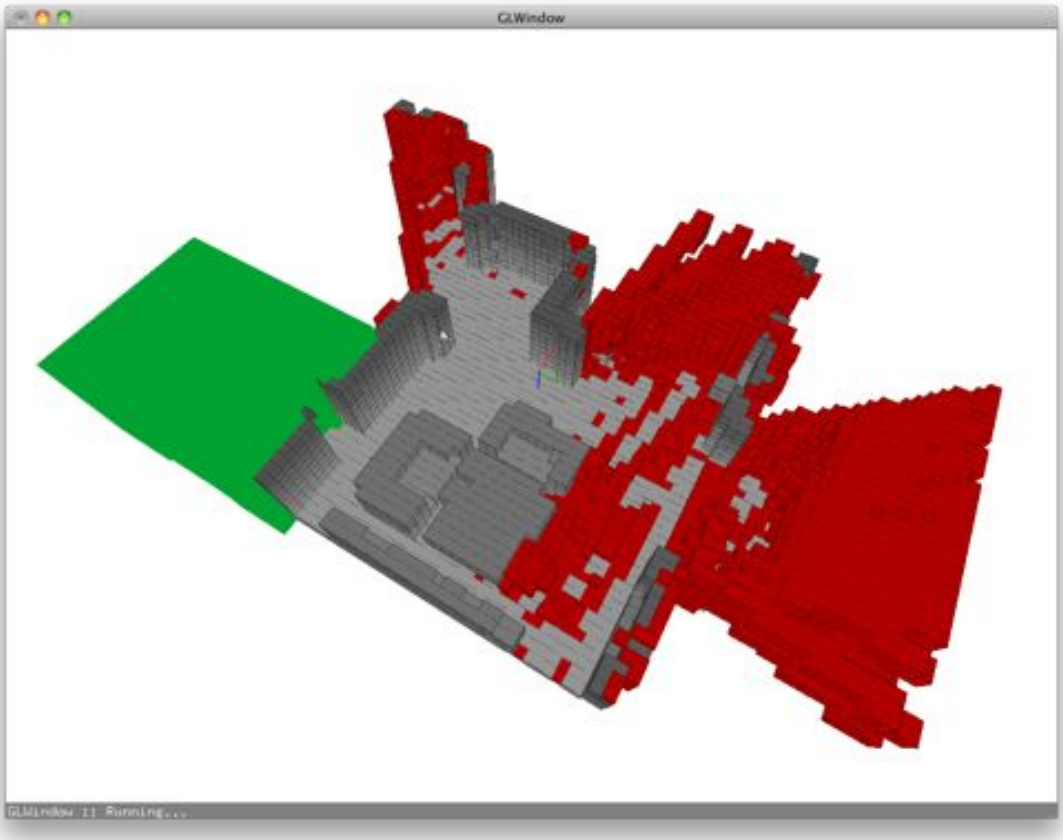


Figure 6.29: Capping surface in 2d. An example of identifying capping surfaces of fully explored regions on real data. This is a 2D slice through an occupancy grid map. Black cubes are obstacles, red cubes are frontiers, and green cubes indicate areas which have been identified as fully explored. These areas can now be ignored in future calculations of ϕ.



(a) (b)



(c)

Figure 6.30: Capping volume in 3D. The capping volume algorithm applied to a 3D map. (a) shows an overview of a simulated environment – the sensor trajectory is shown as a black line. (b) shows detail of the current map. The small room on the left has been fully explored as it contains no red frontier voxels.(c) shows the result of running Algorithm 4 – the small room is correctly identified as fully explored, shown in green.

Chapter 7

Conclusions

This Chapter presents a summary of the material covered in the preceding Chapters, reiterating the main contributions, and discusses some avenues of future work.

7.1 Summary of contributions

We began with the observation that the problem of exploration is about choosing where to go. Chapter 2 presented a summary of related work which provided a foundation for the chapters to come. It covered the problem of pose estimation, and justified the choice of visual odometry for answering the question of where the robot is located. We then looked at various data structures for maintaining consistent maps of explored environments, and settled on the choice of the probabilistic occupancy grid map. A summary of existing exploration methods was then presented, helping to frame the work done in later chapters.

Chapter 3 explained the rationale behind choosing a high resolution stereo camera as the depth sensor of choice in this work, and proceeded to explain the algorithms which were implemented to convert camera images into 3D point clouds. The design and construction of a 3D environment simulator which can be used for repeatable

experiments was discussed.

Chapter 4 presented the robots used in the Mobile Robotics Group, and the software systems which run on them. In particular the physical construction and characteristics of the Europa robot were outlined, and the software engineering which resulted in the high level Commander application was explained.

With the competencies of pose estimation, acquisition of 3D data, and mapping, and a robot system which can be controlled through software systems, we reach Chapter 5. In this Chapter we describe a graph-based exploration algorithm, which has parallels to the Gap Navigation Tree of [126]. Assuming that the world is topologically connected, we attempt to create a graph which is embedded on the surface of the environment. Using depth data from a stereo camera we construct a graph in which edge weights correspond to Euclidean distance between 3D points, and identify areas of the graph with high average edge weight. Such an area is called a rift, and is a target for exploration. We guide the camera across the graph to view the most promising rift and show that by careful consideration of the properties of the graph we can merge depth data from the new pose, and eliminate spurious edges through a visibility check.

The reliance of the graph-based exploration algorithm on arbitrary thresholds, and the unrealistically accurate pose estimates and sensor data required, motivated the search for a parameter-free algorithm which could cope with noisy data. Chapter 6 explains our solution: solving Laplace's equation over the free-space of an occupancy grid map, and following streamlines in the resulting potential flow field to guide the robot to view a previously unexplored region. We discussed the mapping from occupancy grid to a discretised version of Laplace's equation, how to set the boundary conditions appropriately such that exploration is guaranteed to be complete, and finding the solution using an iterative finite difference method.

Ultimately we show that our method produces paths with lengths comparable to

closest frontier methods, and outperforms the oft-used information gain methods. It combines both exploration target identification (where should I go next?) and path planning (how do I get there?) into one framework which can run in realtime when implemented on standard GPU hardware, and is parameter-free. Additionally the properties of the vector field solutions lend themselves to identification of fully explored areas, allowing us to significantly reduce the computation needed for subsequent exploration, and scale to larger environments.

7.2 Future Work

There is plenty of scope for future improvements and extensions to the work presented in this thesis. Some interesting possibilities include:

1. **Incorporating uncertainty from occupancy grid** The probabilistic occupancy grid maps maintain, for each voxel, a probability that the voxel is occupied by an obstacle. When we perform the mapping from occupancy grid to Laplace's equation we base the boundary conditions on the maximum likelihood estimate of the map. An area of the map containing voxels with an 80% occupancy estimate may well be worth further investigation – we could weight the potential values in such cells accordingly and use this to guide the algorithm to re-inspect areas in which we have lower confidence.

2. **Identification of poorly textured regions** Given that we know the sensor pose, and we have a partially explored map, we should be able to make an educated guess as to the sort of depth data we should be generating. For example, a camera pointed directly across a frontier boundary would be expected to resolve voxels lying beyond the boundary (or on it if it turns out to lie along a wall). If we don't see such data, then could we use this to infer that whatever the camera is looking at may be poorly textured or badly lit, and

thus unable to be sensed by the stereo camera? If we can identify connected regions which exhibit this behaviour then we can certainly avoid the futile repeated sensing of the same area.

3. **Applying kinematic constraints** We currently assume that the paths generated by the streamline method are possible for the robot to follow. This was always true for the experiments we carried out, the curving trajectories resulting in smooth robot motion, but for robots with more complex configuration spaces this assumption may no longer be valid.

   Producing a kinematic model of a snake-arm robot would allow us to perform path planning in the high dimensional configuration space using probabilistic roadmap techniques, but by doing so we would be disregarding the paths produced by the continuum method.

   The potential method could still be used to generate exploration targets, but the actual path selection would have to be done seperately. Alternatively with appropriate computational power we could apply the PDE method to known free volumes in the configuration space, but this would scale exponentially with the dimensionality of the configuration space.

4. **Applying size constraints** A real robot has a physical presence and this should be taken into account when performing path planning (to avoid collisions). A simple improvement to the occupancy grid algorithms presented would be to expand obstacle voxels by $r/2$ in all directions, where r is the radius of a sphere which fully envelopes the robot. Paths planned through the reduced free-space of this modified map would then be guaranteed to be collision free if followed exactly.

5. **Providing an exploration evaluation framework** The Middlebury evaluation framework [72] has been of great value in the computer vision community

by providing an independently maintained 'league table' of the performance of dense stereo algorithms. There is currently nothing comparable in the field of robot exploration. A simple to use 3D simulator with a well defined interface which allows an end-user to easily and iteratively test exploration strategies would be beneficial.

7.3 Concluding Remark

This thesis has presented two novel exploration algorithms for mobile robots, and described the software and hardware infrastructure required to support their operation. We hope that our work, and specifically the continuum method based on Laplace's equation, will be useful in the development of fully autonomous exploration in three dimensions as robots leave the ground plane behind.

Bibliography

[1] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, page 10. Citeseer, 1987.

[2] W. F. Ames. *Numerical Methods for Partial Differential Equations*. Academic Press Inc, 1992.

[3] F. Amigoni. Experimental evaluation of some exploration strategies for mobile robots. In *IEEE International Conference on Robotics and Automation*, pages 2818–2823. IEEE, 2008.

[4] A. Ansar, A. Castano, and L. Matthies. Enhanced Real-time Stereo Using Bilateral Filtering. *Second International Symposium on 3D Data Processing*, pages 455–462, 2004.

[5] S. Axler, P. Bourdon, and W. Ramey. *Harmonic Function Theory (Graduate Texts in Mathematics, Vol 137)*. Springer, 2002.

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

[7] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-up robust features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.

[8] R. E. Bellman. *Dynamic Programming*. Number 1 in RAND CORPORATION. Research studies. Princeton University Press, 1957.

[9] P. Besl and H. McKay. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[10] Bluebotics. *http://www.bluebotics.com/*, (accessed May, 2011).

[11] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-eppstein, C. Pantofaru, M. Wise, M. Lorenz, W. Meeussen, and S. Holzer. Towards Autonomous Robotic Butlers: Lessons Learned with the PR2. In *IEEE International Conference on Robotics and Automation*, 2011.

[12] A. C. Bovik. *Handbook of Image and Video Processing*. Academic Press, 2005.

[13] Y. Boykov, O. Veksler, and R. Zabih. Minimization via Graph Cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(11):1222–1239, 2001.

[14] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[15] M. Z. Brown, D. Burschka, G. D. Hager, and S. Member. Advances in Computational Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(8):993–1008, 2003.

[16] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. BRIEF : Binary Robust Independent Elementary Features. *European Conference on Computer Vision (ECCV)*, pages 778–792, 2010.

[17] D. Cole. Using laser range data for 3D SLAM in outdoor environments. *Robotics and Automation, 2006.*, pages 1556–1563, 2006.

[18] C. Connolly, J. Burns, and R. Weiss. Path planning using Laplace's equation. *Proceedings., IEEE International Conference on Robotics and Automation*, pages 2102–2106, 1990.

[19] C. Connolly and R. Grupen. The applications of harmonic functions to robotics. *IEEE International Symposium on Intelligent Control*, pages 498–502, 1992.

[20] P. Corke, D. Strelow, and S. Singh. Omnidirectional visual odometry for a planetary rover. *IEEE International Conference on Intelligent Robots and Systems (IROS)*, 4:4007–4012, 2004.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 2009.

[22] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec. 1959.

[23] H. Du, P. Henry, X. Ren, M. Cheng, D. Goldman, S. Seitz, and D. Fox. Interactive 3D Modeling of Indoor Environments with a Consumer Depth Camera. In *International Conference on Ubiquitous Computing*, 2011.

[24] Y. Du, D. Hsu, H. Kurniawati, W. S. Lee, S. C. W. Ong, and S. W. Png. A POMDP Approach to Robot Motion Planning under Uncertainty. *Int. Conf. on Automated Planning and Scheduling*, 2010.

[25] G. Dudek and M. Jenkin. *Computational Principles of Mobile Robotics*. Cambridge University Press, 2000.

[26] G. Dudek, M. Jenkin, and E. Milios. Robotic exploration as graph construction. *Robotics and Automation,*, 1991.

[27] H. Durrant-Whyte and T. Bailey. Simultaneous Localisation and Mapping: Part I The Essential Algorithms. *Robotics and Automation Magazine, IEEE*, 13(2):99–110, 2006.

[28] A. Elfes. Sonar-based real-world mapping and navigation. *Robotics and Automation, IEEE Journal of*, 1987.

[29] A. Elfes. *Occupancy Grids: A Probabilistic Framework for Robot Perception and Navigation*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1989.

[30] N. Fairfield, G. Kantor, and D. Wettergreen. Real-Time SLAM with Octree Evidence Grids for Exploration in Underwater Tunnels. *Journal of Field Robotics*, 24(1-2):03–21, Jan. 2007.

[31] O. D. Faugeras. *Fundamentals In Computer Vision*. Cambridge University Press, 1983.

[32] J. Fournier, B. Ricard, and D. Laurendeau. Mapping and Exploration of Complex Environments Using Persistent 3D Model. *Fourth Canadian Conference on Computer and Robot Vision (CRV '07)*, pages 403–410, May 2007.

[33] L. Freda, G. Oriolo, and F. Vecchioli. Sensor-based exploration for general robotic systems. In *IEEE International Conference on Intelligent Robots and Systems*, pages 2157–2164. IEEE, Sept. 2008.

[34] P. Fua, I. Sophia-antipolis, R. Lucioles, and F.-V. Cedex. A parallel stereo algorithm that produces dense depth maps and preserves image features. *Machine Vision and Applications*, 6:35–49, 1993.

[35] S. Garrido, L. Moreno, D. Blanco, and F. Martín Monar. Robotic Motion Using Harmonic Functions and Finite Elements. *Journal of Intelligent and Robotic Systems*, 59(1):57–73, Oct. 2009.

[36] M. Giles. Jacobi iteration for a Laplace discretisation on a 3D structured grid. Technical report, Oxford University, 2008.

[37] GLM library. *http://devernay.free.fr/hacks/glm/*, (accessed July, 2011).

[38] H. Gonzalez-Banos and J. Latombe. Navigation strategies for exploring indoor environments. *The International Journal of Robotics Research*, 21(10-11):829, 2002.

[39] J. L. Gross and T. W. Tucker. *Topological Graph Theory*. John Wiley and Sons, 1987.

[40] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[41] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2004.

[42] P. Henry, M. Krainin, and E. Herbst. RGB-D Mapping: Using depth cameras for dense 3D modeling of indoor environments. In *RSS Workshop on Advanced Reasoning with Depth Cameras*, 2010.

[43] L. Hibbert. Snakes Alive. *Professional Engineering*, pages 35–36, Mar. 2007.

[44] H. Hirschmuller, P. Innocent, and J. Garibaldi. Real-time correlation-based stereo vision with reduced border errors. *International Journal of Computer Vision*, 47(1):229–246, 2002.

[45] D. Holz, N. Basilico, F. Amigoni, and S. Behnke. Evaluating the Efficiency of Frontier-based Exploration Strategies. *ISR/ROBOTIK 2010*, 1(June):374–386, 2010.

[46] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface Reconstruction from Unorganized Points. *ACM SIGGRAPH Computer Graphics*, 26(2):71–78, July 1992.

[47] Q. Hu, X. He, and J. Zhou. Multi-Scale Edge Detection with Bilateral Filtering in Spiral Architecture. In *Pan-Sydney area workshop on Visual information processing*, volume 36, pages 29–32, 2004.

[48] iRobot. *http://www.iRobot.com*, (accessed June, 2011).

[49] K. I. Joy. Numerical Methods for Particle Tracing in Vector Fields, 2009.

[50] M. Karnik. A comparative study of Dirichlet and Neumann conditions for path planning through harmonic functions. *Future Generation Computer Systems*, 20(3):441–452, Apr. 2004.

[51] O. Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986.

[52] H. Kitano and S. Tadokoro. A Grand Challenge for Multiagent and Intelligent. *AI Magazine*, 22(1):39–52, 2001.

[53] K. Klasing, D. Althoff, D. Wollherr, and M. Buss. Comparison of surface normal estimation methods for range sensing applications. *2009 IEEE International Conference on Robotics and Automation*, pages 3206–3211, May 2009.

[54] A. Klaus, M. Sormann, and K. Karner. Segment-Based Stereo Matching Using Belief Propagation and a Self-Adapting Dissimilarity Measure. In *Pattern Recognition, International Conference on*, pages 15 – 18, 2006.

[55] V. Kolmogorov. Computing Visual Correspondence with Occlusions via Graph Cuts. *International Conference on Computer Vision*, pages 508—-515, 2001.

[56] K. Konolige. Projected texture stereo. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 148–155. IEEE, 2010.

[57] B. Kuipers and Y.-t. Byun. A Robot Exploration and Mapping Strategy Based on a Semantic Hierarchy of Spatial Representations . *Robotics and Autonomous Systems*, pages 1–23, 1993.

[58] C. Kunz, C. Murphy, H. Singh, C. Pontbriand, R. a. Sohn, S. Singh, T. Sato, C. Roman, K.-i. Nakamura, M. Jakuba, R. Eustice, R. Camilli, and J. Bailey. Toward extraplanetary under-ice exploration: Robotic steps in the Arctic. *Journal of Field Robotics*, 26(4):411–429, Apr. 2009.

[59] Y. Landa, D. Galkowski, Y. R. Huang, A. Joshi, C. Lee, K. K. Leung, G. Malla, J. Treanor, V. Voroninski, A. L. Bertozzi, and Y.-H. R. Tsai. Robotic Path Planning and Visibility with Limited Sensor Data. *2007 American Control Conference*, pages 5425–5430, July 2007.

[60] S. K. Lando and A. K. Zvonkin. *Graphs on surfaces and their applications*, volume 141 of *Encyclopaedia of mathematical sciences ; 141, Low-dimensional topology ; 2*. Springer-Verlag, 2004.

[61] J.-C. Latombe. *Robot Motion Planning*, volume 124 of *The Springer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1991.

[62] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[63] K. Low and A. Lastra. An Adaptive Hierarchical Next-Best-View Algorithm for 3D Reconstruction of Indoor Scenes. In *Proceedings of 14th Pacific Conference on Computer Graphics and Applications (Pacific Graphics 2006)*, pages 1–8, 2006.

[64] D. G. Lowe. Object recognition from local scale-invariant features. *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 2([8):1150–1157 vol.2, 1999.

[65] F. Lu and E. Milios. Robot pose estimation in unknown environments by matching 2D range scans. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition CVPR-94*, pages 935–938, 1994.

[66] M. Maimone and L. Matthies. Visual Odometry on the Mars Exploration Rovers. *2005 IEEE International Conference on Systems, Man and Cybernetics*, pages 903–910, 2005.

[67] D. Marr, E. Hildreth, L. Series, and B. Sciences. Theory of edge detection. *Proceedings of the Royal Society of London*, 207(1167):187–217, 1980.

[68] L. Matthies and S. Shafer. Error modeling in stereo navigation. *Robotics and Automation, IEEE Journal of*, 3(3):239–248, 1987.

[69] D. Mellinger, N. Michael, and V. Kumar. Trajectory Generation and Control for Precise Aggressive Maneuvers with Quadrotors. In *International Symposium on Experimental Robotics*, 2010.

[70] P. Merrell, A. Akbarzadeh, L. Wang, P. Mordohai, J.-M. Frahm, R. Yang, D. Nister, and M. Pollefeys. Real-time visibility-based fusion of depth maps. In *2007 IEEE 11th International Conference on Computer Vision*, pages 1–8. Ieee, 2007.

[71] Microsoft XBox. *http://www.xbox.com*, (accessed May, 2011).

[72] Middlebury Vision Group. *http://vision.middlebury.edu/stereo/*, (accessed June, 2011).

[73] Mission Oriented Operating System (MOOS). *http://www.robots.ox.ac.uk/~pnewman/TheMOOS*, (accessed June, 2011).

[74] A. R. Mitchell. *Computational methods in partial differential equation.* J. Soc. Indust. Appl. Math, 1955.

[75] A. R. Mitchell. *Computational Methods in Partial Differential Equations.* John Wiley & Sons Ltd, 1969.

[76] H. Moravec and A. Elfes. High resolution maps from wide angle sonar. In *Proc of the IEEE International Conference on Robotics and Automation*, volume 2, pages 116–121, 1985.

[77] H. P. Moravec. Robot Spatial Perception by Stereoscopic Vision and 3D Evidence Grids. *Perception*, 1(September), 1996.

[78] B. Morisset, R. B. Rusu, A. Sundaresan, K. Hauser, M. Agrawal, J.-C. Latombe, and M. Beetz. Leaving Flatland: Toward real-time 3D navigation. *2009 IEEE International Conference on Robotics and Automation*, pages 3786–3793, May 2009.

[79] K. Muhlmann, D. Maier, J. Hesser, and R. Manner. Calculating dense disparity maps from color stereo images, an efficient implementation. *International Journal of Computer Vision*, 47(1):79–88, 2002.

[80] E. Murphy. *Planning and Exploring Under Uncertainty.* PhD thesis, Oxford University, 2010.

[81] D. Murray and J. Little. Using real-time stereo vision for mobile robot navigation. *Autonomous Robots*, 8(2):161–171, 2000.

[82] R. Newcombe and A. Davison. Live dense reconstruction with a single moving camera. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1498–1505. IEEE, 2010.

[83] R. Newcombe, S. Lovegrove, and A. Davison. DTAM: Dense Tracking and Mapping in Real-Time. In *International Conference on Computer Vision*, page pp, 2011.

[84] P. Newman. *On the structure and solution of the simultaneous localisation and map building problem*. PhD thesis, University of Sydney, 1999.

[85] P. Newman. Under the Hood of the MOOS Communications API. Technical report, Mobile Robotics Group, Oxford University, 2009.

[86] P. Newman, G. Sibley, M. Smith, M. Cummins, A. Harrison, C. Mei, I. Posner, R. Shade, D. Schroeter, L. Murphy, W. Churchill, D. Cole, and I. Reid. Navigating, Recognizing and Describing Urban Spaces With Vision and Lasers. *The International Journal of Robotics Research*, 28(11-12):1406–1433, July 2009.

[87] D. Nistér, O. Naroditsky, and J. Bergen. Visual odometry for ground vehicle applications. *Journal of Field Robotics*, 23(1):3–20, Jan. 2006.

[88] I. Nourbakhsh and M. Genesereth. Assumptive Planning and Execution: a Simple, Working Robot Architecture. *Autonomous Robots Journal*, 1996.

[89] A. Nüchter, K. Lingemann, J. Hertzberg, S. Birlinghoven, and D.-S. Augustin. 6D SLAM 3D Mapping Outdoor Environments. *Journal of Field Robotics*, 24:699–722, 2007.

[90] NVIDIA Corporation. *http://www.nvidia.com/*, (accessed May, 2011).

[91] OC Robotics. Snake-arm Robots Access the Inaccessible. *Nuclear Technology International*, pages 92–94, 2008.

[92] OC Robotics. *http://www.ocrobotics.com*, (accessed April, 2011).

[93] OpenCV. *http://opencv.willowgarage.com/*, (accessed May, 2011).

[94] OpenGL. *http://www.opengl.org/*, (accessed June, 2011).

[95] A. Patel. Amit's Game Programming, 2009.

[96] P. Pfaff, R. Triebel, and W. Burgard. An Efficient Extension to Elevation Maps for Outdoor Terrain Mapping and Loop Closing. *The International Journal of Robotics Research*, 26(2):217–230, Feb. 2007.

[97] J. Pineau, G. Gordon, and S. Thrun. Anytime point-based approximations for large POMDPs. *Journal of Artificial Intelligence Research*, 27(1):335–380, 2006.

[98] Point Grey Research Inc. *http://www.ptgrey.com*, (accessed May, 2011).

[99] Point Grey Triclops SDK. *http://www.ptgrey.com/products/triclopsSDK*, (accessed May, 2011).

[100] E. Prestes. Exploration method using harmonic functions. *Robotics and Autonomous Systems*, 40(1):25–42, July 2002.

[101] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *IEEE International Conference on Robotics and Automation Workshop on Open Source Software*, 2009.

[102] T. Reenskaug. Models-views-controllers. Technical Report December, Xerox PARC, 1979.

[103] E. Rosten and T. Drummond. Fusing points and lines for high performance tracking. In *IEEE International Conference on Computer Vision*, pages 1508—-1511. IEEE Computer Society, 2005.

[104] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *ECCV*, pages 430–443. Springer, 2006.

[105] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 2009.

[106] T. Saad. Derivation of the Continuity Equation in Cartesian Coordinates, 2009.

[107] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision*, 47(1):7–42, 2002.

[108] Segway. *http://www.segway.com/*, (accessed May, 2011).

[109] R. Shade and P. Newman. Discovering and Mapping Complete Surfaces with Stereo. In *IEEE International Conference on Robotics and Automation*, pages 3910–3915. IEEE, 2010.

[110] R. Shade and P. Newman. Choosing Where To Go: Complete 3D Exploration With Stereo. In *IEEE International Conference on Robotics and Automation*, 2011.

[111] S. C. Shapiro. *Encyclopedia of Artificial Intelligence*. Wiley-Interscience publication. John Wiley & Sons, 1992.

[112] G. Sibley, C. Mei, I. Reid, and P. Newman. Adaptive relative bundle adjustment. In *Robotics Science and Systems Conference*, pages 1–8. Citeseer, 2009.

[113] G. Sibley, C. Mei, I. Reid, and P. Newman. Planes, trains and automobiles - Autonomy for the modern robot. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 285–292. IEEE, 2010.

[114] SICK. *http://www.sick.com*, (accessed May, 2011).

[115] R. Sim and J. Little. Autonomous vision-based robotic exploration and mapping using hybrid maps and particle filters. *Image and Vision Computing*, 27(1-2):167–177, Jan. 2009.

[116] R. Smallwood and E. Sondik. The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, pages 1071–1088, 1973.

[117] M. Smith, I. Baldwin, W. Churchill, R. Paul, and P. Newman. The New College Vision and Laser Data Set. *The International Journal of Robotics Research*, 28(5):595–599, May 2009.

[118] R. Smith, M. Self, and P. Cheeseman. Estimating uncertain spatial relationships in robotics. *Autonomous robot vehicles*, 1:167–193, 1990.

[119] SRI Small Vision Systems. *http://www.ai.sri.com/software/SVS*, (accessed May, 2011).

[120] C. Stachniss, G. Grisetti, and W. Burgard. Information gain-based exploration using rao-blackwellized particle filters. In *Proc. of robotics: science and systems (RSS)*, pages 65–72. Citeseer, 2005.

[121] S. Thrun. Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71, 1998.

[122] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.

[123] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-e. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. V. Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, S. Clara, and P. Mahoney. Stanley : The Robot that Won the DARPA Grand Challenge. *Journal of Field Robotics*, 23(April):661–692, 2006.

[124] H. R. Tiwary. On the Hardness of Computing Intersection, Union andMinkowski Sum of Polytopes. *Discrete & Computational Geometry*, 40(3):469–479, July 2008.

[125] C. Tomasi. Bilateral Filtering for Gray and Color Images. *IEEE International Conference on Computer Vision*, 1998.

[126] B. Tovar, L. Guilamo, and S. LaValle. Gap navigation trees: Minimal representation for visibility-based tasks. *Algorithmic Foundations of Robotics VI*, pages 425–440, 2005.

[127] R. Triebel, P. Pfaff, and W. Burgard. Multi-Level Surface Maps for Outdoor Terrain Mapping and Loop Closing. *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2276–2282, Oct. 2006.

[128] E. Trucco and A. Verri. *Introductory techniques for 3-D computer vision*, volume 93. Prentice Hall New Jersey, 1998.

[129] W. Van Der Mark and D. M. Gavrila. Real-Time Dense Stereo for Intelligent Vehicles. *Intelligent Transportation Systems, IEEE Transactions on*, 7(1):38–50, 2006.

[130] O. Veksler. Stereo Correspondence by Dynamic Programming on a Tree. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 384 – 390, 2005.

[131] D. Wettergreen, C. Thorpe, and R. Whittaker. Exploring Mount Erebus by walking robot. *Robotics and Autonomous Systems*, 11(3-4):171–185, Dec. 1993.

[132] P. Whaite and F. Ferrie. Active Exploration: Knowing When We're Wrong. In *Computer Vision, 1993. Proceedings., Fourth International Conference on*, pages 41–48. IEEE, 1993.

[133] F. M. White. *Fluid Mechanics*, volume 17. McGraw Hill Higher Education; 6th Revised edition edition, 2009.

[134] O. Woodford, P. Torr, I. Reid, and A. Fitzgibbon. Global Stereo Reconstruction under Second Order Smoothness Priors. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(12), 2009.

[135] K. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, 2010.

[136] B. Yamauchi. A frontier-based approach for autonomous exploration. *Proceedings 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation CIRA'97. 'Towards New Computational Principles for Robotics and Automation'*, pages 146–151, 1997.

[137] B. Yamauchi, A. Schultz, and W. Adams. Mobile robot exploration and map-building with continuous localization. *IEEE International Conference on Robotics and Automation*, pages 3715–3720, 1998.